NON-REFORMIST REFORM FOR HASKELL MODULARITY

by SCOTT KILPATRICK

UNIVERSITÄT DES SAARLANDES

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University.

Saarbrücken
October 2019

ABSTRACT

Module systems like that of Haskell permit only a weak form of modularity in which module implementations depend directly on other implementations and must be processed in dependency order. Module systems like that of ML, on the other hand, permit a stronger form of modularity in which explicit interfaces express assumptions about dependencies and each module can be typechecked and reasoned about independently.

In this thesis, I present Backpack, a new language for building separately-typecheckable *packages* on top of a weak module system like Haskell's. The design of Backpack is the first to bring the rich world of type systems to the practical world of packages via *mixin modules*. It's inspired by the MixML module calculus of Rossberg and Dreyer but by choosing practicality over expressivity Backpack both simplifies that semantics and supports a flexible notion of applicative instantiation. Moreover, this design is motivated less by foundational concerns and more by the practical concern of integration into Haskell. The result is a new approach to writing modular software at the scale of packages.

The semantics of Backpack is defined via elaboration into sets of Haskell modules and binary interface files, thus showing how Backpack maintains interoperability with Haskell while retrofitting it with interfaces. In my formalization of Backpack I present a novel type system for Haskell modules and I prove a key soundness theorem to validate Backpack's semantics.

ZUSAMMENFASSUNG

Modulsysteme wie die in Haskell erlauben nur eine weiche Art der Modularität, in dem Modulimplementierungen direkt von anderen Implementierungen abhängen und in dieser Abhängigkeitsreihenfolge verarbeitet werden müssen. Modulsysteme wie die in ML andererseits erlauben eine kräftige Art der Modularität, in dem explizite Schnittstellen Vermutungen über Abhängigkeiten ausdrücken und jeder Modultyp überprüft und unabhängig ergründet werden kann.

In dieser Dissertation präsentiere ich Backpack, eine neue Sprache zur Entwicklung separat-typenüberprüfbarer *Pakete* über einem weichen Modulsystem wie Haskells. Das Design von Backpack überführt erstmalig die reichhaltige Welt der Typsysteme in die praktische Welt der Pakete durch *Mixin-Module*. Es wird von der MixML-Kalkulation von Rossberg und Dreyer angeregt. Backpack vereinfacht allerdings diese Semantik durch die Auswahl von Anwendbarkeit statt Expressivität und fördert eine flexible Art von geeigneter Applicative-Instantiierung. Zudem wird dieses Design weniger von grundlegenden Anliegen als von dem praktischen Anliegen der Eingliederung in Haskell begründet.

Die Semantik von Backpack wird durch die Ausarbeitung in Mengen von Haskell-Modulen und „binary interface files“ definiert, und zeigt so, wie Backpack Interoperabilität mit Haskell erhält, während Backpack es mit Schnittstellen nachrüstet. In meiner Formalisierung Backpacks präsentiere ich ein neuartiges Typsystem für Haskellmodule und überprüfe einen entscheidenen Korrektheitssatz, um die Semantik von Backpack zu validieren.

The philosophers have only interpreted the [modularity] in various ways; the point, however, is to change it.

— Karl Marx, in the spirit of this work

ACKNOWLEDGMENTS

What's a dissertation, anyway? A professional certification? An assemblage of published and unpublished scientific ideas? An extraordinarily labor-intensive memento of frustration, elation, collaboration, and pride?

On the occasion of finishing this dissertation, perhaps a few years too late, the richness of my doctoral experience is palpable. With these acknowledgments I hope not only to recognize all those who contributed to it, but also to express that richness for posterity—for friends, family, and prospective PhD students alike.

Altogether I'm enormously grateful for the wealth of experience I gained in the Max Planck Society (MPG). Most immediately, that experience comes from my advisor, mentors, collaborators, and friends as a PhD student at the Max Planck Institute for Software Systems (MPI-SWS) in Saarbrücken, Germany. Every junior scientist would benefit, as I did, from the experience of living and working abroad in such a respected institution as the MPG. Moreover, I learned a great deal—and hopefully helped change some things for the better—by participating in the democratic representation of the interests of junior scientists in the scientific workplace, first at MPI-SWS and then as part of the Max Planck PhDnet. We shouldn't forget that the boundaries of science extend past technical and mathematical matters; it's also a self-organized profession, with all the material and social concerns that come with the territory of the workplace.

As for the people who played notable roles in my scientific development, allow me to enumerate them chronologically.

First and foremost, my parents—Nancy and Charlie—always loved and supported me, and encouraged me to pursue math and engineering, even though they had no idea what that meant past 10th grade or so. As an undergraduate at the University of Texas at Austin (UT), my teacher and then mentor Greg Lavender introduced me to functional programming, to the lambda calculus, and to independent study.

For introducing me to programming languages research, I'd like to acknowledge Eric Allen, my mentor and supervisor at UT and at Oracle Labs (then Sun Labs). After a rejection from all six PhD programs I had initially applied to, Eric's tutelage and support helped me turn things around throughout my masters. My colleagues at the Oracle Labs Programming Languages Research Group provided an encouraging and responsive atmosphere where I got my feet wet with research: Eric, Steve Heller, Guy Steele, Sukyoung Ryu, Victor Luchangco, David Chase, and, in particular, my friend and fellow intern (and now mathematician) Justin Hilburn.

For introducing me to MPI-SWS, I have my Dumbledore from UT to thank—Lorenzo Alvisi, who told me about the institute and the whole MPG. Working as his TA for graduate distributed computing, as a mere masters student, was one of the most challenging and intimidating experiences of my career. His belief in me to take on that challenge, like Greg Lavender's, had an immense impact on the beginning of my career. As an immigrant scientist himself, Lorenzo also encouraged me to pursue a PhD overseas. (Good advice!) As a periodic visitor to MPI-SWS he continued his Dumbledore role in my life for years.

The most critical figure to acknowledge is my PhD advisor at MPI-SWS and friend, Derek Dreyer. His attention to detail, pursuit of simplification, and affable demeanor have certainly left a lasting impression on me. Although my academic career will likely not continue, I have benefited tremendously from the writing and presentation skills I learned from Derek and from his partner Rose Hoberman, the "soft skills" (more generally useful skills!) instructor at MPI-SWS. Derek also supported me anytime I ruffled feathers in my role as student representative to the faculty, even when he disagreed; I hope he recognizes the emotional weight of that. He also introduced me to many great films and very often, along with Rose, gave me a good meal.

Among the many PhD students, postdocs, and other colleagues at MPI-SWS I enjoyed knowing and working with, I'd like to acknowledge a few in particular, people who left some mark on my scientific career. Beta Ziliani was my "older brother" in Derek's group, a wonderful friend, a fun colleague to brainstorm with, a Coq expert, and a peerless *Grillmeister*. Georg Neis, another student of Derek's, helped with a few proofs but also taught me some Saarlandish culture. Derek's postdoc Neel Krishnaswami was a constant source of PL wisdom and stimulating conversation. External scientific member—and my academic grandparent—Bob Harper was always a joy to talk to and learn from while visting the institute. From MPI-SWS I'd also like to acknowledge conversations with and other valuable input from Ezgi Çiçek, Allen Clement, Natacha Crooks, Nancy Estrada, Pedro Fonseca, Deepak Garg, Roly Perera, and Aaron Turon. Plenty of others, many friends among them, helped to make the institute a stimulating place to work and study for years. Staff members Mary-Lou Albrecht, Brigitta Hansen, Chris Klein, Claudia Richter, and Carina Schmitt all provided key support during my time there as well. Outside the institute, my friend and would-be PhD sibling Michael Greenberg gave me encouragement, technical feedback, and a personal goalpost for years.

The early days of the Backpack project involved multiple visits to Microsoft Research Cambridge to work with my collaborators Simon Peyton Jones and Simon Marlow. From Simon PJ I learned a valuable life skill: the confidence to say "I have no idea what's going on," a rare skill that has been crucial to my career inside and outside academia. I'm very lucky to have written a paper with him. Also at MSR Cambridge, whiteboard discussions and other chit-chats with fellow module systems expert Claudio Russo were a memorable part of my visits, as was his gift of a pilfered copy of *The Commentary on Standard ML*. Later on, Edward Yang made perhaps the most significant contribution to the Backpack research project: taking it and running with it in a big way! His enthusiasm and continued shepherding of Backpack helped keep me interested in the work.

One additional person deserves acknowledgment for technical contributions to my research: Andreas Rossberg. Although his postdoc work with Derek predated my time at MPI-SWS, he spent years as a remote mentor of sorts, someone to whom I could always send hyper-specific module systems ideas and expect a detailed and thoughtful response. His careful and thorough review of this dissertation undoubtedly improved it; I am very grateful for and humbled by that contribution.

Outside of technical collaboration, my involvement in the Max Planck PhDnet added to the richness of my academic experience. Martin Grund and Adrin Jalali were both friends and key collaborators in that aspect of my scientific career. Also non-technical but still important was the logistical help submitting this dissertation remotely from my friend Mohamed Omran. And Birgit Adam touched up my (surely indelicate) German translation of the dissertation's abstract, the *Zusammenfassung*.

Finishing this dissertation and my PhD overall, three years after leaving MPI-SWS to start my software engineering career, was an emotionally complicated prospect. Three people are primarily owed thanks for helping me with that. First, my therapist, Chris Bandini. Every graduate student needs one! Second, my erstwhile colleague two times over, Steve Heller. After helping me make a plan for splitting my time between work work and dissertation work, he spent a couple years casually prodding me about it. Finally, my wonderful wife, Alanna Schubach, who wasn't in the picture when I started this dissertation but is now the most essential figure in its completion. I truly could not have done it without her.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Haskell's module system emerged with surprisingly little debate. At the time, the sophisticated ML module system was becoming well established, and one might have anticipated a vigorous debate about whether to adopt it for Haskell. In fact, this debate never really happened.

Paul Hudak *et al.* (2007), "A History of Haskell: Being Lazy With Class"

A spectre is haunting Haskell—the spectre of modularity.

Specifically, that spectre is Backpack, a new extension to the Glasgow Haskell Compiler (GHC) and the Cabal package management system.[1] Backpack addresses a longtime deficiency of the Haskell programming language and package ecosystem: Haskell's lack of *strong modularity*, particularly at the level of separately-developed packages. With Backpack, strong modularity is finally available in Haskell, and the entire Haskell ecosystem of packages stands to benefit from it. There now exist new ways of expressing modular Haskell programs and libraries, with new potential for modular static checking, new modular programming idioms, and hints at new avenues of development. Overall, Backpack is a leap forward for Haskell, but it's also a leap into the unknown.

Backpack exists today as an engineered artifact. That exciting fact is a testament to the work of my eventual collaborator Edward Yang, whose Ph.D. thesis presented a concrete implementation of Backpack in GHC, along with partially revised semantics to comport with that implementation.[2] As a precursor to the concrete implementation and to Yang's work, this thesis presents the original design and formalization of Backpack that was the subject of the *POPL'14* paper,[3] along with a new formalization of Haskell type classes.

The title of the *POPL'14* paper, "Backpack: Retrofitting Haskell with Interfaces," conveyed the essential idea in Backpack of *retrofitting* the existing Haskell language with a new form of modularity. Retrofitting refers both to building modularity on top of the existing Haskell language, rather than replacing it, and to the research challenge of formalizing a module system for a *real* programming language that's actually in use—Haskell—instead of inventing yet another ML-like system that only exists on paper.

In contrast to the original paper's title, "Non-Reformist Reform for Haskell Modularity" is the title of this thesis. The political concept of non-reformist reform encompasses not only the idea of retrofitting something existing but also the idea of building toward a more definitive break with the existing world. It captures an additional goal of Backpack: to import, technically and conceptually, the more idealized world of *strong* modularity in the ML family of languages into the more ubiquitous world of *weak* modularity in Haskell and many other languages. This non-reformist reform sits somewhere between a revolutionary approach—*e.g.*, designing a new Haskell from scratch, this time starting with strong modularity—and a *reformist* reform approach—*e.g.*, adding a new GHC language extension or a new Cabal directive in order to obtain a more expressive weak modularity.

1 `https://ghc.haskell.org/trac/ghc/wiki/Backpack`
2 Yang (2017), "Backpack: Toward Practical Mix-in Linking in Haskell".
3 Kilpatrick *et al.* (2014), "Backpack: Retrofitting Haskell with Interfaces".

1.1 MODULARITY

Modularity is one of those concepts in programming languages that has countless definitions and connotations. An uncontroversial definition would be that it's a property of a programming system in which programs can be broken up into different components that define *and maintain* their own abstractions. That latter part of the property is where the importance of *type systems* comes in. As per John Reynolds's classic definition, "type structure is a syntactic discipline for enforcing levels of abstraction."[4]

In this thesis I don't try to pin down a more precise definition of modularity. I do, however, cast some light on different considerations of modularity in order to motivate the general approach of Backpack: the *strength* and the *scale* of modularity.

The idea behind these two attributes of modularity is not to categorize whole programming languages but to understand and characterize the modularity offered by their constituent language features. What kind of modularity is offered by Haskell modules? Or by Haskell packages? How does that modularity differ from that of ML modules? My characterization of modularity in these terms helps distill the essential differences and pin down what's novel about Backpack.

1.1.1 *Strength of Modularity*

What makes a programming language have *strong* as opposed to *weak* modularity? I've hinted that the ML family of languages, like Standard ML (SML)[5] and OCaml[6], offers the former while Haskell[7] and most other languages offer the latter. Boiled down to its essence, the difference is best captured by whether there's any facility for *abstraction* over program components, typically expressed through *signatures*. The presence of such abstraction indicates strong modularity. In weak modularity, on the other hand, concrete program components may only depend on other concrete program components without any abstraction over their implementations.

STRONG MODULARITY The *Definition of Standard ML*[8] perfectly summarizes strong modularity in the module system of SML:

> The principle of inferring useful structural information about programs is also represented, at the level of program modules, by the inference of *signatures*. Signatures describe the interfaces between modules, and are vital for robust large-scale programs. When the user combines modules, the signature discipline prevents him from mismatching their interfaces. By programming with interfaces and parametric modules, it becomes possible to focus on the structure of a large system, and to compile parts of it in isolation from one another—even when the system is incomplete.

Client-side data abstraction through parameterization; implementor-side data abstraction through sealing.

This software philosophy has served the ML family of languages for decades. Signatures, along with the matching of implementations against them, constitute a principal language feature in the module systems of the ML family, in particular SML and OCaml. They enable both "client-side data abstraction," in which a *parameterized module*—also called a *functor*—uses a signature to abstract over the implementation of some other module it depends on, and "implementor-side data abstraction," in which an implementation—also called a *structure*—hides away or "seals" its particular details so that only the given signature is visible to clients.

For the purposes of this thesis, it's the first use case of signatures—client-side data abstraction—that illustrates strong modularity. In particular, in the ML family, strong modularity

4 Reynolds (1983), "Types, Abstraction and Parametric Polymorphism".
5 Milner *et al.* (1997), "The Definition of Standard ML (Revised)".
6 Leroy *et al.* (2017), "The OCaml System release 4.06: Documentation and user's manual".
7 Hudak *et al.* (2007), "A history of Haskell: Being lazy with class".
8 Milner *et al.* (1997), "The Definition of Standard ML (Revised)".

comes in the form of functors. As an example, consider the following `list`-based implementation of a finite set of elements in SML. It's a functor, `Set`, that abstracts over the type and structure of elements via the interface `ORDERED_TYPE`:

```
functor Set (E: ORDERED_TYPE) :> SET =
  struct
    type element = E.t
    type set = element list
    let empty = []
    let rec add x s = ... E.leq ...
    let rec member x s = ... E.leq ...
  end
```

But not all ML variants support functors. For example, the popular Microsoft .NET-based F♯[9] and the nascent CakeML[10] both eschew the strong modularity of functors as part of their own technical tradeoffs, resulting in weak modularity. Though considered weak, these languages nonetheless offer signatures and therefore implementor-side data abstraction; their modularity isn't quite strong, but it's not fully weak either.

WEAK MODULARITY In the world of weak modularity, modules cannot abstract away their dependencies; implementations may only depend on other implementations. This is a pretty conventional view of modules. Each module defines some stuff, partly by using stuff that it "imports" from other modules. For example, consider a Haskell module called Server that imports a socket data structure from another module called Socket.

With weak modularity, processing a module, *e.g.*, type-checking it, requires first processing the imported modules in order to understand the specifications of what was imported. Because the imported modules must already exist in order to be so processed, there exists a rigid order in which modules may be processed.[11] And not just the order of processing, but even the order of development: the downstream modules must be developed *after* the upstream ones. In the example, developing and reasoning about the Server module can only occur after the Socket module has been fully fleshed out and implemented.[12]

In a strong modular system, there would be a facility for expressing the expected interface of the imported modules, with implementations of those modules linked into the program later, so that this module can be processed independently of the modules eventually linked. Server would instead depend on some socket interface that specifies the socket data structure, and only when Server's import is linked with Socket is the program considered complete.

Another drawback to weak modularity is that a module cannot be reused, within the same program, by depending on related but different modules. How can the Server module be reused with different implementations of sockets? For example, to test it against a mock implementation of sockets. Since the dependency goes from implementation to implementation, there's no way to break it and move it to a different one.

Aside from this standard notion of modules, weak modularity regularly occurs in another form—as *packages* in *package management systems*. In such systems, a *package* represents a bundle of source files, with build-time dependencies on other, externally defined packages, that bears a unique name and version number. This system of structuring (open-source) programs spans various software ecosystems, from language-agnostic systems like Debian's APT[13] to systems specific to a particular programming language like Haskell's Cabal/Hackage.[14]

Packages are a form of weak modularity, despite indefinite dependencies.

9 Syme (2012), "The F♯ 3.0 Language Specification".
10 Myreen and Owens (2014), "Proof-producing translation of higher-order logic into pure and stateful ML".
11 Some systems of weak modularity allow for module import dependency to be cyclic, thereby defining recursive modules. In some systems with cyclic dependency there still exists a rigid order imposed—for example, (GHC) Haskell, whose recursive modules will be the subject of much discussion in §1.2.4. In other systems, like mutually recursive classes in Java, there is no rigid order imposed. In all cases, however, implementations depend on implementations.
12 If one really wishes to develop Server first, one would likely write "stub implementations" for Socket's definitions.
13 https://wiki.debian.org/Apt
14 https://hackage.haskell.org/

In package management systems, packages are implementations that depend on other implementations—weak modularity. Generally this dependency is expressed as a package name and a range of version numbers; *e.g.*, `socketlib >= 2.1` is a dependency on the package (rather, series of package definitions) named `socketlib`, but only those implementations marked version `2.1` or later. A package with this dependency must be built and linked against any such implementation version of `socketlib`, for the internal entities defined in the package depend on entities defined in `socketlib`.

While such version ranges *appear* to abstract over multiple implementations, thereby transcending weak modularity, they do not enable the independent development of the client package. If package `myserver` version `1.2.3` depends on `socketlib >= 2.1`, because `myserver`'s `Server` entity uses `socketlib`'s `Socket` entity, then `myserver` cannot be processed or built independently of *a particular implementation* of `socketlib`, be it `2,1`, `2.5`, or `4.0`, with particular implementations of its own dependencies in turn.

My language of strong vs. weak modularity[15] is intended to connote the conventional distinction between strong vs. weak *typing* in programming languages. With *strong* typing, one has stronger guarantees about the runtime behavior of a program, *e.g.*, the absence of calling a function with the wrong type of argument; with strong modularity, one has stronger guarantees about the runtime or even link-time behavior of a module, *e.g.*, the absence of linking a dependent component with the wrong type of implementation. On the other hand, with *weak* typing and with *weak* modularity, one has weaker guarantees—or even no guarantees—about such behavior.

1.1.2 *Scale of Modularity*

The concepts of "programming in the large" vs. "programming in the small" come from decades of folklore and vague generalities, a lot like "modularity." In the types literature, MacQueen described the former as "concerned with using [modularity constructs for information hiding] to impose structure on large programs," in contrast to the latter, which "deals with the detailed implementation of algorithms in terms of data structures and control constructs."[16] In this thesis I adapt these concepts as the *scale* of modularity: *small* vs. *large*.

SMALL MODULARITY Small modularity describes language constructs which capture fine-grained data abstraction, typically regarding the implementation of data structures and the enforcement of their invariants across abstraction boundaries. These constructs are referred to in the course of programming and *execution* at the core level of a language, and, in *strong* cases, are reused/instantiated multiple times in multiple distinct ways within a (top-level, fully-closed) program. Such constructs are sometimes considered the "unit of development," "typechecking," and "reasoning." In languages with only *weak* modularity, the "unit of development" is taken to be the "unit of compilation" as well, a unification of the two granularities that confers *small* rather than large modularity.

Examples of small modularity include ML modules like the ones above, Haskell modules, Java classes and interfaces, and C++ classes. All of these forms of modularity define and enforce data abstraction to varying degrees. In particular, in the case of Haskell modules, even though they have the dual role of both reasoning and compilation, they are fundamentally part of data abstraction and the type system of the core language.

LARGE MODULARITY Large modularity, on the other hand, describes language constructs which aggregate the smaller modular constructs into a coherent whole that will exist in a fixed form, as source code, to be distributed across systems. At this scale, modularity is

15 Conventionally, the terms *definite* vs. *indefinite* references are used to make this distinction. The problem with these terms is that they refer not to a characteristic of a language's modularity but to particular expression forms of dependency. This isn't the correct distinction to make; for example, virtually any language supporting strong modularity nonetheless allows for both definite and indefinite dependencies on other components.

16 MacQueen (1986), "Using Dependent Types to Express Modular Structure".

more concerned with the organization of code dependencies and less concerned with execution; consequently, it's primarily the domain of static semantics and not dynamic semantics. Reuse/instantiation within a single (top-level, fully-closed) program is limited. Such constructs might be considered the "unit of distribution," "authorship," or "versioning" of a language. Examples of large modularity include Haskell packages, OCaml's interpretation of source files into modules (via `ocamlc`), SMLSC's units,[17] and generally any package management system, like OCaml's OPAM[18] and Haskell's Hackage/Cabal.[19]

The dominant form of large modularity comes in the form of *packages*, as described earlier. Such systems have arisen out of the increasing need for distributed respositories of (open-source) modular, interdependent components as software systems evolve in that direction.

Large modularity cannot be described as merely a particular mode of use of small modularity. Essentially, the way small modularity is expressed and abstracted does not conveniently match the use cases of large modularity. The following discussion of *abstraction barriers*, *instantiation*, and, in the case of strong modularity, "fully-functorized style" (§1.2.2), provides weight to that claim.

ABSTRACTION BARRIERS Since modularity is about organizing, defining, and maintaining abstractions, the *abstraction barriers* that circumscribe them help illustrate the differences in scale.

In small modularity, an abstraction barrier delineates tight specifications about abstract types and the invariants that operations enforce about them—in other words, data abstraction. Think of the example of the `SET` functor in ML. The set structure (the body of the functor) depends on an unknown ordering structure (the parameter of the functor). There's an abstraction barrier isolating the set structure. From inside it, the set structure knows only the interface of the ordering structure (the `ORDERED_TYPE` signature): the abstract type `t` and the invariant that the `leq` function is a total order on `t`.[20] And from outside the abstraction barrier, clients of the functor know only the interface of the set structure (the `SET` signature), not its internal implementation (the representation and operations in terms of `list`). To clients, the abstraction barrier enforces certain invariants about the abstract set type: for example, any value of type `set` only contains values of type `elt` and moreover must have been constructed using the structure's `empty` and `insert` operations.

Abstraction barriers in small modularity delineate tight specifications about data.

In contrast to imposing data abstraction, abstraction barriers in large modularity serve primarily to delineate known vs. unknown components—mine vs. yours—in an *open* program. An open program is typically one which has free, unbound variables to be substituted, *i.e.*, stuff to be filled in. In the context of modularity, an open program is one which has a dependency on unknown components to be linked in (*cf.* Cardelli (1997)). Consider the previous example of the `myserver` package. On one side of the abstraction barrier are the package requirements (the Socket module) and on the other side the package's provisions (the Server module) for clients of the package to import and use. The separation between *requirements* and *provisions* is exactly the separation between "yours" and "my" components.

Abstraction barriers in large modularity delineate known vs. unknown components.

INSTANTIATION Another way to illustrate the distinction between small and large modularity is to look at how modular components are *instantiated*. This analysis only concerns those indefinite modular components which are abstractly defined with respect to some kind of unknown parameter—functors and packages, not Haskell modules, for instance.

Small modularity is geared toward easy reuse through multiple instantiation. In the example of small modularity, the `Set` functor might be instantiated multiple times in the same program, for different element types or even different orderings on the same element type. The syntax of functors makes such reuse straightforward, as intended: two different uses of `Set` can be expressed as `Set(IntOrd)` and `Set(StringOrd)`, which in turn define two distinct abstract `set` types.

17 Swasey *et al.* (2006), "A separate compilation extension to Standard ML".
18 Gazagnaire *et al.* (2018), "The OCaml Package Manager 2.0: The opam manual".
19 Coutts *et al.* (2008), "Haskell: Batteries Included".
20 The total order invariant is not expressible in the type system of ML, but the functor body still presumes it's true.

Instantiation in small modularity is like substitution. Instantiation in large modularity is like linking.

Large modularity, on the other hand, does not make it easy to instantiate the same indefinite component multiple times in different ways. That's because the unknown "parameter" in large modularity is not really a parameter to be substituted but an open dependency to be linked. The body of the `Set` functor is meant to be substituted with any and every implementation of `ORDERED_TYPE` thrown at it, whereas the `server` package just needs *some* implementation of `socketlib` to be linked in. One can squint at that distinction and see universal vs. existential quantification, respectively, but it's more of a conceptual guide than a formal observation.

* * *

With the nebulous concept of modularity fleshed out into these attributes of strength and scale, we can now better appreciate what Backpack brings to the world of modular programming: strong, large modularity. Backpack has *strong modularity* because it allows for implementations to depend on interfaces (as well as multiple instantiation of these abstract implementations). Backpack has *large modularity* because its components, *packages*, organize constituent modules into domains of "mine vs. yours," allowing for the (typed) abstraction over the unknown "yours" via interfaces.

1.2 SURVEY OF MODULARITY

In the remainder of this chapter, I provide some deeper background on modularity in real programming. This survey of modularity "in the wild" is intended to give the reader an even firmer footing in what constitutes modularity, along with a sense for what language features, ideas, and research directions I have drawn from in the design and formalization of Backpack.

1.2.1 *Modularity in Haskell*

The first form of modularity in this survey is that of Haskell. Its relevance is clear.

As already described in §1.1, Haskell modules are a form of weak, small modularity.[21] Weak because implementations depend on implementations, and small because modules are more about units of compilation and reasoning than about units of distribution. Haskell modules are simply not intended to play as large a role in the static semantics of the language as were ML modules. Instead, the module system in Haskell plays two roles.

- First, modules carve up large programs into units of compilation and reasoning. Each Haskell source file defines a module, and modules are "processed"—type-checked, reasoned about, and compiled—in order of their dependency on one another. That makes them a form of incremental modular development and weak modularity.
- Second, modules organize core-level entities and give them unique identities. Those core level *entities* are comprised of types, values, datatypes, data constructors, type classes, type class instances. Each module imports some entities from other modules, defines new entities, and exports some of those combined entities for use by other modules. Each core entity is uniquely identified by its entity name (a syntactic name except for type class instances) and the name of the module that originally defined it.

As some of the Haskell designers have explained, "we eventually converged on a very simple design: the module system is a namespace control mechanism, nothing more and nothing less. This had the great merit of simplicity and clarity—for example, the module system is specified completely separately from the type system—but, even so, some tricky corners remained unexplored for several years."[22]

21 This characterization should be interpreted not as pejorative but as indicative of the ways in which Backpack improves on Haskell modularity.
22 Hudak *et al.* (2007), "A history of Haskell: Being lazy with class," §8.2.

DATA ABSTRACTION IN HASKELL Haskell modules do offer a limited form of implementor-side data abstraction: the implementor chooses to expose only certain entities from a module implementation. For example, the analogue in Haskell of ML's `Set` functor that creates an abstract type for sets is a `Set` datatype whose term constructors aren't exported; clients can use the type but have to use the provided `Set` operations rather than constructing or destructing values of that type. See Figure 1.1 for the code for this example, albeit specialized to sets of integers, an abstract type `IntSetT`.

A noteworthy difference between abstract types in Haskell and in ML is that in Haskell they can only be created as fresh, named datatypes. In ML, on the other hand, abstract types can be introduced without creating new core-level data types; for example, the abstract type `set` in the Set functor example was defined simply as an application of the `list` type.

Haskell type classes offer client-side data abstraction. More on that below.

PROGRAMS IN HASKELL A "program" in Haskell is a collection of module definitions, one of which is considered the "main" module and exports a value `main` of type `IO a`. A program could be a single module or ten or a thousand; the modules could be defined in a local directory and written by the user, or they could be defined in a *package* and stored in some repository cache (§1.2.6), or a mixture of both.

A program exists to be executed, by executing the `main` method, but first must be statically processed. Such processing occurs in dependency order: if module B imports module A, then the latter must be processed first. This chain of imports might yield a cycle, in which case the modules are said to be *recursive*; that will be explained in §1.2.4.

In the case of the de-facto standard compiler, GHC, the compilation of a module source file yields a "binary interface file" describing, in binary representation, all of the static contents of the module, *i.e.*, the names and specifications of all entities imported into, defined by, and exported out of the module. When compiling module B, it processes the import of module A by reading in A's binary interface file, which it must already have synthesized.

TYPE SYSTEM FOR MODULES Is there a *type system* for Haskell modules? Not really, at least not one defined in the Haskell Language Report,[23] which does describe, to a limited extent, a type system for Haskell's core language, however.

In the case of GHC, there are those binary interface files. Each one acts a little like the "type" of a module. A model of GHC might involve a typing judgment whose hypothetical context contains mappings from (already-processed) module names to their types and whose main "term : type" classification comprises a module's definition and its resulting type. (Indeed, this is the essence of the module type system defined by Backpack.) But this is merely a conceptual model, not something formally described.

What might a module type look like? As mentioned above, GHC's binary interface files indicate the structure of a module type: static specifications of all the entities the module concerns. That means a module type would be composed, at some level, of constituent *core-level* types. For example, regarding the modules defined in Figure 1.1, the type of the IntSetM module would necessarily involve the type of the `empty` value.

And how might that value's type be expressed? Syntactically, it's `IntSetT`, but there could be plenty of other types in the program with that same syntactic name. The answer is to identify the type with its "original name,"[24] *i.e.*, the syntactic name of the type paired with the unique name of the module that defined it; in this case that'd be something like "IntSetM.`IntSetT`". As we will soon see, Backpack generalizes original names into "physical names" and module names into structured "module identities," the latter being one of Backpack's technical contributions.

23 Marlow (2010) and Peyton Jones (2003).

24 Rather frustratingly, the concept of original names of entities is not defined anywhere in the Haskell Language Report, nor does the Report explain equality on entities, though it refers to such a notion. In stray documentation from older specifications of the language, in the documentation and implementation of GHC, and in the semantics of Faxén (2002, §2.3) the term "original name" is used as defined here.

```haskell
-- IntSetM.hs
module IntSetM
    ( IntSetT(), -- Expose IntSetT and the 'public'
      empty,     -- operations, but not the OfList
      add,       -- constructor or private function split.
      member ) where

  -- sets as lists, to be kept in order by the operations
  data IntSetT = OfList [Int]

  empty :: IntSetT
  empty = OfList []

  -- operation that maintains the order
  add :: Int -> IntSetT -> IntSetT
  add n (OfList ns) = OfList $ case split n ns [] of
      (nsLess, [])   -> nsLess ++ [n]
      (nsLess, m:ms)
          | n == m    -> ns     -- n is already present
          | otherwise -> nsLess ++ [n, m] ++ ms

  member :: Int -> IntSetT -> Bool
  member n (OfList ns) = (not $ null nsGeq) && n == head nsGeq
      where (nsLess, nsGeq) = split n ns []

  -- private function to split list into (< n), (>= n)
  split :: Int -> [Int] -> [Int] -> ([Int], [Int])
  split _ []     accumLess = (accumLess, [])
  split n (m:ms) accumLess
      | n <= m    = (accumLess, m:ms)
      | otherwise = split n ms (accumLess ++ [m])
```

```haskell
-- Client.hs
module Client where
  import IntSetM -- imports IntSetT, empty, add, member

  -- Client must build IntSetT using add and empty, not Tip and Bin,
  -- so we are guaranteed to maintain the abstract type's invariant
  -- that the IntSetT is a balanced binary tree.
  s :: IntSetT
  s = add 5 (add 10 empty)
```

Figure 1.1: Simple example of defining an abstract data type in Haskell: the `IntSetT` type.

STATIC SEMANTICS ONLY Unlike modules in ML, modules in Haskell are purely static things: there is no notion of evaluation or reduction for Haskell modules. This shouldn't be surprising since, unlike in ML, there is nothing resembling functions or function applications, the elemental beta conversion, at the level of modules. Although the Haskell Language Report describes modules and how they interact with the core-level of the language, it doesn't provide a succinct, formal definition. A formal static semantics of Haskell modules has been defined, to varying degrees, by Faxén (2002) and by Diatchki *et al.* (2002), although only the former establishes a type system, of sorts, for modules.

TYPE CLASSES Some might consider Haskell's type classes as a form of modularity. After all, type classes provide a way for program components to "define and maintain their own abstractions" about data types. That makes them a form of *small* modularity—perhaps even "micro" modularity, since they are not able to structure or organize the Haskell core language; for instance, type classes cannot organize other type classes.

Type classes are moreover a form of *strong* modularity since they offer a way to define implementations with respect to abstract interfaces. These are all values of *qualified type*, *i.e.*, generic values with type class constraints. Moreover, this abstraction is not merely at the value level: with *associated data types*[25] even data types, using a GHC Haskell extension, can be parameterized by type class instances. Modules, however, cannot be.

Although I consider type classes to be a form of strong modularity, the "programmer's contract" is different between ML modules and Haskell type classes. In the strong modularity of ML modules, a client program abstracted over a module `M : S` is agnostic to the particular implementation of `M` because the contract of ML data abstraction concerns the *modular* notion of *representation independence*: "it doesn't matter which implementation of `S` is linked in."[26] In the more limited strong modularity of Haskell type classes, on the other hand, a client program abstracted over a type class constraint `C a` is agnostic to the particular implementation (instance) of `C a` because the contract of type classes concerns the *antimodular* notion of *global uniqueness of instances*: "there can be only one implementation of `C a` anyway."[27] Both of these contracts are an improvement on that of weak modularity, which only expresses "I need that implementation, that one, there."

More tellingly, type classes do not offer any kind of implementor-side data abstraction (*e.g.*, sealing in ML) since the implementations (instances) are not expressible or identifiable anyway. So although I consider type classes to be a form of strong modularity for the purposes of this thesis—because they support client-side data abstraction, the focus of Backpack—the modularity they offer is nonetheless weaker than that of ML modules.

Type classes were not considered in the original Backpack work.[28] This thesis corrects that conspicuous omission by incorporating type classes and instances into the Backpack formalization. This is the subject of Chapter 4.

1.2.2 *Modularity in ML*

Next in the survey is modularity in ML. Again, the relevance to this thesis should be clear: one of the primarily goals of Backpack is to bring the strong modularity (and research tradition) of ML into Haskell.

A TRUE MODULE LANGUAGE Across all languages in the ML family, modules structure and organize programs in fundamentally different ways in ML than they do in Haskell. Rather than extra-linguistic constructs tied to source files, ML modules have their own ex-

25 https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/type-families.html#assoc-decl
26 See Mitchell (1986) for an early formulation of representation independence with respect to System F; Mitchell and Harper (1988) for a discussion of the property in ML; and Crary (2017) for a contemporary formulation of such abstraction properties in an ML-like module language.
27 Much more will be said about global uniqueness in §4.1 and in the Backpack formalization.
28 Kilpatrick *et al.* (2014), "Backpack: Retrofitting Haskell with Interfaces".

```
module SetExtension
    (S : (functor E : ORDERED_TYPE -> SET))
    (E : ORDERED_TYPE) : SET
  = struct
      module Set = S(E)
      open Set
      let closure f s = ...
    end
```

Figure 1.2: Example of a higher-order functor, adapted from Biswas (1995). `SetExtension` returns a new `SET` implementation that extends the given `SET` functor (like the earlier example) with an additional value component: the `closure` function that computes the closure of a given `set` value under the given `f`.

pression language. As such, there is syntax for introducing module expressions (structures and functors) and for binding variables to them with lexical scope.

Structures are modules that contain named type and value components, a bit like Haskell modules. But unlike Haskell modules, they are hierarchical: a structure can have named components that are themselves modules.

Functors abstract structures over other structures, creating a parameterized module that can be instantiated with other modules.

Signatures describe module types and can be used for encapsulation and for parameterization.

In this thesis, ML modules are divided into the two categories of *structures* and *functors*, both of which are closed under this union in the following sense:[29]

$$\begin{array}{lcl} \mathit{Module} & ::= & \mathit{Structure} \mid \mathit{Functor} \\ \mathit{Structure} & ::= & \{\, \overline{\mathit{Label} \mapsto \mathit{Module}} \,\} \\ \mathit{Functor} & ::= & \lambda\,(\mathrm{X} : \mathit{Signature})\ .\ \mathit{Module} \end{array}$$

Structures and functors don't have nested *structure* components; they more generally have nested *modules*. Structures can contain nested functors, and functors can be parameterized by functors. The latter are known as *higher-order functors*, as opposed to first-order functors that only abstract over structures. Figure 1.2 contains an example higher-order functor in OCaml. Higher-order functors are not universal among ML dialects. In particular, SML as formally defined lacks them though OCaml supports them.[30]

DATA ABSTRACTION IN ML The module language sits atop the core language of ML. Unlike in Haskell where it was merely there to provide "original names," the module language plays a large role in the type system of the core language. In particular, ML's module language offers rich forms of data abstraction for its core language.

ML modules support "implementor-side data abstraction": a structure can be sealed against a signature to encapsulate or hide some of its components. Moreover, the presence of type components can be exposed while hiding their definitions, effectively producing an *abstract type* whose actual representation is unknown outside the seal. This creation of abstract types is the essence of data abstraction in ML. We saw this in §1.1 in the body of the `Set` functor, which created the abstract type `set`.

29 To better elucidate the point about the module language here, the core-level type and term components are omitted from the structures in this pseudo-grammar.

30 The reason an ML dialect would omit them is because capturing them in the type system properly is quite tricky. In particular, that complexity stems from the possibility of parameterized functors producing abstract types when applied in the body, making the applicative semantics of functors ideal for the higher-order case. The module systems research literature saw many attempts at semantics for higher-order functors over the years; see Rossberg *et al.* (2014, p. 600) for a summary.

Functors inherently offer "client-side data abstraction": the body of the functor, considered the client module, abstractly depends on the parameter of the functor, knowing only the designated signature of that parameter. The functor can then be applied to any argument module that matches the signature, inducing a dependency of the client module on that argument module—a dependency which is abstracted to only the signature, not the full implementation of the argument module. Again, the `Set` functor from §1.1 exemplified client-side data abstraction, as does any functor.

STANDARD ML AND OCAML The ML family includes multiple languages, most notably Standard ML and OCaml. The module systems of both languages exhibit all the features described in this section, albeit with a key difference in how abstract types are created as the result of functor application.

In SML, functor application has "generative semantics," which means that each functor application yields a fresh abstract type for each opaque type component in its body. Each instantiation of a functor, even to the same argument module, yields a distinct abstract type. With the `Set` functor, this means that every application `Set(M)` will produce a fresh `set` type in the result, even when applied to the same argument twice:

```
structure S₁ = Set(IntOrd)
structure S₂ = Set(IntOrd)
let s = S₁.add 5 (S₂.empty) (* ill-typed! *)
```

Until more recent versions, in OCaml, functor application had only "applicative semantics," which means that each application of a functor to the same module argument produces the same abstract types for the functor body's opaque type components. This semantics better models *functions*: the results of two equivalent function applications are equivalent. However, different implementations (and formal models) of applicative semantics have different interpretations of how to judge functor arguments to be "the same." For example, OCaml determines functor arguments to be equivalent, for the purposes of abstract types produced by the functor, if they're both the same path reference to the same module variable.

More recently, however, OCaml offers generative semantics of functors with a slightly alternative syntax.[31] Indeed, both functor semantics have their uses. Applicative semantics may better model the behavior of functions, but for some functors, a side effect[32] of creating fresh abstract types on every application—even to the same module argument—is actually desired. For example, a module representing a symbol table can be written as a generative functor, taking no arguments, which creates a fresh abstract symbol type on every application, thereby preventing two different instantiations of symbol tables from interacting with each other's symbols.[33]

The two dialects have countless other differences ranging from surface-level to more extensive. As an example of the latter, OCaml defines an entire object system, as in object-oriented programming, but as with most studies of ML module systems this thesis is not at all concerned with that.

For reasons that will become clearer later on, this thesis is concerned with applicative semantics, not generative semantics.

TYPE SYSTEM FOR MODULES As a major departure from Haskell, the ML module system has a type system with a formal definition—or rather, the module system of the Standard ML dialect has a type system with a formal definition, and various dialects with various extensions have various formal definitions of their type systems.

Indeed, there have been decades worth of type systems formalized for ML modules in the research literature, beginning with the original *Definition of Standard ML*.[34] Some of the major

31 Generative functors were first introduced in OCaml 4.02 in 2014: `https://ocaml.org/releases/4.02.html`
32 For a system that straightforwardly supports both semantics as a distinction of side effects—the creation of abstract types—see the *F-ing* system of Rossberg *et al.* (2014).
33 Dreyer (2005), "Understanding and Evolving the ML Module System," Fig. 1.8.
34 Milner *et al.* (1990), "The Definition of Standard ML".

developments in type systems for ML modules thereafter were the introduction of "manifest types"/"translucent signatures"[35] (*i.e.*, type components of a signature that would *not* yield fresh abstract types when sealing a structure), applicative functor application[36] (explained shortly below), and recursive modules (explained in §1.2.4). Of particular note is the "F-ing" semantics of Rossberg *et al.* (2010) introduced a full 20 years after the original *Definition*.[37]

Having a type system means that ML modules exhibit a static semantics with a *modular* and *compositional* process for reasoning about them. For example, the type of a module expression is synthesized from the types of its constituent module subexpressions, without regard for the *manner* in which those subexpressions were typed, and all such types represent the specifications of module expressions without regard for the *values* to which those expressions evaluate.

FULLY FUNCTORIZED STYLE If every modular component just directly depended on every other component, without using functors to make those dependencies abstract, then components cannot be developed, checked, and reasoned about independently of each other; they must then be processed in (dependency) order. This is how a modular development in a *weak*-modular language like Haskell would necessarily be organized. In ML, however, there's another alternative: using the "fully functorized" style of modular development.

The central idea behind fully functorized modular development is that every direct module dependency is rewritten as a functor that must be applied to its dependency as an argument. For example, if module M_2 depends on another module M_1, then it's rewritten as a functor F_2 that is parameterized by (a signature for) M_1, with the dependency on the particular M_2 now expressed as the linking application $F_2(M_1)$. To carry this out "fully," one also performs the same rewriting on M_1, and thus the process continues to cover the entire chain of dependencies.

The result of this rewriting is that the components now exhibit strong modularity: implementations are abstracted from their dependencies via signatures. But there's a major cost. Writing the components as functors necessitates tedious, manual sharing constraints on functor parameters, in order to establish, in the type system, coherence among each component's dependencies.

For example, Figure 1.3 presents a basic modular development, adapting MacQueen's example program from (MacQueen, 1984, §2.3) to SML '90 syntax. At the bottom of the hierarchy, the functorized `Geometry` module needs to impose a sharing constraint asserting that the parameterized `RECT` and `CIRCLE` implementations contain the same `POINT` implementation—and therefore the same `point` abstract type. The typechecking of `Geometry` *requires* that coherence of the `point` type when comparing `R`'s `center` result with `C`'s. Moreover, the sharing constraint in the example allows the two intended (*i.e.*, coherent) linkages of all components and rejects the unintended (*i.e.*, incoherent) one.

MacQueen described a fully-functorized modular development (without using that phrase) in his early presentation of an ML module system[38] and, later, in a more formal development of type theory of modules.[39] The authors of the ML Kit, a Standard ML implementation itself implemented in Standard ML, employed the fully-functorized style in order to link together the key components of their system.[40] The verbosity and rigor of the sharing constraints in the ML Kit implementation is really something to behold! For example, the `Environments` functor is parameterized by 17 structures with 18 type sharing constraints. Similarly, the `STATIC_OBJECTS` signature declares only five constituent structures but 66 sharing constraints between them.[41]

35 Leroy (1994) and Harper and Lillibridge (1994), which were presented at the exact same POPL conference in 1994, surely a sign of the former research interest in the subject.
36 Leroy (1995), "Applicative Functors and Fully Transparent Higher-Order Modules".
37 Milner *et al.* (1990), "The Definition of Standard ML".
38 MacQueen (1984), "Modules for Standard ML".
39 MacQueen (1986), "Using Dependent Types to Express Modular Structure".
40 Birkedal *et al.* (1993), "The ML Kit, Version 1," §7.3.
41 The ML Kit source code and documentation currently can be found at `http://www.cs.cmu.edu/afs/cs/user/birkedal/pub/kit.tar.gz`

That meticulous effort was cited in the module systems literature for years as a negative example for the fully-functorized style, *e.g.*, in Leroy (1994). As Harper and Pierce summarized it in their chapter of *Advanced Topics in Types and Programming Languages* on modularity in ML:

> *Experience has shown this to be a bad idea: all this parameterization—most of it unnecessary—gives rise to spurious coherence issues, which must be dealt with by explicitly (and tediously) decorating module code with numerous sharing declarations, resulting in a net decrease in clarity and readability for most programs.*[42]

Another potential drawback to the fully-functorized style is that it requires higher-order functors if one wishes to rewrite dependencies not on structures but on functors.[43] For example, if M_2 depends on a functor F_1, then the functorization of M_2 would look like the following:

```
functor F2 (F1 : (functor (X : S11) : S12))
  = struct (* ... use F1 internally ... *) end
```

Indeed, since SML '90 lacks higher-order functors, the ML Kit implementation could not make use of such dependencies.

1.2.3 *Forms of Modularity: Compilation Units*

Outside the world of module systems on paper, there are module systems in real programming languages with real tools for checking, compiling, and executing modular programs. Each such language implementation includes some way to define modules in source files that can then be checked and compiled by the language's toolchain. These tools often live *above* the modular programming language, as they are mostly *informal* tools for processing modular programs. Indeed, this is the case for packages.

It is these *compilation units* that generally constitute *modules* for most languages and programmers, rather than a more formally described system like that of ML. When compilation units are *the* atomic structuring mechanism in a language, as in Haskell, they are considered *small* modularity. When they constitute a level above a more atomic structuring mechanism, in order to link large modular programs against unknown dependencies, as in ML's notions of units, they are considered *large* modularity.

For example, in Haskell, there's a one-to-one correspondence between module definitions and source files. The module *is* the compilation unit; the unit of development and reasoning *is* the unit of compilation. Since Haskell modules support only weak modularity, the modules must be compiled in dependency order. The GHC compiler, for instance, can then link together the resulting object files to produce executable code.[44]

SEPARATE COMPILATION If a language exhibits strong modularity then it has some facility for writing module implementations with respect to the interfaces of their dependencies. Since the original Backpack work[45] I've been careful to call this "separate modular development" rather than the more conventional term "separate compilation." That's because the extralinguistic compilation units determine whether the language supports separate compilation. Moreover, another reason to use the more general term "separate modular development" is that the act of *compiling* components in a modular system is losing its centrality as the way we interact with programs expands beyond merely compiling, linking, and executing them. Interactive theorem proving (*e.g.*, Coq[46]), type-driven development (*e.g.*, Idris[47]), and IDE tool-

Separate modular development, not separate compilation.

42 Harper and Pierce (2005), "Design Considerations for ML-Style Module Systems," pp. 335–336.
43 Appel and MacQueen (1994), "Separate Compilation for Standard ML".
44 This implementation of processing Haskell modules differs from the semantics given in the more formalized treatment of Diatchki *et al.* (2002). A key difference lies in the two approaches to recursive modules, which will be explained shortly.
45 Kilpatrick *et al.* (2014), "Backpack: Retrofitting Haskell with Interfaces," §1.
46 The Coq Development Team (2017), "The Coq Proof Assistant, version 8.7.0".
47 Brady (2013), "Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation".

```
signature POINT =
  sig
    type point
    val equal : point -> point -> bool
    ...
  end

(* a couple distinct POINT implementations *)
structure Point1 : POINT = (* some representation of points *)
structure Point2 : POINT = (* a different representation of points *)

signature RECT =                         signature CIRCLE =
  sig                                      sig
    structure P: POINT                       structure P: POINT
    type rect                                type circle
    val center : rect -> P.point             val center : circle -> P.point
  end                                      end

functor Rect(                            functor Circle(
          structure P: POINT                       structure P: POINT
        ) : RECT =                               ) : CIRCLE =
  struct                                   struct
    structure P = P                          structure P = P
    type rect = ... P.point ...              type circle = ... P.point ...
    fun center r = ...                       fun center c = ...
  end                                      end


functor Geometry(
          structure R: RECT
          structure C: CIRCLE
            sharing C.P = R.P  (* key sharing constraint! *)
        ) =
  struct
    structure P = C.P (* same as R.P *)
    structure R = R
    structure C = C

    (* requires that R.P.point = C.P.point *)
    fun sameCenter r c = P.equals (R.center r) (C.center c)
  end

(* well-typed *)
structure G1  = Geometry(Rect(Point1), Circle(Point1))
structure G2  = Geometry(Rect(Point2), Circle(Point2))

(* XXX ill-typed! Point1 =/= Point2 *)
structure G12 = Geometry(Rect(Point1), Circle(Point2))
```

Figure 1.3: Example of a fully functorized modular development, adapted from MacQueen (1984, §2.3). Note the sharing constraint in the functor parameters of `Geometry`.

ing (*e.g.*, type-checking expressions in the editor) all present new considerations for modular programming, and therefore all their processes should be folded into the idea of separately *developing* modules.

COMPILATION UNITS IN THE ML FAMILY The ML module system, as formally defined in the *Definition of Standard ML*, does not describe compilation units even though it contains functors, which are the canonical example of strong modularity and separate modular development. From the *Definition*:

> *In practice, ML implementations may provide a directive as a form of top-level declaration for including programs from files rather than directly from the terminal. [...] Rather than introducing a distinction between, say, batch programs and interactive programs, we shall tacitly regard all programs as interactive, and leave to implementers to clarify how the inclusion of files, if provided, affects the updating of the [top-level environment].*[48]

Addressing that conspicuous gap in Standard ML, Swasey *et al.* (2006) designed an extension to ML, called SMLSC, that supports *units* on top of the module system. SMLSC units offer strong modularity and are intended to better model compilation units than do functors. Additionally, they provide some form of mixins for convenient sharing of dependencies according to their names. As a language for describing both open and closed modular programs, sitting on top of the module system that provides facilities for data abstraction in the core language, SMLSC units are considered *large* modularity.

OCaml specifies compilation units on top of the module system as a way to "bridge the module system and the separate compilation system."[49] Each unit comprises an implementation and an interface and is interpreted as a module sealed with the corresponding signature. The OCaml compiler, `ocamlc`, allows users to write a unit's implementation as a file `Foo.ml` and its interface as `Foo.mli`, thereby creating a unit, `Foo`, that can be referred to as a module from within other units.[50]

At first glance this relationship between file system and module system might resemble Haskell's. However, OCaml's compilation units provide separate compilation while Haskell's do not. The key ingredient present in OCaml/`ocamlc` and missing in Haskell/GHC is the interface file, `Foo.mli`, which allows units that depend on `Foo` to be type-checked and even compiled before writing—let alone type-checking and compiling—the implementation of `Foo`.

So are OCaml's compilation units small or large modularity? Like SMLSC units, they are an abstraction designed on top of the unit of reasoning—modules, which provide the data abstraction for the core language—in order to facilitate the development of large modular programs; that fundamentally makes them a form of *large* modularity. Notably, however, their primary use case is to break a single author's modular program into units of compilation, so they don't exactly play the role of units of authorship and distribution.

INSTANTIATION What compilation units tend to lack is the ability to *instantiate* them multiple times to be *reused* in other contexts. For this reason, as described in §1.1.2, compilation units are a form of *large modularity*, not small modularity.

Again taking the Haskell program in Figure 1.1 as an example, there's no way to link the Client module (read: compilation unit) against some other implementation of integer sets precisely because it directly depends on the IntSetM module (compilation unit). Indeed, this limitation is the principal one that is overcome by Backpack.

The situation is the same in compilation units for ML, both with SMLSC's unit language and with OCaml's extralinguistic unit compilation system. That's because units provide a means to *link* indefinite or abstract dependencies against definite implementations, but they do not provide a means to *substitute* multiple such implementations for their abstract dependencies. (Recall the distinction between linking and substitution from §1.1.2.) The linking

48 Milner *et al.* (1997), "The Definition of Standard ML (Revised)," p. 64.
49 Leroy *et al.* (2017), "The OCaml System release 4.06: Documentation and user's manual," §7.12.
50 Ibid., §9.3.

form of instantiation is another telling sign that compilation units—for strong modular languages at least—are a form of *large* modularity.

1.2.4 *Forms of Modularity: Recursive Modules*

When multiple modules refer to each other in their definitions they are said to be *recursive modules*. The same goes for a module whose definition refers to itself.

In some settings, like in Haskell, recursive modules are quite natural to employ and to reason about. In other settings, particularly in ML, they are seemingly straightforward but extraordinarily complicated—both for the designer, who wishes to give them a precise formal definition and then to implement them, and for the user, who needs to reason about them in real programs.

Backpack continues Haskell's support for recursive modules and further, as one of its technical contributions, supports *recursive linking* of packages. Though they might seem straightforward at first glance, recursive modules are notoriously challenging to (precisely) define, implement, and even reason about as a user. Here I sketch some of their manifestations in Haskell and ML and give a sense for their complexity.

recursive modules in haskell Among compilation units, recursive modules arise naturally in the form of *cyclical imports*, *i.e.*, dependency cycles. All the modules in some strongly connected component, according to the import links, constitute a mutually recursive knot. Each module implementation in the knot must be checked, in dependency order, with some mechanism to break the cycle so that they may be processed linearly.

The Haskell language specification allows recursive modules but doesn't explain how any particular Haskell compiler should support them *qua* compilation units. The GHC compiler, for example, introduces the mechanism of "boot files" to break module import cycles. (As we shall see shortly, this mechanism plays a central role in the design of Backpack.) If modules A and B import each other, then one must create a boot file, `A.hs-sig`, which contains specifications of all the entities from A that are imported in B, and change B to import that instead. Then the import sequence, now finite, goes A → B → (`A.hs-sig`) → □, which GHC ties into a recursive knot.

GHC boot files resemble modules but contain only type-level information; there are no values to type-check in them. Actually they resemble OCaml's unit interface files.

Another semantics for Haskell modules, that of Diatchki *et al.* (2002), handles recursion by taking the least fixed-point of the function that processes a module's imported and exported entities.

recursive modules in ml In extensions to the ML module systems, recursive modules generally require a "forward reference," a module variable that acts as the recursive reference within the body, along with a signature for that forward reference that specifies all the recursively-referenced entities within the body. Crary *et al.* introduced this key syntactic form in their original work, "What is a Recursive Module?"[51] In OCaml, mutually recursive modules are necessarily defined together as one syntactic knot of definitions, each one designating a name and a signature acting as its forward reference. See Figure 1.4 for an example.

First, however, we consider a simple case of a single recursive module that refers only to itself. The following module, `RNat`, is a variant of a conventional natural number module but written as a recursive module in OCaml. The type and value entities of `RNat` use recursion *through the module system* rather than *through the core type system*: the recursive references within the type `t` and the function `to_int` both go through the forward *module* reference `RNat`. The resulting module has the same type and behavior as the more conventional definition without recursive modules (but with a recursive core data type and a recursive core function).

```
module rec RNat : sig
```

51 Crary *et al.* (1999), "What is a Recursive Module?"

```
signature TREE =
  sig
    type t
    type f
    val leaf : int -> t
    val fork : f -> t
    val push : t -> t -> t
    val has  : int -> t -> bool
  end

signature FOREST =
  sig
    type f
    type t
    val empty : f
    val add : t -> f -> f
    val contains : int -> f -> bool
  end

(* a recursively dependent signature *)
signature TREE_FOREST =
  rec (X) sig
    structure Tree   : TREE   where type f = X.Forest.f
    structure Forest : FOREST where type t = X.Tree.t
  end

(* a recursive module with two subcomponents *)
structure TreeForest =
  rec (X : TREE_FOREST) struct
    structure Tree =
      struct
        type t = Leaf of int | Fork of X.Forest.f
        type f = X.Forest.f
        val leaf = Leaf
        val fork = Fork
        val push x y = case y of
          | Leaf i -> Fork (X.Forest.add x (X.Forest.add y X.Forest.empty))
          | Fork f -> Fork (X.Forest.add x f)
        val has n = function
          | Leaf i -> i = n
          | Fork f -> X.Forest.contains n f
      end
    structure Forest =
      struct
        type f = list t
        type t = T.t
        val empty = List.nil
        val add = List.cons
        val contains n xs = List.exists (Tree.has n) xs
      end
  end
```

Figure 1.4: Example of mutually recursive modules in a hypothetical ML module system that resembles RMC (Dreyer, 2007a).

```
        type t
        val zero : t
        val succ : t -> t
        val to_int : t -> int
      end
    = struct
        type t = Zero | Succ of RNat.t
        let zero = Zero
        let succ = Succ
        let to_int = function
          | Zero -> 0
          | Succ m -> RNat.to_int m + 1
      end
```

THEY'RE HARD TO GET RIGHT The reasonable-looking definition of `RNat` above belies the deep technical challenges behind designing, formalizing, and implementing support for recursive modules in ML. Addressing this unanticipated challenge, Dreyer writes:

> *Certainly, for simple examples of recursive modules, it is difficult to convincingly argue why ML could not be extended in some ad hoc way to allow them. However, when one considers the semantics of a general recursive module mechanism, one runs into several interesting problems for which the "right" solutions are far from obvious.*[52]

Recursive modules were completely absent from the *Definition of Standard ML*, and even OCaml only added them in version `3.07`,[53] over a decade after its precursor Caml Light introduced the module system.[54] In a years-long attempt to "get it right," there have been various designs for and implementations of recursive modules in ML. Most notable is Dreyer's line of research, which sought to develop a type-theoretic formulation of recursive modules.[55]

A key technical problem that arises with recursive modules in ML is what Dreyer called the "double vision" problem,[56] whereby the type system does not equate abstract types created within the recursive module with their corresponding type components in the forward reference; as a result, the type system "sees double"—two types instead of one. In the `RNat` example, the problem manifests when the type system sees `t` and `RNat.t` as distinct types within the module definition.

Solving the double vision problem has generally required two key ingredients: "forward-declaring" abstract types in the forward reference for the module, and defining a "static pass" over a recursive module definition that, prior to the full typing pass over the definition, identifies any abstract types defined in the body and equates them with those in their forward declarations. Both are key ingredients in Backpack as well.

RECURSION AND SEPARATE MODULAR DEVELOPMENT The ML module system is the standard-bearer for strong modularity and separate modular development; one would hope not to lose those benefits when defining recursive modules. Mutually recursive modules, both in OCaml and in the research literature, must be fully defined together syntactically, thereby breaking a claim to support separate modular development when it comes to recursion.[57] The `TreeForest` recursive module from Figure 1.4 is an example; it must be defined syntactically together with its constituent recursive modules, `Tree` and `Forest`.

What one might then do is define each module in the recursive knot as a *functor* parameterized by the forward reference for the whole knot. See Figure 1.5 for a *purported* example of this technique in the RMC system[58] and Figure 1.6 for an analogous *purported* example in

52 Dreyer (2007b), "Recursive Type Generativity," p. 433.
53 Leroy (2003), "A proposal for recursive modules in Objective Caml".
54 See (Milner *et al.*, 1997, p. 95) for the historical note about Caml Light. Moreover, Dreyer's thesis (2005, §5.3) provides a full accounting of the various ML dialects' and implementations' support for recursive modules.
55 Dreyer (2005, 2007a,b).
56 Dreyer (2005), "Understanding and Evolving the ML Module System".
57 Ibid., §5.2.4.
58 Dreyer (2007a), "A Type System for Recursive Modules".

OCaml. The `MkTree` and `MkForest` functors are developed separately with no connection to each other, each with an interface standing in for the other. The intention of these purported examples is that these functors are then instantiated and *recursively* linked together within a recursive module, `SepTreeForest`.

These two `SepTreeForest` examples *seem to* exhibit recursive linking of separately-developed modular components, *i.e.*, functors, but are they well-typed in their respective systems? The answer is extraordinarily difficult to reason about for the programmer! In both cases the answer is *no*—`SepTreeForest` is ill-typed in both examples. Why are they ill-typed? Because in both cases the type system cannot equate the abstract type produced in its body, *e.g.*, `t` in `MkTree`, with the corresponding type appearing in its functor parameter, *e.g.*, `X.Tree.t`. Concretely, the occurrence of `y` in the `Leaf` branch of the `push` definition in `MkTree`'s body has the (locally produced) type `t`, but the type system sees the first parameter of `X.Forest.add` as having a different type, `X.Tree.t`. The type system knows that `X.Forest.t` is equal to `X.Tree.t`, due to the *recursively dependent signature*,[59] but that's not enough.

At first glance this seems like the double vision problem. OCaml doesn't solve that problem, but doesn't RMC? Wasn't that the whole point of it? Actually, it's not the double vision problem! The forward declaration of the abstract type is not actually a forward declaration at all; it's merely a type in a functor parameter, which may or may not have anything to do with the abstract type produced in the body, as far as the type system is concerned. The functor parameter is only treated *ex post facto* as a forward reference downstream in the body of the `SepTreeForest` recursive module, but by then the functor has already proven to be ill-typed.

As we'll see in the next subsection, on *mixins*, this example of separately developed recursive modules can indeed be expressed in the MixML system. In fact, it was designed in part for that purpose, and this example was drawn from that work.

Finally, one more observation about the `SepTreeForest` example in OCaml: it cannot be written such that `Tree` and `Forest` are OCaml *compilation units* instead of functors. In addition to the problem already identified, these compilation units' *interfaces* depend on each other, via their cross-defined type components, but OCaml *units* cannot express such recursively dependent signatures. OCaml modules, on the other hand, can express them; that's `TREE_FOREST`.

RECURSION AND DYNAMIC SEMANTICS OCaml's rejection of the last example demonstrates an additional concern about *dynamic semantics* for recursive modules. If modules `A` and `B` are defined as a mutually recursive knot, and if some of the `val` components in those modules introduce side effects when evaluated—say, printing to the screen, or mutating the heap—then the evaluation order of those two modules becomes quite important, as does the requirement that they're each evaluated only once. Because Haskell's purity means there are no such side effects in the core level when processing the module level, Backpack is not concerned with dynamic semantics; see Chapter 5 for more discussion of the goals of the Backpack formalization.

1.2.5 *Forms of Modularity: Mixins*

Backpack borrows heavily from the idea of *mixin modules*. Essentially, it uses them as a basis for packages, thereby bestowing the package level with a proper type system. Here I provide a high-level overview of mixins, focusing on the particular systems from which Backpack draws heavily.

In the literature, a mixin module typically refers to a namespace of entities (*e.g.*, types, values, or other modules), each of which has a name and a specification (*e.g.*, a kind, type, or interface) but which may or may not be defined. Linking two mixin modules causes their constituent entities to link together by name, with undefined entities of the same name merging together, and defined entities from one module instantiating undefined entities of the same

59 Crary *et al.* (1999), "What is a Recursive Module?"

```
(* TREE, FOREST, and TREE_FOREST same as before *)

functor MkTree (X : TREE_FOREST) :> TREE where type f = X.Forest.f =
  struct
    type t = Leaf of int | Fork of X.Forest.f
    type f = X.Forest.f
    val leaf = Leaf
    val fork = Fork
    fun push x y =
      case y of
        Leaf i -> Fork (X.Forest.add x (X.Forest.add y X.Forest.empty))
      | Fork f -> Fork (X.Forest.add x f)
    fun has n xs =
      case xs of
        Leaf i => i = n
      | Fork f => X.Forest.contains n f
  end

functor MkForest (X : TREE_FOREST) :> FOREST where type t = X.Tree.t =
  struct
    type f = list t
    type t = X.Tree.t
    val empty = List.nil
    val add = List.cons
    val contains n xs = List.exists (X.Tree.has n) xs
  end

(* ill-typed! *)
structure SepTreeForest =
  rec (X : TREE_FOREST) struct
    structure Tree   = MkTree(X)
    structure Forest = MkForest(X)
  end
```

Figure 1.5: Ill-typed rewrite, in RMC, of the implementations in Figure 1.4 to exhibit separate modular development. Each of the `Tree` and `Forest` implementations can be defined and type-checked separately from each other, as functors, but their recursive linking as `SepTreeForest` is ill-typed.

```
(* rewrite of the previous TREE_FOREST in OCaml syntax *)
module type TREE_FOREST =
  sig
    module rec Tree   : (TREE   with type f = Forest.f)
           and Forest : (FOREST with type t = Tree.t)
  end

(* ill-typed! *)
module rec SepTreeForest : TREE_FOREST =
  struct
    (* MkTree and MkForest as defined above *)
    module Tree   = MkTree(TreeForest)
    module Forest = MkForest(TreeForest)
  end
```

Figure 1.6: The recursively dependent signature and recursive module from the example in Figure 1.5, but with the recursive `SepTreeForest` module written in modern OCaml. As in the RMC example, this is an ill-typed module.

name in the other module, so long as their specifications in both modules are in some sense compatible.

Mixin modules originated primarily in an object-oriented form, as "abstract subclasses," proposed by Bracha and Cook (1990) for Modula-3 and showing up in modern object-oriented languages like Ruby and Scala. The conceptual presentation of mixins in this thesis more closely matches that of the Jigsaw system,[60] which developed mixin modules into a broader concept outside of object-oriented programming.

The true basis for mixins in this thesis, however, is MixML,[61] a core calculus of mixin modules that subsumes the entire ML module system.

In the following example based on MixML, two mixin modules, each containing a mixture of type and value entities, abstract declarations and concrete definitions, are linked together, via the `with` expression, into a larger mixin module `M`. The entities link together by name, creating type substitutions to apply on the result. For example, the r.h.s. mixin's `t` and `x` definitions will match up against the l.h.s. mixin by applying a substitution `t = int`; the l.h.s. specification of `x`, with substituted type `int`, therefore matches the corresponding r.h.s. definition.

```
         ⎧ t : type                 ⎫
         ⎪ x : t                    ⎪      ⎧ u : type             ⎫
         ⎪ f : t -> t               ⎪      ⎪ v : u                ⎪
let M =  ⎨ u = bool                 ⎬ with ⎪ g : u -> t           ⎪
         ⎪ v = true                 ⎪      ⎨ t = int              ⎬
         ⎪ g = fun b ->             ⎪      ⎪ x = 5                ⎪
         ⎩   if b then x else (f x) ⎭      ⎩ f = fun n -> n + (g u)⎭
```

UNITS FOR MIXINS Another major idea behind mixin modules is that of the *unit*,[62] *i.e.*, a suspended or unevaluated mixin module. (This should not be confused with the *compilation units* of §1.2.3.) As made clear by Owens and Flatt (2006) units act like functors in ML: they allow a mixin module to be instantiated and reused—re-linked—in different contexts. Owens and Flatt were also the first to suggest that units offer an alternative to the "fully-functorized" style of development in ML (§1.2.2).[63]

The mixin modules in the earlier example can be rewritten as units, allowing two different linked results. In the syntax of MixML, in which units are introduced with square brakets and instantiated with new, that would look like the following:

```
          ⎡⎧ t : type                 ⎫⎤
          ⎢⎪ x : t                    ⎪⎥
          ⎢⎪ f : t -> t               ⎪⎥
let U1 =  ⎢⎨ u = bool                 ⎬⎥
          ⎢⎪ v = true                 ⎪⎥           ⎡⎧ t = bool  ⎫⎤
          ⎢⎪ g = fun b ->             ⎪⎥           ⎢⎪ u = bool  ⎪⎥
          ⎣⎩   if b then x else (f x) ⎭⎦  let U3 = ⎢⎨ x = false ⎬⎥
          ⎡⎧ u : type             ⎫⎤              ⎣⎩ f = not   ⎭⎦
          ⎢⎪ v : u                ⎪⎥  let M' = (new U1) with (new U3)
          ⎢⎪ g : u -> t           ⎪⎥
let U2 =  ⎢⎨ t = int              ⎬⎥
          ⎢⎪ x = 5                ⎪⎥
          ⎣⎩ f = fun n -> n + (g u)⎭⎦
let M = (new U1) with (new U2)
```

60 Bracha and Lindstrom (1992), "Modularity Meets Inheritance".
61 Dreyer and Rossberg (2008) and Rossberg and Dreyer (2013).
62 Flatt and Felleisen (1998) and Owens and Flatt (2006).
63 Owens and Flatt (2006), "From Structures and Functors to Modules and Units," §9.

$$
\begin{array}{lcl}
\texttt{MkTree} & = & \left[\begin{array}{l}(\texttt{X} = \texttt{new TREE\_FOREST})\ \texttt{with} \\ \{\texttt{Tree} = \texttt{new TREE}(\texttt{X.Forest.f})\ \texttt{seals}\ \{\ldots\}\}\end{array}\right] \\
\\
\texttt{MkForest} & = & \left[\begin{array}{l}(\texttt{X} = \texttt{new TREE\_FOREST})\ \texttt{with} \\ \{\texttt{Forest} = \texttt{new FOREST}(\texttt{X.Tree.t})\ \texttt{seals}\ \{\ldots\}\}\end{array}\right] \\
\\
\texttt{SepTreeForest} & = & \texttt{new MkTree with new MkForest}
\end{array}
$$

Figure 1.7: Adaptation of the `TreeForest` example in the MixML system, from Rossberg and Dreyer (2013, Fig. 3). The elided parts of the unit definitions refer to the actual core-level definitions of `MkTree` and `MkForest`, just like in the original presentation in Figure 1.4. Unlike the RMC (Figure 1.5) and OCaml (Figure 1.6) examples, this one is indeed well-typed.

The `M` module is exactly the same as before. The additional `M'` module shows a *reuse* of the unit `U1` linked against a different module, the instantiation of `U3`.

RECURSIVE LINKING Mixin module linking does not impose a *syntactic* restriction on which of the two linked modules may contain undefined specifications and which may contain defined implementations. In the examples above, both sides of the mixin linking contain defined and undefined entities, which link together to form a completely defined module. This is known as *recursive linking*. Moreover, in the example with units, separately developed units were recursively linked together, thereby combining two strong features of modularity: separate modular development and recursive modules.

With ML functors, on the other hand, there's a strict syntactic abstraction barrier between linked modules. The body of a functor provides implementations that may refer to entities declared in the parameter of the functor. Functor application "links" the functor body with the functor argument. The original example couldn't be written as functor application since the argument cannot contain undefined parts to be satisfied by the functor.

MIXML AND STRONG MODULARITY WITH RECURSION MixML was designed in part to remedy the conspicuous gap in earlier proposals for recursive modules in ML: separate modular development. MixML accomplishes this with its own formulation of the two key ingredients for recursive modules: forward declaration of abstract types, which is folded into the key linking operation (`with`), and a "static pass" before typing in order to solve double vision.

Indeed, unlike RMC and OCaml, MixML can express the motivating example of separately developed recursive modules from the last subsection, `SepTreeForest`; see Figure 1.7.[64] The key point to make is that MixML's expression of `MkTree` and `MkForest` are now as *units* instead of functors, and in particular, the general recursive linking operation will equate the forward-declared type `X.Tree.t` and the locally defined type `Tree.t` within the body/r.h.s. of `MkTree`. That's because the `X` is no longer a disconnected functor parameter—that just so happens to be used downstream like a forward reference—but instead actually a recursive reference to the definition in the r.h.s.

With this example of strong modularity with recursion now properly expressed and well-typed, it's more clear why MixML was chosen as the technical basis for Backpack's type system. This choice satisfies Backpack's goal of strong modularity and the constraint that Haskell modules are often written recursively.

64 The reader should refer to the original MixML presentation of this example, in Rossberg and Dreyer (2013, Fig. 3), for the complete details.

1.2.6 *Forms of Modularity: Packages*

Package management systems, as described in §1.1, are the canonical example of *weak*, *large* modularity: package implementations depend on other implementations—a key point made here—and they constitute a unit of authorship, versioning, and distribution in large repositories of open-source modular programming.

In this section, I lay out what challenges packages pose for modularity and how Backpack addresses them. I spell out packages in more detail here than I did with other forms of modularity above, for two reasons. First, there are, to my knowledge, no standard formal treatments of packages in the research literature on types or modularity, making their expanded inclusion in this survey of modularity all the more necessary. Second, being the level at which Backpack is defined in Haskell, packages have a certain centrality in the broader mission of Backpack, and the practical problems with them that Backpack addresses require some careful motivation.

PACKAGES IN HASKELL Packages come in many different *forms*, but they generally do not come in *formalism*. Instead, there are various bespoke systems, for general open-source programming and sometimes for particular programming languages, that implement their own custom tools and file formats.

Because of its singular relevance to Backpack, the most salient package management system is that of Haskell, the Hackage/Cabal framework,[65] consisting of Hackage, the repository, and Cabal, the specification and suite of tools. The intentionally limited presentation of packages in this thesis is therefore tailored to Hackage/Cabal, but it's nonetheless a presentation that includes enough unifying concepts to cover a myriad of systems.

Like most package management systems, it concerns itself primarily with tools for organizing, distributing, and building packages *as Haskell code*. Also like virtually every package management system, Cabal packages are not defined as an expression language with a type system and with module expressions as constituent phrases. In lieu of a type system, the accompanying tools check some properties (like uniqueness of package identifiers in the repository) and leave others for users to govern via socially enforced policy (like the Haskell Package Versioning Policy[66], or PVP).

Hackage[67] constitutes the central repository of Haskell packages. Cabal,[68] short for "Common Architecture for Building Applications and Libraries,"[69] constitutes the package specification and build tools, as well as the dependency resolution tool, called `cabal-install`. Moreover, Haskell implementations (like GHC) provide the tools for installation and local installation caches.

PACKAGE SYSTEMS AS FORMAL LANGUAGES In this thesis, rather than an informal suite of tools, packages constitute a language level above that of modules, which itself sits above the level of the core language of types and terms. One of the key insights in this thesis is that, like the module and core levels beneath it, that package level should be a formally defined language *with a type system*. Given the "non-reformist reform" orientation of this thesis, consider it part of my ideological goal to bring the research program of formal type systems for modularity, as a language, to the largely informal domain of package management.

Backpack does exactly that: it defines an expression language for packages, with a type system, on top of Haskell's module system. Backpack turns the *weak* modularity of package management systems into *strong* modularity by "retrofitting [the module level] with interfaces."[70] In so doing, Backpack addresses two fundamental limitations of package systems:

65 Coutts *et al.* (2008), "Haskell: Batteries Included".
66 `https://pvp.haskell.org/`
67 `http://hackage.haskell.org/`
68 `https://www.haskell.org/cabal/`
69 Ibid.
70 To be more precise, Backpack's packages are still *weak* in the sense that packages depend directly on other packages, which, as mixins, have no tangible interface-vs.-implementation trade-off. Practically speaking, however, they still allow implementations to depend on interfaces, albeit at the level of modules.

the inability to define implementations abstractly, in a typed manner, in terms of their dependencies (a problem with authorship) and the ubiquity of "dependency hell" configurations of packages (a problem with versioning). It does not, however, formally describe either authorship or versioning; these crucial roles of package systems are left to the informal world of implementation—and, hopefully, future work on type systems for packages.

A CONVENTIONAL DEFINITION OF PACKAGES What are the key concepts for package management systems? A *package* P defines a modular component that depends on other modular components in some central *repository* (*e.g.*, Hackage). Each package has four key constituent parts.

- First, each package has a *package name* p, like `twitterlib`, that designates not merely a single package but a *package lineage* within the repository. That lineage includes all known versions of the package.

- Second, each package has a *version* ν, like `1.0.2` or just `5`, that is unique within the package's lineage. Together, a name and a version constitute a *package identifier* which uniquely identifies a package (within some repository) as p-ν, like `twitterlib-1.0.2`. It's important to note that the package name alone does not identify a package; it instead identifies a package lineage.

- Third, each package contains a bundle of source files that constitute the modular component it provides. The modular component exists at the level of some underlying programming language (or languages). Typically the component defines a library for others to use in their own programs, or it could define an executable "main" program, or both. For example, `twitterlib-1.0.2` would provide a `Twitter` module, in some underlying language, which provides functionality for interacting with Twitter's services.

- Fourth, each package has a set of *dependencies* on other packages. A dependency specifies a package name q, thereby designating a package lineage, and a *version range* V. The interpretation is that the defining package depends on any version ν of q such that $\nu \in V$. The broader the range V, the more versions of q that can satisfy the dependency. Our `twitterlib-1.0.2` example might have a dependency `httplib >= 2.0`: the `httplib` package lineage constrained to any version number at or after `2.0`.

  To be precise, it's not quite correct to say that "package P depends on package Q" since the dependency is on some *version range* V *within the lineage of* q rather than on a *particular package* q-ν.

PACKAGE DEPENDENCIES AS INTERFACES Why does package P depend on package (lineage) q? Generally, it means there's a dependency in the underlying program entities of the two packages: P provides a module M_{p} that imports another module M_{q} that is provided by package (lineage) q. Because of the underlying program dependency, one cannot build (a concrete instantiation of) P without first building some (concrete instantiation of) package Q whose version satisfies the dependency. The `Twitter` module inside `twitterlib-1.0.2` needs to import the `Http` module provided by `httplib >= 2.0`, for example.

And why does package P depend on a particular *version range* of (lineage) q? Because in the absence of a type system at the level of packages, the version range approximates an *interface* for the depended-upon package. The `httplib >= 2.0` dependency perhaps designates a particular function (signature) in the provided `Http` module, a function that was not present before version `2.0`, and a function that is presumed to continue existing (with the same signature) in later versions; as an example of such a function, consider `getNumLikes` which has type `TweetID -> Int`.

An upper bound, like `httplib < 5.0`, designates a "known unknown": the interface of the dependency is known at least up to the `4.*` versions of `httplib`, but after that it might change in a way that's incompatible. For example, version `5.0` might remove `getNumLikes`, or it might change its signature to `Auth -> TweetID -> Int`; then again, it might leave the function alone.

As one can imagine, this approximation of interfaces by version ranges of package lineages is prone to all sorts of errors. Instead of mechanical checks about interfaces, it relies on social enforcement of the aforementioned Package Versioning Policy, which essentially acts to tie interfaces to version numbering.

INSTANTIATION A package identifier determines the four pieces above—in particular, the source files for a modular component in an underlying language. But it does not yet determine a *complete* component that can be checked, reasoned about, or built in that underlying language. That's because the dependencies specify subsets of package lineages rather than particular versions of those lineages.[71]

A concrete *instantiation* or *instance* of a package is the pairing of the package itself with the instantiations of its dependencies. As a complete component, a package instance can finally be checked, reasoned about, and built. For example, although `twitterlib-1.0.2` contains a bundle of source files, namely the file defining the `Twitter` module, the `Http` module gotten from `httplib >= 2.0` has not been chosen yet. Once instantiated with a particular instance of, say, `httplib-2.5`, then `twitterlib-1.0.2` can finally be built against that instance.

Because of their incompleteness and their ability to be instantiated, we'll see shortly that a package is, in the parlance of ML, more like a functor than a structure—with a key caveat.

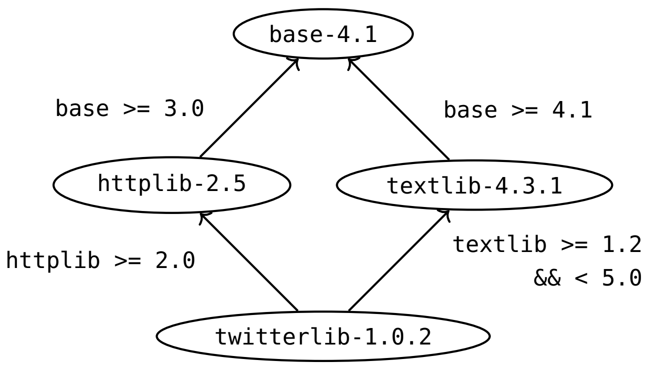
BUILDING AND DEPENDENCY RESOLUTION A package management system also principally includes two tools: a *build tool* for building a package's provided source files (read: modular components) with respect to particular instantiations of its dependencies, and a *dependency resolution tool* to locate appropriate packages that satisfy given dependencies, *i.e.*, version range constraints.

The build tool encapsulates one of the two main limitations of packages that Backpack addresses: a package's source files can only be compiled once *particular instantiations* of its dependencies have been chosen. In other words, as the author of a package, there's no way to build the source files of its provided components *abstractly* against arbitrary dependencies. That's because there's no notion of interface for those dependencies, apart from the name and version range which collectively don't describe any actual code to consider.

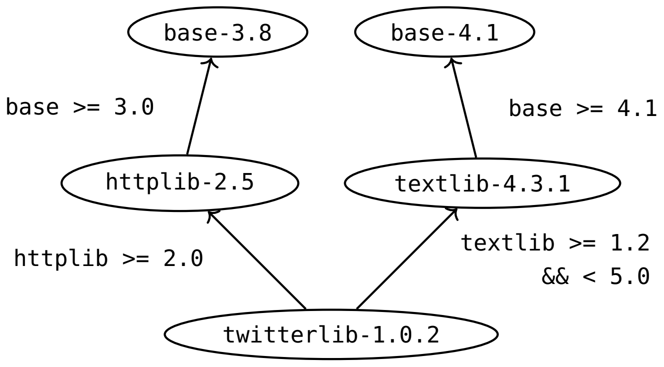
Dependency resolution is intricately related to the idea of an *installation cache* of packages: a local repository of "installed" packages at particular instantiations. When a user wants to build and install a package, like `twitterlib-1.0.2`, the dependency resolution tool must pick particular instantiations of all transitive dependencies.

DIAMOND DEPENDENCIES In order to minimize the number of different instantiations of packages, the tool will attempt to *reuse* packages from the installation cache. For example, if a particular instantiation of `httplib-2.5` lives in the cache, then the dependency resolution tool will try to satisfy constraints like `httplib >= 2.0` with that particular instance of `httplib-2.5`, which in turn determines the particular instance of `base` that `httplib-2.5` was built against, `base-4.1`. Then when it needs to satisfy `twitterlib`'s other dependency constraint, `textlib >= 1.2 && < 5.0`, it will attempt to reuse that same instance of `base` for `textlib`'s transitive dependency on it. This setup results in the "diamond dependency" below, in which the two "sides" depend on *the same* instance at the top.

71 Even in the case that P depends on a single version of q—*i.e.*, because the version range is a singleton, $\{v\}$—that $q - v$ might itself have open dependencies on other packages.



But this process can easily go wrong. What happens if the installation cache contained `httplib-2.5` built against `baselib-3.8` and then dependency resolution needed to find an instance of `textlib` for `twitterlib`'s dependency? Since `textlib` depends on `baselib >= 4.1`, the version in the cache won't work. Then dependency resolution will choose some other instance of `baselib` to satisfy `textlib`'s dependency, say, `baselib-4.1`.



The result is what is commonly called "dependency hell." The diamond is broken, and now the left and right packages see *different* versions of the top while the bottom sees *inconsistent* versions of it.

DEPENDENCY HELL IN THE TYPE SYSTEM Though the dependency hell situation certainly looks bad, it's not clear what's fundamentally wrong with it. One pragmatic answer is that there are now two sets of build artifacts for the two instances of `baselib` in the installation cache, when ideally there would have just been one. But there's a deeper problem with dependency hell, one that concerns the type system of the underlying core language in the modular components provided by the packages: the bottom package sees two inconsistent versions of the data types defined in (the modular components provided by) the top package.

Let's presume the dependency hell example were defined in Haskell's package system. Now consider the `String` data type defined in the `base` package. It's quite likely that within `twitterlib`, there's some function that tries to pass a `String` value gotten from the left package, `httplib`, into a `String` function gotten from the right package, `textlib`.

But that function in `twitterlib` would be ill-typed in Haskell. That's because the *original name* of `String` (recall §1.2.1) in the left package would differ from that in the right package, due to their originating in two different modules from two different instances of `base`.

Dependency hell therefore poses a problem in the type system *below* the package level that cannot be detected *at* the package level. As a result, such package configurations must be *conservatively ruled out*, even if there was no such intermingling of data types to cause the underlying type error. After all, if there was no such intermingling of `String` between the left and right packages, then there'd be no *type* problem in allowing the two instances of the top package.

Indeed, there are common configurations of packages for which this broken diamond should by no means be considered "hell." Consider the case that the top package is some package providing unit testing functionality, like `QuickCheck`, that the left and right packages use *internally*, without exposing it in their own interfaces. Left and right could depend on totally incompatible version ranges—`QuickCheck-2.*` introduced API-breaking changes—without any problems for left, right, or bottom packages.

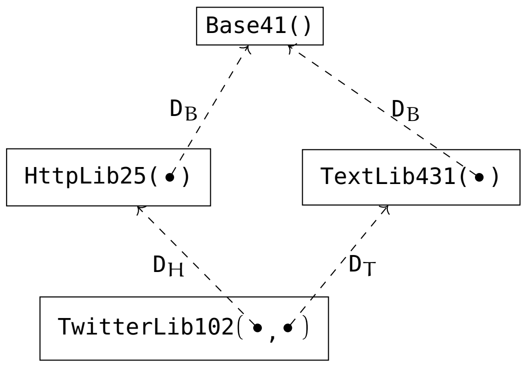
To address the dependency hell problem, Backpack introduces a type system at the package level. Since the types of packages are composed of the types of modules, the package level is aware of exactly those configurations of dependencies which would lead to type errors. Put simply, dependency hell ceases to be an *inherent* problem for the type system.

A STRONGER VIEW OF PACKAGES Packages are a form of *weak* modularity since implementations depend directly on implementations. To get a better comparative frame to understand what *strong* modularity for packages might look like, let's consider a hypothetic model of packages in terms of the gold standard of modularity—ML modules. The result is a strong-modular language of packages that looks more like the various examples of ML modularity I've presented so far.

Because a package is an open modular component whose dependencies do not identify direct instantiations, *i.e.*, implementations, a package acts like a parameterized module—a functor—that abstracts over particular instantations of its dependencies. A package instance is therefore like a functor application: the pairing of the provided component with the concrete implementations it depends on.

With this analogy established, we can view a package as a fully functorized module (§1.2.2). Each package depends on other packages through an abstract reference. The installation tool's policy to keep consistent instantiations of upstream packages resembles the use of sharing constraints among functor parameters: in both cases it's a way to ensure that diamond dependencies "share" the same implementation of the top, upstream corner of the diamond. That sharing is the way to avoid "dependency hell."

```
module TwitterLib102 =
  functor (D_H : HTTPLIB_2, D_T : TEXTLIB_4) with D_H.D_B = D_T.D_B ->
    struct ... end
```



But there's a key problem with this example that demonstrates the weakness of the "packages as functors" model. The `TwitterLib102` functor's parameters abstract over signatures `HTTPLIB_2` and `TEXTLIB_4`, but in the world of packages, there are no such signatures. Instead of typed interfaces, package dependency abstracts over version ranges of package lineages. This deficiency means that a package cannot be checked, reasoned about, or built, abstractly, without first picking an instantiation of each of its dependencies to process it with. An important consequence of this deficiency is that packages must be processed in order, only after all implementations, *i.e.*, all instantiations of dependencies, have been fully resolved. Packages therefore cannot be said to provide "client-side abstraction" as in ML; they're inherently restricted to weak modularity.

1.3 CONTRIBUTIONS

So far I've taken a stab at describing modularity, partly through my own definitions and partly through a survey of all the related forms and features of modularity that concern this thesis. My hope was to plant the reader in a broader research tradition—multiple traditions, in fact—so that she might better understand the novelty of Backpack. Here I summarize more definitively what that novelty is, *i.e.*, what this thesis contributes, first in hazier terms of modularity and design and then in more precise terms of formal, technical content.

1.3.1 *Modularity and Design Contributions*

STRONG, LARGE MODULARITY Backpack is an extension to Haskell that supports *strong modularity* and separate modular development on top of Haskell's weak modularity and incremental modular development. Like in the ML module system, Backpack relies on the notion of *interfaces* to establish that strong modularity, thereby allowing implementations to abstract over their dependencies.

As a language and type system of modular packages, Backpack exhibits *large modularity* similar to—but more expressively than—other systems of compilation units. That means that modularity in Backpack is concerned with the static organization of modules according to their dependencies on abstract, coarse-grained components, rather than the dynamic execution of code—or even the creation and enforcement of data abstraction in the core level's type system.

CONTRIBUTION TO PACKAGES Packages in Backpack are expressed with a *mixin*-based language and a *type system* that characterizes their well-formedness with respect to the underlying type system of modules and core entities. Backpack brings the principled and formal approach of type systems to the mostly unprincipled and informal world of package management systems.

This approach addresses two practical problems with Haskell packages today. First, a package author can develop her package implementation with respect to *abstract dependencies* on other packages, *i.e.*, with respect to typed interfaces for those dependencies. Second, "dependency hell" can be mitigated through ordinary appeals to the package type system. In Backpack, dependency hell is ruled out by the package type system exactly when it's ruled out by the underlying language's type system.

CONTRIBUTION TO MIXINS Backpack brings a fresh perspective on *mixins*, an area of programming languages that has seen some familiar usage in object-oriented languages but not much attention in the types and modules research literature. A key idea in this thesis is that *mixins offer a perfect model for packages*. To my knowledge this is the first research effort to formally model packages, as they're conventionally defined, as mixins.

By twisting existing ideas and notation, mixins allow packages with deeply-nested dependencies to seamlessly link and share those dependencies without the syntactic overhead of sharing constraints that the "fully functorized" idiom of ML functors would require in order to model packages. Moreover, the recursive linking of mixins could enable entirely new forms of modular package development that aren't expressible in conventional package systems.

GENERICITY OF THE APPROACH The design of Backpack needn't be fixed to Haskell; the same approach could be applied to any language with only weak, small modularity, of which there are many. Essentially, Backpack "hijacks" the module import mechanism of the underlying language in order to allow abstract dependency. It also hijacks the representation of core entities by embodying "original names" with structured *module identities* rather than static module names.

PACKAGE, NOT MODULE LEVEL Rather than supplanting the existing module system of Haskell, Backpack is designed at the level of packages, for three key reasons.

- First, unlike Haskell's module level, which is part of the Haskell language definition[72] and has seen further formalization,[73] the package level is largely comprised of (well-designed) tools without formal specification. Put simply, it's no where near as set in stone as Haskell's module level, so it makes for an easier target for the non-reformist reform project of Backpack.[74] Not only that, by leaving the module system largely as-is, Backpack can reuse the existing tools for the module level, *e.g.*, the mechanism for type-checking a module implementation with respect to a binary representation of another module's interface.

- Second, the large modularity of packages, in Haskell and elsewhere, has remained largely unexplored by the types research community. It's therefore an interesting direction to take in its own right. Backpack expands the principled reach of type systems into the quite unprincipled—and increasingly critical for modern software development—domain of distributed, open-source packages.

  Moreover, with that expanded reach of types, a couple practical problems with packages in Haskell today can be addressed in a more principled manner: a package author can develop her package implementation with respect to *abstract dependencies* on other packages, *i.e.*, with respect to interfaces for those dependencies, and "dependency hell" can be mitigated through ordinary appeals to the package type system.

- Third, although some might wish to define and use ML functors within the Haskell module system, it's unclear how they would integrate with the other language features. Type classes in Haskell can be used to parameterize code over bundles of types and accompanying operations, as a functor does, although not nearly as expansively: only individual values can be so parameterized, not whole modules. And similarly the data abstraction offered by ML functors and opaque sealing can be approximated within Haskell by carefully exposing data types without their term constructors and by employing the `newtype` feature.

1.3.2 *Technical Contributions*

This thesis is first and foremost a design—or even ideological—contribution. As a familiar saying in the field goes, with Backpack I've staked out a new corner of the design space for modular programming. With that caveat aside, this thesis also presents some technical contributions:

- The principal technical concept of *module identity*. Some prior type systems for modules included a notion of module identity, usually called structure stamps.[75] But this thesis is the first to expand the notion to the theory of *recursive first-order terms* in order to model both *applicative instantiation* and *recursive module identity*. The modeling of core-level abstract types and all all other core entities *indirectly* via module identities—as recursive first-order terms—is also a contribution.

- The formalization of Haskell modules on top of an abstract, axiomatized formalization of the Haskell core language. It's the first such formalization to offer a conventional typing judgment complete with a deterministic algorithmization and accompanying metatheory about the judgment. Moreover, it's the first type system for the Haskell module system that supports recursive modules.

72 Marlow (2010) and Peyton Jones (2003).
73 Diatchki *et al.* (2002) and Faxén (2002).
74 Even at the package level, implementing Backpack requires considerable changes not just to the Cabal/Hackage package management tools but also to the GHC compiler itself. See the discussion of Yang's Backpack'17 implementation in §10.8.
75 More discussion on structure stamps and sharing in prior type systems can be found in §10.4.

- The formalization of the (new) package level of Backpack. Though rooted in MixML, this formalization adopts multiple simplifications for the setting of Backpack, resulting in a smaller mixin module system that could be applied elsewhere. In particular it relies on the recursion inherent to its *module identities* to formalize the concept of recursive mixin linking.

- Metatheoretic evidence of the antimodular nature of Haskell type classes. My formalization of Backpack involved the development of metatheory about type classes' interaction with the module system. In some places that metatheory required complicated side conditions on otherwise conventional theorems about the type system, like *Weakening*. One contribution of this thesis is new, *technical* evidence for the folklore claim that type classes are antimodular.

- An elaboration semantics from Backpack packages to plain Haskell modules. This elaboration offers a formal "implementation" of Backpack that illustrates the essence of the idea: the package level is an abstraction that is "compiled away" into (a straightforward model of) plain Haskell modules.

  A main soundness theorem formally specifies that the source and target of elaboration have related modules, *i.e.*, that the Haskell core level sees the exact same modular components before and after elaboration. This theorem, complete with its hefty proof, acts as technical validation of the Backpack design.

Part I

BACKPACK: RETROFITTING HASKELL WITH INTERFACES

2 DESIGN OF BACKPACK

Now that modularity in the ML tradition and in Haskell have been mapped out, it's time to dive into Backpack. The language supports *strong, large modularity*, as described in the introduction but in the context of other languages. What does that actually look like *in Backpack*?

In this chapter I attempt to answer that question with a high-level overview of Backpack and its features, focusing on its package level and its interaction with the module level. Following this one, Chapters 3 and 4 round out Part 1 of this thesis. My aim with this organization is that most readers can absorb the main *design* ideas of Backpack by reading Part I, while a subset of those readers can peruse Part II for the *formalization* of Backpack.[1]

* * *

Figure 2.1 gives the syntax of Backpack. A *package definition* D gives a package name P to a a sequence of *bindings* $\overline{B}$. The simplest form of binding is a *concrete module binding*, of the form $p = [M]$, which binds the module name p to the implementation [M]. For example:

package ab-1 **where**
  A = [x = True]
  B = [import A; y = not x]

The code in square brackets represents module *implementations*, whose syntax is just that of a Haskell module (details in Chapter 7). Indeed, in a practical implementation of Backpack, the term [M] might be realized as the name of a file containing the module's code. However, note that the module lacks a header "module M where ..." because the module's name is given by the Backpack description.[2]

Package ab-1 binds two modules named A and B. The first module, bound to A, imports nothing and defines a core value x, and the second module, bound to B, imports the first module and makes use of that x in its definition of y. The type of this package expresses that it contains a module A which defines `x :: Bool` and a module B which defines `y :: Bool`. (We will more precisely discuss types, at the package and module levels, in Chapter 3.)

1 This chapter is an adaptation of §2 in the original Backpack paper. The substantial addition of *type classes* to Backpack, an addition since the original presentation of Backpack that this chapter is based on, will be introduced in Chapter 4.
2 We still provide syntax for optional "export lists" of core language entities; only the module name disappears.

Package Names	$P \in \mathit{PkgNames}$
Module Path Names	$p \in \mathit{ModPaths}$
Package Respositories	$R ::= \overline{D}$
Package Definitions	$D ::= \textbf{package}\ P\ t\ \textbf{where}\ \overline{B}$
Bindings	$B ::= p = [M] \mid p :: [S] \mid p = p \mid \textbf{include}\ P\ t\ r$
Thinning Specs	$t ::= (\overline{p})$
Renaming Specs	$r ::= \langle \overline{p \mapsto p} \rangle$
Module Expressions	$M ::= \ldots$
Signature Expressions	$S ::= \ldots$

Figure 2.1: Backpack syntax.

The module bindings in a package are explicitly sequenced: each module can refer only to the modules bound earlier in the sequence. In fact the bindings should be interpreted as iteratively building up a local module context that tracks the name and type of each module encountered. For example, if the order of the two bindings were reversed, then this package would cease to be well-typed, as the module reference A would no longer make sense.

Module bindings do not shadow. Rather, if the same module name is bound twice, the two bindings are *linked*; see Section 2.3.

2.1 TOP LEVEL AND DEPENDENCIES

A *package repository* consists of an ordered list of package definitions. Each package in a repository sees only those packages whose definitions occur earlier in the sequence. To make use of those earlier packages — *i.e.*, to depend on them — a package *includes* them using the **include** binding form, thus:

```
package abcd-1 where
  C = [x = False]
  include ab-1
      ⎡  import qualified A ⎤
  D = ⎢ import qualified C  ⎥
      ⎣ z = A.x && C.x      ⎦
```

One should think of an **include** construct as picking up a package and dumping all of its contents into the current namespace. In this case, the modules A and B are inserted into the package abcd-1 as if they were bound between C and D. Consequently the module bound to D can import both A and C. The type of abcd-1 says that it provides four modules: C (which provides `x :: Bool`), D (which provides `z :: Bool`), and the two modules A and B from package ab-1, even though they were defined there and merely included here. (The modules exposed by a package can be controlled with syntax that resembles that of the module level; this feature is discussed as a special case of *thinning* in §2.4.)

Throughout this thesis I treat the example package definitions as the bindings in a single package repository. At this point, that top level includes the definition for ab-1 followed by abcd-1.

2.2 ABSTRACTION VIA INTERFACES

Up to this point, the package system appears only to support weak modularity since each module can only be checked after those that it depends on. For example, abcd-1 could only be developed and checked *after* the package ab-1 had already been developed and checked; otherwise we would not be able to make sense of the import declaration import qualified A and the subsequent usage of `A.x` as a `Bool`.

To support strong modularity as well, Backpack packages may additionally contain *abstract module bindings*, or *holes*. To specify a hole, a developer provides a set of core-language declarations, called a *signature* S, and binds a module name p to it by writing p :: [S]. One should think of holes as obligations to eventually provide implementing modules; a package is not *complete* until all such obligations are met. Concrete modules, on the other hand, are simply those bound to actual implementations (as in all previous examples). This combination of abstract and concrete components reflects the mixin-module basis of our package system.

As our first example, we simulate how the abcd-1 package might have been developed modularly by specifying holes for the "other" components, A and B:

```
package abcd-holes-1 where
  A :: [x :: Bool]
  B :: [y :: Bool]
  C = [... as before ...]
  D = [... as before ...]
```

By "stubbing out" the other components, the developer of abcd-holes-1 can typecheck her code (in C and D) entirely separately from the developer who provides A and B. In contrast, in the existing Cabal package system, developers cannot typecheck their package code without first choosing particular version instances of their dependencies. Effectively, they test the well-typedness of their code with respect to individual configurations of dependencies which may or may not be the ones their users have installed.

Manually writing the holes for depended-upon components, as above, involves too much duplication. Instead a developer can define a package full of holes that designates the interface of an entire component. A client developer includes that package of holes and thus brings them into her own package without writing all those signatures by hand. The following two packages achieve the same net result (and have the same type) as abcd-holes-1, but without signatures in the client package:

```
package ab-sigs where      package abcd-holes-2 where
  A :: [x :: Bool]           include ab-sigs
  B :: [y :: Bool]           C = [... as before ...]
                             D = [... as before ...]
```

Holes are included in exactly the same manner as concrete modules, and they retain their status as holes after inclusion. Under the interpretation of holes as obligations, inclusion propagates the obligations into the including package.

In these two examples we have named the packages abcd-holes-1 and abcd-holes-2, which might suggest multiple versions of a single package abcd-holes (*e.g.*, in Cabal). However, while they may convey that informal intuition, in the present work we focus on modularity of packages, leaving a semantic account of versioning for future work.

2.3 LINKING AND SIGNATURE MATCHING

So far, all package examples have contained bindings with distinct names. What has appeared to be mere sequencing of bindings is actually a special case of a more general by-name linking mechanism: linking two mixin modules with strictly distinct names merely concatenates them. Whenever two bindings share the same name, however, the modules to which they are bound must themselves link together. This gives rise to **three cases**: hole-hole, mod-mod, and mod-hole.

First, when linking two holes together, we merge their two interfaces into one. This effectively joins together all the core language declarations from their respective signatures. The resulting hole provides exactly the entities that both original holes provided.

```
package yourlib where
  Prelude :: [data List a = ...]
  Yours    = [import Prelude; ...]
package mylib where
  Prelude :: [data Bool = ...]
  include yourlib
  Mine     = [import Prelude; import Yours; ...]
```

The mylib package above declares its own hole for Prelude and also includes the hole for Prelude from yourlib. Before the binding for Mine is checked, the previous bindings of Prelude must have linked together. This module can see both `List` and `Bool` since they are both in the interface of the linked hole, whereas the Yours module could only see the `List` datatype in Prelude. (Swapping the order of the first two bindings of mylib has no effect here.)

This example highlights another aspect of programming with mixin-based packages: each package has the option of writing *precise* interfaces for the other packages (*i.e.*, modules) it depends on. Specifically, yourlib only needs the `List` datatype from the standard library's Prelude module, rather than the entire module's myriad other entities. This results in a stronger type for yourlib since the assumptions it makes about the Prelude module are more precise and focused.

Not all interface merges are valid. For example, if mylib had also declared a `List` datatype, but of a different kind from that in yourlib (*e.g.*, `data List a b = ...`), then the merge would be invalid and the package would be ill-typed. The *merging judgment* that rules out such definitions will be presented in §3.2.

Second, when linking two module implementations together, it intuitively makes no sense to link together two *different* implementations since they define different code and different types. Backpack therefore requires that mod-mod linking only succeed if the two implementations are *the same*, in which case the linkage is a no-op.To test this, we require equivalence of their *module identities* (about which see Sections 2.4 and 3.1).

Consider the following classic diamond dependency:

```
package top where          package right where
  Top  = [...]               include top
                             Right  = [...]
package left where         package bottom where
  include top                include left; include right
  Left  = [...]              Bottom  = [...]
```

The bottom package of the diamond links together the packages left and right, each of which provides a module named Top that it got from the top package. The linking resulting from the inclusions in bottom is well-typed because left and right provide the *same* module Top from package top.

Third, when linking a module with a hole, the module's type must be a subtype of the hole's, and we say that the module "fills," "matches," or "implements" that hole. This form of linking most closely resembles the traditional concept of linking, or of functor application; it also corresponds to how structures match signatures in ML. Roughly, a module implements a hole if it defines all the entities declared in that hole and with the exact same specifications.

The mylib package above has a hole for the Prelude module. As this package is not yet complete, it can be typechecked, but not yet compiled and executed. (Supporting separate *compilation* would require sweeping changes to GHC's existing infrastructure.) We therefore link mylib with a particular implementation of its Prelude hole so that it may now be compiled and used:

```
package mylib-complete-1 where
  include mylib
             ┌   data List a = ... ┐
  Prelude  = │ data Bool = ...     │
             │ null xs = ...       │
             └                     ┘
```

The implementation of Prelude provides the two entities declared in the hole (included from mylib) and an additional third entity, the value `null`. This implementation matches the interface of the hole, so the linkage is well-typed.

For simplicity, our definition of when a module matches a hole is based on *width* rather than *depth* subtyping. In other words, a module may provide more entities than specified by the hole it is filling, but the types of any values it provides must be the same as the types declared for those values in the hole's signature. In particular, the match will be invalid if the implemented types are more general than the declared types. For example, a polymorphic identity function of type `forall a :: *. a -> a` will not match a hole that declares it as having type `Int -> Int`.

2.4 INSTANTIATION AND REUSE

Developers can reuse a package's concrete modules in different ways by including the package multiple times and linking it with distinct implementations for its holes; we call each such linkage an *instantiation* of the package. Furthermore, in Backpack, packages can be instantiated multiple times, and those distinct instantiations can even coexist in the same linked result. (In contrast, both Cabal and GHC currently prevent users from ever having two instantiations of a single package in the same program.)

Figure 2.2 provides an example of multiple instantiations in the multinst package, but this example employs a couple features of Backpack we must first introduce—*thinning* and *renaming*.[3]

The two packages arrays-a and arrays-b provide two distinct implementations of the Array module described by the hole specification in the earlier arrays-sig package. The next two packages grab the Graph implementation from structures and implement its Array hole with the respective array implementations. Since structures also defines Set and Tree, these (unwanted) modules would naively be included along with Prelude and Array and would thus pollute the namespaces of graph-a and graph-b. Instead, these packages *thin* the structures package upon inclusion so that only the desired modules, Prelude and Array, are added to graph-a and graph-b. (This closely resembles the *import* lists of Haskell modules, which may select specific entities to be imported.) Similarly, implementation details of a package definition can be hidden—rather than provided to clients—by thinning the definition to expose only certain module names. (This closely resembles the *export* lists of Haskell modules.) By thinning their definitions to expose only Prelude and Graph, both packages graph-a and graph-b hide the internal Array modules used to implement their Graph modules.

At this point, graph-a and graph-b provide distinct instantiations of the Graph module from structures, distinct in the sense that they do not have the same *module identity*. The identity of a module—a crucial notion in Backpack's semantics (see §3.1)—essentially encodes a dependency graph of the module's source code. Since the Graph modules in graph-a and graph-b import two different module sources for the Array hole—one from arrays-a and the other from arrays-b—they do not share the same dependency graph and hence have distinct identities.

Thus, if the final package multinst were to naively include both graph-a and graph-b, Backpack would complain that multinst was trying to merge two distinct implementations with the same name. To avoid this error, the inclusions of graph-a and graph-b employ *renaming* clauses that rename Graph to GA and GB, respectively, so that the two Graph implementations do not clash.

One may wonder whether it is necessary to track dependency information in module identities: why not just generate fresh module identities to represent each instantiation of a package? To see the motivation for tracking more precise dependency information, consider the example in Figure 2.3. Both the applic-left and applic-right packages *separately* instantiate the Graph module from structures with the *same* Array implementation from arrays-a—*i.e.*, both instantiations refer to the same identity for Array. Backpack thus treats the two resulting Graph modules (and their `G` types) as one and the same, which means the code in applic-bot is well-

3 In our examples so far, we have omitted thinning specs entirely. But actually, according to Figure 2.1, all package definitions and inclusions should contain a thinning spec. Thinning and renaming can be inferred from context; see §8.1.1 for more details.

```
package prelude-sig where
  Prelude :: [ data List a = Nil | Cons a (List a) ]

package arrays-sig where
  include prelude-sig
            ⎡   import Prelude              ⎤
  Array ::  ⎢ data Arr (i::*) (e::*)        ⎥
            ⎣ something :: List (Arr i e)   ⎦

package structures where
  include prelude-sig
  include arrays-sig
  Set    = [import Prelude; data S ...]
  Graph  = [import Prelude; import Array; data G ...]
  Tree   = [import Prelude; import Graph; data T ...]

package arrays-a where
  include prelude-sig
            ⎡   import qualified Prelude as P ⎤
  Array =   ⎢ data Arr i e = MkArr ...        ⎥
            ⎣ something = P.Nil               ⎦

package arrays-b where
  include prelude-sig
            ⎡   import Prelude              ⎤
  Array =   ⎢ data Arr i e = ANil | ...     ⎥
            ⎣ something = Cons ANil Nil     ⎦

package graph-a (Graph, Prelude) where
  include arrays-a
  include structures (Graph, Prelude, Array)

package graph-b (Graph, Prelude) where
  include arrays-b
  include structures (Graph, Prelude, Array)

package multinst where
  include graph-a ⟨Graph ↦ GA⟩
  include graph-b ⟨Graph ↦ GB⟩
            ⎡   import qualified GA               ⎤
  Client =  ⎢ import qualified GB                 ⎥
            ⎢ export (main, GA.G)                 ⎥
            ⎣ main = ... GA.G ... GB.G ...        ⎦
```

Figure 2.2: Running example: Data structures library and client.

```
package applic-left (Prelude,Left) where
  include structures
  include arrays-a
  Left = [ import Graph; x :: G = ... ]

package applic-right (Prelude,Right) where
  include arrays-a
  include structures
  Right = [ import Graph; f :: G -> G = ... ]

package applic-bot where
  include applic-left
  include applic-right
  Bot = [ import Left; import Right; ... f x ... ]
```

Figure 2.3: Example of applicativity.

typed. In other words, the identity of Graph inside applic-left is equivalent to that of Graph inside applic-right, and thus the `G` types mentioned in both packages are compatible.

As this example indicates, our treatment of identity instantiation exhibits sharing behavior. We call this an *applicative* semantics of identity instantiation, as opposed to a potential *generative* semantics in which the two instantiations—even when instantiated with the same identity—would produce distinct identities.

As is well known in the ML modules literature[4], applicativity enables significantly more flexibility regarding when module instantiation must occur in the hierarchy of dependencies. In the previous example, the authors of applic-left and applic-right were free to instantiate Graph *inside* their own packages. Under a generative semantics, on the other hand, in order to get the same Graph instantiation in both packages, it would need to be instantiated in an earlier package (like graph-a from Figure 2.2) and then included in both applic-left and applic-right; hence, the code as written in Figure 2.3 would under a generative semantics produce two distinct Graph identities and `G` types. As Rossberg *et al.* have noted[5], applicative semantics is generally safe only when used in conjunction with purely functional modules. It is thus ideally suited to Haskell, which isolates computational effects monadically.

2.5 ALIASES

Occasionally one wants to link two holes whose names differ. The binding form $p = p$ in Figure 2.1 allows the programmer to add such aliases, which may be viewed as sharing constraints. For example:

```
package share where
  include foo1 (A, X)
  include foo2 (B, Y)
  X = Y
```

Here, A (from foo1) depends on hole X, and B (from foo2) on hole Y, and we want to require the two holes to be ultimately instantiated by the same module. The binding X = Y expresses this constraint.

4 Leroy (1995) and Rossberg *et al.* (2014).
5 Rossberg *et al.* (2014), “F-ing modules”.

2.6 RECURSIVE MODULES

By using holes as "forward declarations" of implementations, packages can define recursive modules, *i.e.*, modules that transitively import themselves. The Haskell Language Report ostensibly allows recursive modules, but it leaves them almost entirely unspecified, letting Haskell implementations decide how to handle them. Our approach to handling recursive modules follows that of MixML.

The example below defines two modules, A and B, which import each other. By forward-declaring the parts of B that A depends on, the first implementation makes sense—*i.e.*, it knows the names and types of entities it imports from B—and, naturally, the second implementation makes sense after that. This definition is analogous to how these modules would be defined in GHC today.[6]

package ab-rec **where**
  B :: $[S_B]$
  A = [import B; ...]
  B = [import A; ...]

Normal mixin linking ties the recursive knot, ensuring that the import B actually resolves to the B implementation in the end.

GHC allows recursive modules only within a single (Cabal) package. Backpack, on the other hand, allows more flexible recursion. Although packages themselves are not defined recursively, they may be recursively *linked*. Consider the following:

package ab-sigs **where**
  A :: $[S_A]$
  B :: $[S_B]$

package b-from-a **where**
  include ab-sigs
  B = [import A; ...]

package a-from-b **where**
  include ab-sigs
  A = [import B; ...]

package ab-rec-sep **where**
  include a-from-b
  include b-from-a

At the level of packages, these definitions do not involve any recursive inclusion, which is good, because that would be illegal! Rather, they form a diamond dependency, like the earlier packages top, left, right, and bottom. There is no recursion *within* the definitions of ab-sigs, a-from-b, and b-from-a either. The recursion instead occurs implicitly, as a result of the mixin linking of modules A and B in the package ab-rec-sep. (Separately typechecked, recursive units may be defined in MixML in roughly the same way.)

Finally, note that Backpack's semantics (presented in the next chapter) explicitly addresses one of the key stumbling blocks in supporting recursive linking in the presence of abstract data types, namely the so-called *double vision problem*[7]. In the context of the above example, the problem is that, in ab-sigs, the specification S_A of the hole A may specify an abstract type T, which S_B then depends on in the types of its core-level entities. Subsequently, in a-from-b, when the implementation of A imports B, it will want to know that the type T that it defines is the same as the one mentioned in S_B, or else it will suffer from "double vision", seeing two distinct names for the same underlying type. Avoiding double vision is known to be challenging[8], but crucial for enabling common patterns of recursive module programming. Backpack's semantics avoids double vision completely.

6 In GHC, instead of explicit bindings to a signature and two modules, there would be the two module source files and an additional "boot file" for B that looks exactly like S_B. Moreover, the import B within the A module would include a "source pragma" that tells the compiler to import the boot file instead of the full module.
7 Crary *et al.* (1999) and Dreyer (2007a).
8 Dreyer (2007a,b) and Rossberg and Dreyer (2013).

3 MAKING SENSE OF BACKPACK

The previous chapter introduced Backpack's syntax and design at a high level, along with key features of the language like abstraction via holes and instantiation via package inclusion. What is the semantics of Backpack that gives meaning to those features? How does Backpack make sense of them in combination with each other? In this chapter I aim to answer these questions with enough technical detail to convey the main ideas, leaving the full details to the formalization in Part II.[1]

* * *

The main top-level judgment defining the semantics of Backpack is

$$\Delta \vdash D : \forall\overline{\alpha}.\Xi \;\; \boxed{\rightsquigarrow \lambda\overline{\alpha}.dexp}$$

Given a *package definition* D, along with a *package environment* Δ describing the types and elaborations of other packages on which D depends, this judgment ascribes D a *package type* $\forall\overline{\alpha}.\Xi$, and also elaborates D into a *parameterized directory expression* $\lambda\overline{\alpha}.dexp$, which is essentially a set of well-typed Haskell module files.

The above judgment is implemented by a two-pass algorithm. The first pass, called *shaping*, synthesizes a *package shape* $\widehat{\Xi}$ for D, which effectively explains the *macro-level* structure of the package, *i.e.*, the modules contained in D, the names of all the entities defined in those modules, and how they all depend on one another. The second pass, called *typing*, augments the structural information in $\widehat{\Xi}$ with additional information about the *micro-level* structure of D. In particular, it fills in the types of core-language entities, forming a *package type* Ξ and checking that D is well-formed at Ξ. As discussed in §1.2.4, the proper handling of recursive modules requires the "static pass" as a key ingredient in order to handle *double vision*; the shaping pass is exactly that.

Central to both passes of Backpack typechecking is a notion of *module identity*. Using the multinst package (and its dependencies) from Figure 2.2 as a running example, we will motivate the role and structure of module identities.

3.1 MODULE IDENTITIES

Figure 3.2 shows the *shapes* and *types* of multinst and its dependencies. We proceed by explaining Figure 3.2 in a left-to-right fashion.

The first column of Figure 3.2 contains the first key component of package types: a mapping from modules' *logical* names ℓ (*i.e.*, their names at the level of Backpack) to their *physical* identities ν (*i.e.*, the names of the Haskell modules to which they elaborate). The reason for distinguishing between logical names and physical identities is simple: due to aliasing (§2.5), there may be multiple logical names for the same physical module.

In order to motivate the particular logical mappings in Figure 3.2, let us first explore what physical identities are, which means reviewing how module names work in Haskell.

MODULE NAMES IN HASKELL Modules in Haskell have fixed names, which we call "physical" because they are globally unique in a program, and module definitions may then depend

1 This chapter is adapted from §3 in the original Backpack paper, accounting for changes in the formalization of Backpack. It presents the semantics of Backpack's *package level*, which will be more definitively covered, albeit with fewer examples, in Chapter 8. The substantial addition of *worlds* to Backpack's semantics, an addition since the original presentation of Backpack that this chapter is based on, will be introduced in Chapter 4.

Identity Variables	$\alpha, \beta \in \mathit{IdentVars}$
Identity Constructors	$\mathcal{K} \in \mathit{IdentCtors}$
Identities	$\nu ::= \alpha \mid \mu\alpha.\mathcal{K}\ \overline{\nu}$
Identity Substitutions	$\theta ::= \{\overline{\alpha := \nu}\}$

Figure 3.1: Module identities.

on one another by importing these physical names. Modules serve two related roles: (1) as points of origin for core-level entities, and (2) as syntactic namespaces. Concerning (1), a module may *define* new entities, such as values or abstract data types. Concerning (2), a module may *export* a set of entities, some of which it has defined itself and others of which it has imported from other modules. For example, a module Foo may define a data type named `T`. A subsequent module Bar may then import `Foo.T` and re-export it as `Bar.T`. To ensure that type identity is tracked properly, the Haskell type system models each core-level entity semantically as a pair $[\nu]$`T` of its core-level name `T` and its *provenance* ν, *i.e.*, the module that originally defined it (in this example, Foo). Thus, `Foo.T` and `Bar.T` will be viewed as equal by Haskell since they are both just different names for the same semantic entity [Foo]`T`.

To ensure compatibility with Haskell, our semantics for Backpack inherits Haskell's use of physical names to identify abstract types. However, Haskell's flat physical module namespace is not expressive enough to support Backpack's holes, applicative module instantiation, and recursive linking. To account for these features, we enrich the language of physical names with a bit more interesting structure. Figure 3.1 displays this enriched language of—as we call them—*physical module identities*.[2]

VARIABLE AND APPLICATIVE IDENTITIES Physical module identities ν are either (1) *variables* α, which are used to represent holes; (2) *applications* of identity *constructors* $\mathcal{K}$, which are used to model dependency of modules on one another, as needed to implement applicative instantiation; or (3) *recursive* module identities, defined via μ-constructors. We start by explaining the first two.

Each explicit module expression [M] that occurs in a package definition corresponds (statically) to a globally unique identity constructor $\mathcal{K}$ that encodes it. For example, if a single module source M appears on the right-hand side of three distinct module bindings in a package P, then the three distinct identity constructors of those modules are, roughly, $\langle$P.M.1$\rangle$, $\langle$P.M.2$\rangle$, and $\langle$P.M.3$\rangle$.[3]

In the absence of recursive modules, each module identity ν is then a finite tree term—either a variable α, or a constructor $\mathcal{K}$ applied to zero or more subterms, $\overline{\nu}$. The identity of a module is the constructor $\mathcal{K}$ that encodes its source M, applied to the n identities to which M's n import statements resolved (in order). For instance, in the very first example from Chapter 2, ab-1, the identities of A and B are $\mathcal{K}_a$ and $\mathcal{K}_b\ \mathcal{K}_a$, respectively, where $\mathcal{K}_a$ encodes the first module expression and $\mathcal{K}_b$ the second. In a package with holes, each hole gets a fresh variable (within the package definition) as its identity; in abcd-holes-1 the identities of the four modules are, in order, α_a, α_b, $\mathcal{K}_c$, and $\mathcal{K}_d\ \alpha_a\ \mathcal{K}_c$.

Consider now the module identities in the Graph instantiations in multinst, as shown in Figure 3.2. In the definition of structures, assume that the variables for Prelude and Array are α_P and α_A respectively, and that M_G is the module source that Graph is bound to. Then the identity of Graph is $\nu_G = \langle\text{structures}.M_G\rangle\ \alpha_P\ \alpha_A$. Similarly, the identities of the two array implementations in Figure 2.2 are $\nu_{AA} = \langle\text{arrays-a}.M_A\rangle\ \alpha_P$ and $\nu_{AB} = \langle\text{arrays-b}.M_B\rangle\ \alpha_P$.

2 To make use of these enriched physical names in our elaboration, we embed them into the space of Haskell's physical names; see §3.4.

3 We write simply $\langle$P.M$\rangle$, eliding the integer part of the identity constructor, when only one instance of [M] exists in the definition of package P.

The package graph-a is more interesting because it *links* the packages arrays-a and structures together, with the implementation of Array from arrays-a *instantiating* the hole Array from structures. This linking is reflected in the identity of the Graph module in graph-a: whereas in structures it was

$$\nu_G = \langle \mathsf{structures}.M_G \rangle\ \alpha_P\ \alpha_A,$$

in graph-a it is

$$\nu_{GA} = \nu_G[\nu_{AA}/\alpha_A] = \langle \mathsf{structures}.M_G \rangle\ \alpha_P\ \nu_{AA}.$$

Similarly, the identity of Graph in graph-b is

$$\nu_{GB} = \nu_G[\nu_{AB}/\alpha_A] = \langle \mathsf{structures}.M_G \rangle\ \alpha_P\ \nu_{AB}.$$

Thus, linking consists of substituting the variable identity of a hole by the concrete identity of the module filling that hole.

Lastly, multinst makes use of both of these Graph modules, under the aliases GA and GB, respectively. Consequently, in the Client module, GA.G and GB.G will be correctly viewed as distinct types since they originate in modules with distinct identities.

As multinst illustrates, module identities effectively encode dependency graphs. The primary motivation for encoding this information in identities is our desire for an *applicative* semantics of instantiation, as needed for instance in the example of Figure 2.3. In that example, both the packages applic-left and applic-right individually link arrays-a with structures. The client package applic-bot subsequently wishes to use both the Left module from applic-left and the Right module from applic-right, and depends on the fact that both modules operate over the same Graph.G type. This fact will be checked when the packages applic-left and applic-right are both **include**d in the same namespace of applic-bot, and the semantics of mixin linking will insist that their Graph modules have the same identity. Thanks to the dependency tracking in our module identities, we know that the Graph module has identity ν_{GA} in both packages.

RECURSIVE MODULE IDENTITIES In the presence of recursive modules, module identities are no longer simple finite trees.

Consider again the ab-rec-sep example from §2.6. (Although this example does not concern our current focus, multinst, the careful treatment of recursive module identities deserves explanation.) Suppose that ν_A and ν_B are the identities of A and B, and that M_A and M_B are those modules' defining module expressions, respectively. Because M_A imports B and M_B imports A, the two identities should satisfy the recursive equations

$$\begin{aligned}\nu_A &= \langle \mathsf{a\text{-}from\text{-}b}.M_A \rangle\ \nu_B\\ \nu_B &= \langle \mathsf{b\text{-}from\text{-}a}.M_B \rangle\ \nu_A\end{aligned}$$

These identity equations have no solution in the domain of finite trees, but they do in the domain of regular, *infinite* trees, which we denote (finitely) as

$$\begin{aligned}\nu_A &= \mu\alpha_A.\langle \mathsf{a\text{-}from\text{-}b}.M_A \rangle\ (\langle \mathsf{b\text{-}from\text{-}a}.M_B \rangle\ \alpha_A)\\ \nu_B &= \mu\alpha_B.\langle \mathsf{b\text{-}from\text{-}a}.M_B \rangle\ (\langle \mathsf{a\text{-}from\text{-}b}.M_A \rangle\ \alpha_B)\end{aligned}$$

The semantics of Backpack relies on the ability to perform both *unification* and *equivalence* testing on identities. In the presence of recursive identities, however, simple unification and syntactic equivalence of identities no longer suffices since, *e.g.*, the identity $\langle \mathsf{a\text{-}from\text{-}b}.M_A \rangle\ \nu_B$ represents the exact same module as ν_A, albeit in a syntactically distinct way. Fortunately, we can use Huet's well-known unification algorithm for regular trees instead.[4] More details can be found in Appendix §A.1.1.

4 Gauthier and Pottier (2004) and Huet (1976).

ALPHA RENAMING AND FRESHNESS In the examples so far, care has been taken to create *fresh* module identity variables for each new hole, *e.g.*, α_A distinct from α_P. Indeed, as we'll see in the next section, freshness of module identity variables is actually part of the definition of the shaping pass. But what does this mean exactly?

As indicated by the μ notion for recursive module identities, we presume conventional alpha conversion[5] on module identities with respect to the μ binder—and, later, $\forall$ and λ binders. For example, the recursive module identity ν below can be alpha-converted in its *bound* variable, α_1, resulting in an identical module identity:

$$\nu \;=\; \mu\alpha_1.\mathcal{K}_2\ (\mathcal{K}_1\ \alpha_1)\ \alpha_2 \;=\; \mu\alpha_1'.\mathcal{K}_2\ (\mathcal{K}_1\ \alpha_1')\ \alpha_2$$

The *free* variable in ν, α_2, cannot be so converted.

With alpha conversion in mind, the assignment of fresh variables for holes becomes clearer. Consider a package double-prelude-sig that simply splits the Prelude hole from prelude-sig into two distinct holes P_1 and P_2 by including the package twice:

package double-prelude-sig **where**
  include prelude-sig ⟨Prelude $\mapsto$ P_1⟩
  include prelude-sig ⟨Prelude $\mapsto$ P_2⟩

As we'll see in the next section, the type of the prelude-sig package is $\forall\alpha_P.\Xi_P$, where $\Xi_P = (\ldots\alpha_P\ldots)$. When designating a module identity variable for its hole P_1 in the first inclusion, we might choose α_1, a variable that doesn't exist yet, *i.e.*, the variables of the implicit ambient context of module identities do not already include α_1. Then when designating a variable for the hole P_2 in the second inclusion, we must choose some distinct variable that's not equal to α_1, which now does exist in the implicit ambient context; let's choose α_2. By alpha-converting the type of prelude-sig to match each fresh variable, we instantiate its package type to be $\Xi_P[\alpha_1/\alpha_P]$ for the first **include** binding and $\Xi_P[\alpha_2/\alpha_P]$ for the second.[6] The point is that we must choose two distinct variables for the α_P, not the same variable twice! And shaping enforces that requirement.

3.2 SHAPING

Constructing the mapping from logical names to physical identities is but one part of a larger task we call *shaping*, which constitutes the most unusual and interesting part of Backpack's type system.

The goal of shaping is to compute the shape (*i.e.*, the macro-level structure) of the package. Formally, a package shape $\widehat{\Xi} = (\widehat{\Phi};\mathcal{L})$ has two parts.[7] The first is a *physical shape context* $\widehat{\Phi} = \overline{\nu{:}\hat{\tau}^m}$, which, for each module in the package, maps its physical identity ν to a *polarity* m and a *module shape* $\hat{\tau}$. The polarity m specifies whether the module ν is implemented $(+)$ or a hole $(-)$. The module shape $\hat{\tau} = \langle\, \overline{\widehat{dspc}}\ ;\ \overline{espc}\,\rangle$ enumerates ν's *defined entities* $\overline{\widehat{dspc}}$—*i.e.*, the entities that the module ν itself defines—as well as *export specs* $\overline{espc}$, which list the names and provenances of the entities that ν exports.[8] Note that these are not the same thing: a module ν may import and re-export entities that originated in (*i.e.*, whose provenances are) some other modules $\overline{\nu'}$, and it may also choose *not* to export all of the entities that it defines

5 I write "alpha conversion" instead of "α conversion" to distinguish it from the metavariable for module identities.
6 This technical point is realized in the shaping rule for **include** bindings, (ShInc), which abuses notation by assuming that the package type $\forall\overline{\alpha}.\Xi$ is α-converted so that the $\overline{\alpha}$ are exactly the freshly chosen variables.
7 We write a hat ($\hat{\cdot}$) on the metavariables of certain shape objects (*e.g.*, $\hat{\tau}$) not to denote a meta-level operation, but to highlight these objects' similarity to their corresponding type objects (*e.g.*, τ).
8 In the full definitions of the formalization, in Part II, module shapes/types also include the vector of imported module identities, written N. They're omitted in this chapter and its examples for brevity. See the full semantic objects of the module level in §7.3.

	Logical Mapping	Physical Shapes	Physical Types
prelude-sig	Prelude $\mapsto \alpha_P$	$\hat\Phi_P$	Φ_P
arrays-sig	Prelude $\mapsto \alpha_P$ Array $\mapsto \alpha_A$	$\hat\Phi_P, \hat\Phi_A$	Φ_P, Φ_A
structures	Prelude $\mapsto \alpha_P$ Array $\mapsto \alpha_A$ Set $\mapsto \nu_S$ Graph $\mapsto \nu_G$ Tree $\mapsto \nu_T$	$\hat\Phi_P, \hat\Phi_A,$ $\nu_S{:}\langle\ \texttt{S(...)}\,;[\nu_S]\texttt{S(...)}\ \rangle^+$ $\nu_G{:}\langle\ \texttt{G(...)}\,;[\nu_G]\texttt{G(...)}\ \rangle^+$ $\nu_T{:}\langle\ \texttt{T(...)}\,;[\nu_T]\texttt{T(...)}\ \rangle^+$	$\Phi_P, \Phi_A,$ $\nu_S{:}\langle\!\vert\ \texttt{data S ...}\,;[\nu_S]\texttt{S(...)}\ \vert\!\rangle^+$ $\nu_G{:}\langle\!\vert\ \texttt{data G ...}\,;[\nu_G]\texttt{G(...)}\ \vert\!\rangle^+$ $\nu_T{:}\langle\!\vert\ \texttt{data T ...}\,;[\nu_T]\texttt{T(...)}\ \vert\!\rangle^+$
arrays-a	Prelude $\mapsto \alpha_P$ Array $\mapsto \nu_{AA}$	$\hat\Phi_P, \hat\Phi_{AA}$	Φ_P, Φ_{AA}
arrays-b	Prelude $\mapsto \alpha_P$ Array $\mapsto \nu_{AB}$	$\hat\Phi_P, \hat\Phi_{AB}$	Φ_P, Φ_{AB}
graph-a	Prelude $\mapsto \alpha_P$ Graph $\mapsto \nu_{GA}$	$\hat\Phi_P, \hat\Phi_{AA},$ $\nu_{GA}{:}\langle\ \texttt{G(...)}\,;[\nu_{GA}]\texttt{G(...)}\ \rangle^+$	$\Phi_P, \Phi_{AA},$ $\nu_{GA}{:}\langle\!\vert\ \texttt{data G ...}\,;[\nu_{GA}]\texttt{G(...)}\ \vert\!\rangle^+$
graph-b	Prelude $\mapsto \alpha_P$ Graph $\mapsto \nu_{GB}$	$\hat\Phi_P, \hat\Phi_{AB},$ $\nu_{GB}{:}\langle\ \texttt{G(...)}\,;[\nu_{GB}]\texttt{G(...)}\ \rangle^+$	$\Phi_P, \Phi_{AB},$ $\nu_{GB}{:}\langle\!\vert\ \texttt{data G ...}\,;[\nu_{GB}]\texttt{G(...)}\ \vert\!\rangle^+$
multinst	Prelude $\mapsto \alpha_P$ GA $\mapsto \nu_{GA}$ GB $\mapsto \nu_{GB}$ Client $\mapsto \nu_C$	$\hat\Phi_P, \hat\Phi_{AA}, \hat\Phi_{AB},$ $\nu_{GA}{:}\langle\ \texttt{G(...)}\,;[\nu_{GA}]\texttt{G(...)}\ \rangle^+$ $\nu_{GB}{:}\langle\ \texttt{G(...)}\,;[\nu_{GB}]\texttt{G(...)}\ \rangle^+$ $\nu_C{:}\langle\ \texttt{main}\,;[\nu_C]\texttt{main}, [\nu_{GA}]\texttt{G()}\ \rangle^+$	$\Phi_P, \Phi_{AA}, \Phi_{AB},$ $\nu_{GA}{:}\langle\!\vert\ \texttt{data G ...}\,;[\nu_{GA}]\texttt{G(...)}\ \vert\!\rangle^+$ $\nu_{GB}{:}\langle\!\vert\ \texttt{data G ...}\,;[\nu_{GA}]\texttt{G(...)}\ \vert\!\rangle^+$ $\nu_C{:}\langle\!\vert\ \texttt{main} :: \texttt{...}\,;[\nu_C]\texttt{main}, [\nu_{GA}]\texttt{G()}\ \vert\!\rangle^+$

$$\nu_{AA} \triangleq \langle \textsf{arrays-a}.\mathcal{M}_A\rangle\ \alpha_P \qquad \nu_S \triangleq \langle \textsf{structures}.\mathcal{M}_S\rangle\ \alpha_P \qquad \nu_T \triangleq \langle \textsf{structures}.\mathcal{M}_T\rangle\ \alpha_P\ \nu_G$$

$$\nu_{AB} \triangleq \langle \textsf{arrays-b}.\mathcal{M}_B\rangle\ \alpha_P \qquad \nu_G \triangleq \langle \textsf{structures}.\mathcal{M}_G\rangle\ \alpha_P\ \alpha_A$$

$$\nu_{GA} \triangleq \langle \textsf{structures}.\mathcal{M}_G\rangle\ \alpha_P\ \nu_{AA} \qquad \nu_C \triangleq \langle \textsf{multinst}.\mathcal{M}_C\rangle\ \nu_{GA}\ \nu_{GB}$$

$$\nu_{GB} \triangleq \langle \textsf{structures}.\mathcal{M}_G\rangle\ \alpha_P\ \nu_{AB}$$

$$\hat\Phi_P \triangleq \begin{pmatrix} \alpha_P{:}\langle & \cdot & ;[\beta_{PL}]\texttt{List(Nil,Cons)}\ \rangle^- \\ \beta_{PL}{:}\langle & \texttt{List(Nil,Cons)} & ;[\beta_{PL}]\texttt{List(Nil,Cons)}\ \rangle^- \end{pmatrix}$$

$$\Phi_P \triangleq \begin{pmatrix} \alpha_P{:}\langle\!| & \cdot & ;[\beta_{PL}]\texttt{List(Nil,Cons)}\ |\!\rangle^- \\ \beta_{PL}{:}\left\langle\!\middle| \begin{array}{c} \texttt{data List(a} :: * \texttt{)} = \\ \texttt{Nil | Cons a (}[\beta_{PL}]\texttt{List a)} \end{array}\right. & ;[\beta_{PL}]\texttt{List(Nil,Cons)} & \left|\!\vphantom{\begin{array}{c}a\\a\end{array}}\right\rangle^- \end{pmatrix}$$

$$\hat\Phi_A \triangleq \begin{pmatrix} \alpha_A{:}\langle & \cdot & ;[\beta_{AA}]\texttt{Arr()}, [\beta_{AS}]\texttt{something}\ \rangle^- \\ \beta_{AA}{:}\langle & \texttt{Arr} & ;\quad [\beta_{AA}]\texttt{Arr()}\ \rangle^- \\ \beta_{AS}{:}\langle & \texttt{something} & ;\quad [\beta_{AS}]\texttt{something}\ \rangle^- \end{pmatrix}$$

$$\Phi_A \triangleq \begin{pmatrix} \alpha_A{:}\langle\!| & \cdot & ;[\beta_{AA}]\texttt{Arr()}, [\beta_{AS}]\texttt{something}\ |\!\rangle^- \\ \beta_{AA}{:}\langle\!| & \texttt{data Arr (i} :: *\texttt{) (e} :: *\texttt{)} & ;\quad [\beta_{AA}]\texttt{Arr()}\ |\!\rangle^- \\ \beta_{AS}{:}\langle\!| & \texttt{something} :: [\beta_{PL}]\texttt{List (}[\beta_{AA}]\texttt{Arr i e)} & ;\quad [\beta_{AS}]\texttt{something}\ |\!\rangle^- \end{pmatrix}$$

$$\hat\Phi_{AA} \triangleq \nu_{AA}{:}\langle\quad \texttt{Arr(MkArr), something} \quad ; \quad [\nu_{AA}]\texttt{Arr(MkArr)}, [\nu_{AA}]\texttt{something}\quad \rangle^+$$

$$\Phi_{AA} \triangleq \nu_{AA}{:}\left\langle \begin{array}{l} \texttt{data Arr (i} :: *\texttt{) (e} :: *\texttt{)} = \texttt{MkArr ...} \\ \texttt{something} :: [\beta_{PL}]\texttt{List (}[\nu_{AA}]\texttt{Arr i e)} \end{array} \quad ; \quad \begin{array}{c} [\nu_{AA}]\texttt{Arr(MkArr)} \\ [\nu_{AA}]\texttt{something} \end{array} \right\rangle^+$$

$$\hat\Phi_{AB} \triangleq \nu_{AB}{:}\langle\quad \texttt{Arr(ANil,...), something} \quad ; \quad [\nu_{AB}]\texttt{Arr(ANil,...)}, [\nu_{AB}]\texttt{something}\quad \rangle^+$$

$$\Phi_{AB} \triangleq \nu_{AB}{:}\left\langle \begin{array}{l} \texttt{data Arr (i} :: *\texttt{) (e} :: *\texttt{)} = \texttt{ANil|...} \\ \texttt{something} :: [\beta_{PL}]\texttt{List (}[\nu_{AB}]\texttt{Arr i e)} \end{array} \quad ; \quad \begin{array}{c} [\nu_{AB}]\texttt{Arr(ANil,...)} \\ [\nu_{AB}]\texttt{something} \end{array} \right\rangle^+$$

Figure 3.2: Example package types and shapes for the multinst package and its dependencies. In each module shape/type the imported module identities (N) have been omitted for brevity.

Core Level:	
Value Names	$x \in \mathit{ValNames}$
Type Names	$T \in \mathit{TypeNames}$
Constructor Names	$K \in \mathit{CtorNames}$
Entity Names	$\chi ::= x \mid T \mid K$
Kind Environments	$\mathit{kenv} ::= \ldots$
Semantic Types	$\mathit{typ} ::= [\nu]T\,\overline{\mathit{typ}} \mid \ldots$
Defined Entity Specs	$\mathit{dspc} ::= \texttt{data}\ T\ \mathit{kenv} = \overline{K\ \overline{\mathit{typ}}}$
	$\mid\ \texttt{data}\ T\ \mathit{kenv} \quad \mid\ \ x :: \mathit{typ}$
Export Specs	$\mathit{espc} ::= [\nu]\chi \mid [\nu]\chi(\overline{\chi})$
Module Level:	
Module Polarities	$m ::= + \mid -$
Module Types	$\tau, \sigma ::= \langle\!\langle\, \overline{\mathit{dspc}}\, ;\, \overline{\mathit{espc}}\, \rangle\!\rangle$
Physical Module Ctxts	$\Phi ::= \overline{\nu{:}\tau^{m}}$
Logical Module Ctxts	$\mathcal{L} ::= \overline{\ell \mapsto \nu}$
Package Level:	
Package Types	$\Xi, \Gamma ::= (\Phi; \mathcal{L})$
Package Environments	$\Delta ::= \cdot \mid \Delta,\ P = \lambda\overline{\alpha}.\mathit{dexp} : \forall\overline{\alpha}.\Xi$
Shaping Objects:	
Defined Entity Shape Specs	$\widehat{\mathit{dspc}} ::= x \mid T \mid T(\overline{K})$
Module Shapes	$\hat{\tau} ::= \langle\, \overline{\widehat{\mathit{dspc}}}\, ;\, \overline{\mathit{espc}}\, \rangle$
Physical Shape Ctxts	$\widehat{\Phi} ::= \overline{\nu{:}\hat{\tau}^{m}}$
Package Shapes	$\widehat{\Xi}, \widehat{\Gamma} ::= (\widehat{\Phi}; \mathcal{L})$

Figure 3.3: Semantic objects for shaping and typing, abridged. In particular, type classes and worlds (Chapter 4) are completely absent. More complete semantic objects will be presented later, in the chapters on the formalization of Backpack (Part II).

internally.[9] In our running example in Figure 3.2, the physical shape contexts $\widehat{\Phi}$ computed for each package are shown in the second column.

The second part of the package shape is a *logical shape context* $\mathcal{L} = \overline{\ell \mapsto \nu}$, which, for each module in the package, maps its logical name ℓ to its physical identity ν. (This is the mapping shown in the first column of Figure 3.2, which we have already discussed in detail in §3.1).

Figure 3.3 defines the semantic objects for shaping and typing, and Figure 3.4 gives some of the key rules implementing shaping.

SHAPING RULES The main shaping judgment, $\Delta \Vdash \overline{B} \Rightarrow \widehat{\Xi}$, takes as input the body of a package definition, which is just a sequence of bindings $\overline{B}$. Rule ShSeq synthesizes the shape of $\overline{B}$ by proceeding, in left-to-right order, to synthesize the shape of each individual binding B (via the judgment $\Delta; \widehat{\Gamma} \Vdash B \Rightarrow \widehat{\Xi}$) and then *merge* it with the shapes of the previous bindings (via the judgment $\Vdash \widehat{\Xi}_1 + \widehat{\Xi}_2 \Rightarrow \widehat{\Xi}$).

Let us begin with the judgment that shapes an individual binding. The rule ShAlias should be self-explanatory.

The rule ShInc is simple as well, choosing fresh identity variables $\overline{\alpha}$ to represent the holes in package P and applying the renaming r to P's shape. Note that it uses some simple aux-

9 The reader might find the distinction between *dspcs* and *espcs* in module types confusing. MixML and other systems rooted in ML have no such separation of exposed core names in the types of modules because modules in those systems are not "namespace control mechanisms." In Haskell, however, this distinction aides the typechecking of core entities: simply lookup the module type τ for the defining module ν of an imported entity $[\nu]\chi$, knowing that a *dspc* for χ exists in τ. For more justification, see the discussion in Part II on typechecking the core level (§6.4) and on unification via the provenances of *espcs* (§8.5).

iliary definitions: rename, for applying a renaming to the $\mathcal{L}$ part of a shape, and shape, for erasing a package type Ξ to a shape by removing typing information. Moreover, by alpha-converting the type of P we rename its variables to match the freshly chosen $\overline{\alpha}$, as mentioned with the double-prelude-sig example in the discussion on module identities.

The rule ShMod generates the appropriate globally unique identity ν_0 to represent [M], and then calls out to a shaping judgment for Haskell modules, $\widehat{\Gamma};\nu_0 \quad \Vdash \ _c M : \widehat{\tau}$, which generates the shape $\widehat{\tau}$ of M assuming that ν_0 is the module's identity. As an example of this, observe the shape generated for the Client module ν_C in multinst in Figure 3.2. The shape ascribes provenance ν_C to the `main` entity, since it is freshly defined in Client, while ascribing provenance ν_{GA} to the `G` type, since it was imported from GA and is only being re-exported by Client.

The rule ShSig, for shaping hole declarations, is a bit subtler than the other rules. Perhaps surprisingly, the generated shape declares not only a fresh identity variable α for the hole itself, but also a set of fresh identity variables $\overline{\beta}$, one for each entity specified in the hole signature S. (The intermediate $\widehat{\tau}_0$ merely encodes these fresh identities as input to the signature shaping judgment.) The reason for this is simply to maximize flexibility: there is no reason to demand *a priori* that the module that fills in the hole (*i.e.*, the module whose identity ν will end up getting substituted for α) must itself be responsible for *defining* all the entities specified in the hole signature—it need only be responsible for *exporting* those entities, which may very well have been defined in other modules.

The shape $\widehat{\Phi}_A$ in Figure 3.2 illustrates the output of ShSig on the Array hole in package arrays-sig. This shape specifies that β_{AA} is a module defining an entity called `Arr`, that β_{AS} is a module defining an entity called `something`, and that α_A is a module bringing $[\beta_{AA}]$`Arr` and $[\beta_{AS}]$`something` together in its export spec. Of course, when the hole is eventually filled (*e.g.*, in the graph-a package, whose shaping is discussed below), it may indeed be the case that the same module identity ν is substituted for α_A, β_{AA}, and β_{AS}—*i.e.*, that ν both defines *and* exports `Arr` and `something`—but ShSig does not require this.

Returning now to the merging judgment $\Vdash \ \widehat{\Xi}_1 + \widehat{\Xi}_2 \Rightarrow \widehat{\Xi}$ that is invoked in the last premise of (ShSeq): This merging judgment is where the real "meat" of shaping occurs—in particular, this is where mixin *linking* is performed by *unification* of module identities. If a module with logical name ℓ is mapped by $\widehat{\Xi}_1$ and $\widehat{\Xi}_2$ to physical identities ν_1 and ν_2, respectively, the merging judgment will unify ν_1 and ν_2 together. Moreover, if ν_1 and ν_2 are specified by $\widehat{\Xi}_1$ and $\widehat{\Xi}_2$ as having different module shapes $\widehat{\tau}_1$ and $\widehat{\tau}_2$, respectively, those shapes will be merged as well, with the resulting shape containing all of the components specified in either $\widehat{\tau}_1$ and $\widehat{\tau}_2$. For any entities appearing in both $\widehat{\tau}_1$ and $\widehat{\tau}_2$, their provenances will be unified.

To see a concrete instance of this, consider the merging that occurs during the shaping of the graph-a package in our running example in Figure 3.2. The graph-a package **include**s two packages defined earlier: arrays-a and structures. As per rule (ShInc), each inclusion will generate fresh identity variables for the packages' holes (say, α_P, β_{PL}, α_A, β_{AA}, β_{AS} for structures, and α'_P, β'_{PL} for arrays-a). Since both packages export Prelude, the merging judgment will unify α_P and α'_P, the physical identities associated with Prelude in the shapes of the two packages; consequently, the shape of α_P, namely $\langle \cdot ; [\beta_{PL}]$`List(Nil,Cons)` $\rangle$, will be unified with the shape of α'_P, namely $\langle \cdot ; [\beta'_{PL}]$`List(Nil,Cons)` $\rangle$, resulting in the unification of β_{PL} and β'_{PL} as well.

Similarly, since both packages export Array, the merging judgment will link the implementation of Array in arrays-a with the hole for Array in structures by unifying α_A, β_{AA}, and β_{AS} with ν_{AA}. As a result, the occurrences of α_A, β_{AA}, and β_{AS} in ν_G (and its shape) get substituted with ν_{AA}, which explains why the shape of graph-a maps Graph to $\nu_{GA} = \nu_G[\nu_{AA}/\alpha_A]$. Lastly, merging will check that the implementation of Array in arrays-a actually provides all the entities required by the hole specification in structures, *i.e.*, that $\widehat{\Phi}_{AA}$ subsumes $\widehat{\Phi}_A$, which indeed it does.

$$\boxed{\Delta;\hat{\Gamma} \Vdash B \Rightarrow \hat{\Xi}} \qquad \frac{\ell' \mapsto \nu \in \widehat{\Gamma}}{\Delta;\hat{\Gamma} \Vdash \ell = \ell' \Rightarrow (\cdot;\ell \mapsto \nu)}\ (\textsc{ShAlias})$$

$$\frac{\nu_0 = \mathsf{mkident}(M;\widehat{\Gamma}.\mathcal{L}) \qquad \widehat{\Gamma};\nu_0 \Vdash M \Rightarrow \widehat{\tau}}{\Delta;\hat{\Gamma} \Vdash \ell = [M] \Rightarrow (\nu_0{:}\widehat{\tau}^{+};\ell \mapsto \nu_0)}\ (\textsc{ShMod})$$

$$\frac{\alpha,\overline{\beta}\ \mathsf{fresh} \qquad \widehat{\Gamma};\overline{\beta} \Vdash S \Rightarrow \widehat{\sigma} \mid \widehat{\Phi}_{\mathsf{sig}}}{\Delta;\hat{\Gamma} \Vdash \ell :: [S] \Rightarrow ((\alpha{:}\widehat{\sigma}^{-}, \widehat{\Phi}_{\mathsf{sig}});\ell \mapsto \alpha)}\ (\textsc{ShSig})$$

$$\frac{\overline{\alpha}\ \mathsf{fresh} \qquad (P : \forall\overline{\alpha}.\Xi) \in \Delta \qquad \Xi' = \mathsf{rename}(r;\Xi)}{\Delta;\hat{\Gamma} \Vdash \textbf{include}\ P\ r \Rightarrow \mathsf{shape}(\Xi')}\ (\textsc{ShInc})$$

$$\boxed{\Delta \Vdash \overline{B} \Rightarrow \widehat{\Xi}} \qquad \frac{}{\Delta \Vdash \cdot \Rightarrow (\cdot;\cdot)}\ (\textsc{ShNil})$$

$$\frac{\Delta \Vdash \overline{B_1} \Rightarrow \widehat{\Xi}_1 \qquad \Delta;\widehat{\Xi}_1 \Vdash B_2 \Rightarrow \widehat{\Xi}_2 \qquad \Vdash \widehat{\Xi}_1 + \widehat{\Xi}_2 \Rightarrow \widehat{\Xi}}{\Delta \Vdash \overline{B_1}, B_2 \Rightarrow \widehat{\Xi}}\ (\textsc{ShSeq})$$

Figure 3.4: Shaping rules, abridged. A more complete definition will be presented in the formalization (Chapter 8), including package thinning and world semantics.

3.3 TYPING

In our running example thus far, we have not yet performed any *typechecking* of core-level code, such as the code inside multinst's Client module. There is a good reason for this: before shaping, we don't know whether core-level types such as GA.G and GB.G (imported by Client) are equal, because we don't know what the identities of GA and GB are. But after shaping, we have all the identity information we need to perform typechecking proper.

Thus, as seen in the top-level package rule TyPkg in Figure 3.5, the output of the shaping judgment—namely, $\widehat{\Xi}_{\mathsf{pkg}}$—is passed as input to the *typing* judgment, $\Delta;\widehat{\Xi}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \rightsquigarrow \mathit{dexp}$. Typing, in turn, produces a *package type* Ξ, which enriches the package shape $\widehat{\Xi}_{\mathsf{pkg}}$ with core-level (*i.e.*, Haskell-level) typing information. The final type returned for the package, $\forall\overline{\alpha}.\Xi$, then just quantifies over the variable identities $\overline{\alpha}$ of the holes in Ξ, so that they may be instantiated in different ways by subsequent package definitions.

The package types Ξ generated for the packages in our running example appear in the third column of Figure 3.2. Formally, the only difference between these package types and the package shapes in the second column of Figure 3.2 lies in the difference between their constituent *module* types $\tau = \langle\!\langle\, \overline{\mathit{dspc}}\,;\overline{\mathit{espc}}\,\rangle\!\rangle$ and *module* shapes $\widehat{\tau} = \langle\, \overline{\widehat{\mathit{dspc}}}\,;\overline{\mathit{espc}}\,\rangle$. Whereas the "defined entities" component ($\widehat{\mathit{dspc}}$) of $\widehat{\tau}$ only *names* the entities defined by a module, the "defined entity specs" component (dspc) of τ additionally specifies their core-level kinds/types. For example, observe the module type ascribed to arrays-a's module ν_{AA} in Φ_{AA}. This type enriches the pre-computed shape (in $\widehat{\Phi}_{AA}$) with additional information about the kind of `Arr` and the type of `something`.

Let us now explain the typing rules in Figure 3.5. For the moment, we will ignore the shaded parts of the rules concerning elaboration into Haskell; we will return to them in §3.4.

The rules TyNil and TySeq implement typing of a sequence of bindings $\overline{B}$. The procedure is structurally very similar to the one used in the shaping of $\overline{B}$: we process (in left-to-right order) each constituent binding B, producing a *type* that we *merge* into the types of the previous bindings. The key difference is that the partial merge operator $\oplus$ does not perform any unification on module identities—it merely performs a mixin merge, which checks that all specifications (kinds or types) assigned to any particular core-level entity are equal. For instance, when typing graph-a, the mixin merge will check that the type of `something` in the Array implementation from arrays-a is equal to the type of `something` in the Array hole from structures, and thus that the implementation satisfies the requirements of the hole.

$$\boxed{\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash B : \Xi \;\; \leadsto \mathit{dexp}}$$

$$\frac{\ell' \mapsto \nu \in \Gamma}{\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \ell = \ell' : (\!|\, \cdot \,;\, \ell \mapsto \nu \,|\!) \;\; \leadsto \{\}}\;(\textsc{TyAlias})$$

$$\frac{\ell \mapsto \nu_0 \in \hat{\hat{\Xi}}_{\mathsf{pkg}} \qquad \Gamma;\, \nu_0 \vdash M : \tau \leadsto \mathit{hsmod}}{\begin{array}{c}\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \ell = [M] : (\!|\, \nu_0{:}\tau^{+} \,;\, \ell \mapsto \nu_0 \,|\!) \\ \leadsto \{\nu_0{}^{\star} \mapsto \mathit{hsmod} : \tau^{\star}\}\end{array}}\;(\textsc{TyMod})$$

$$\frac{\begin{array}{c}\ell \mapsto \nu_0 \in \hat{\hat{\Xi}}_{\mathsf{pkg}} \qquad (\nu_0{:}\hat{\tau}_0^{m}\,) \in \hat{\hat{\Xi}}_{\mathsf{pkg}} \\ \Gamma;\, \hat{\tau}_0 \vdash S : \sigma \mid \Phi_{\mathsf{sig}} \qquad \Phi' = \nu_0{:}\sigma^{-} \oplus \Phi_{\mathsf{sig}} \text{ defined}\end{array}}{\begin{array}{c}\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \ell :: [S] : (\!|\, \Phi' \,;\, \ell \mapsto \nu_0 \,|\!) \\ \leadsto \{\nu^{\star} \mapsto - : \tau^{\star} \mid \nu{:}\tau^{-} \in \Phi'\}\end{array}}\;(\textsc{TySig})$$

$$\frac{\begin{array}{c}\overline{\alpha} \text{ fresh} \qquad (P{=}\lambda\overline{\alpha}.\mathit{dexp} : \forall\overline{\alpha}.\Xi) \in \Delta \qquad \Xi' {=} \mathsf{rename}(r;\Xi) \\ \vdash \hat{\hat{\Xi}}_{\mathsf{pkg}} \leqslant_{\overline{\alpha}} \Xi' \leadsto \theta\end{array}}{\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \textbf{include}\; P\; r : \mathsf{apply}(\theta;\Xi') \;\; \leadsto \mathsf{apply}(\theta^{\star};\mathit{dexp})}\;(\textsc{TyInc})$$

$$\boxed{\Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \;\; \leadsto \mathit{dexp}} \qquad \frac{}{\Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \cdot : (\!|\, \cdot \,;\, \cdot \,|\!) \;\; \leadsto \{\}}\;(\textsc{TyNil})$$

$$\frac{\begin{array}{c}\Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \overline{B_1} : \Xi_1 \;\; \leadsto \mathit{dexp}_1 \qquad \Xi = \Xi_1 \oplus \Xi_2 \text{ defined} \\ \Delta;\Xi_1;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash B_2 : \Xi_2 \;\; \leadsto \mathit{dexp}_2\end{array}}{\Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \overline{B_1}, B_2 : \Xi \;\; \leadsto \mathit{dexp}_1 \oplus \mathit{dexp}_2}\;(\textsc{TySeq})$$

$$\boxed{\Delta \vdash D : \forall\overline{\alpha}.\Xi \;\; \leadsto \lambda\overline{\alpha}.\mathit{dexp}}$$

$$\frac{\Delta \Vdash \overline{B} \Rightarrow \hat{\hat{\Xi}}_{\mathsf{pkg}} \qquad \Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \;\; \leadsto \mathit{dexp} \qquad \overline{\alpha}{=}\mathsf{fv}(\Xi)}{\Delta \vdash \textbf{package}\; P\; \textbf{where}\; \overline{B} : \forall\overline{\alpha}.\Xi \;\; \leadsto \lambda\overline{\alpha}.\mathit{dexp}}\;(\textsc{TyPkg})$$

Figure 3.5: Typing and elaboration rules (ignoring thinning).

(Module Names)	$\mathsf{f} \in \mathit{IlModNames}$
(Module Sources)	$\mathit{hsmod} ::= \ldots$
(File Expressions)	$\mathit{fexp} ::= \mathit{hsmod} \mid -$
(File Types)	$\mathit{ftyp} ::= \langle\!\langle\, \overline{\mathit{dspc}^\star} \,;\, \overline{\mathit{espc}^\star} \,\rangle\!\rangle$
(Typed File Expressions)	$\mathit{tfexp} ::= \mathit{fexp} : \mathit{ftyp}$
(Directory Expressions)	$\mathit{dexp} ::= \{\overline{\mathsf{f} \mapsto \mathit{tfexp}}\}$
(Identity Translation)	$(-)^\star \in \mathit{Identities}/{\equiv_\mu} \rightarrowtail \mathit{IlModNames}$

Figure 3.6: IL syntax, abridged. $\mathit{dspc}^\star$ and $\mathit{espc}^\star$ mention f instead of ν. A more complete definition will be presented in the formalization (Chapter 9).

The remaining rules concern the typing of individual bindings, $\Delta; \Gamma; \hat{\Xi}_{\mathsf{pkg}} \vdash \mathsf{B} : \Xi \;\rightsquigarrow\; \mathit{dexp}$. The typing rules TyMod and TySig are structurally very similar to the corresponding shaping rules given in Figure 3.4. The key difference is that, whereas ShMod and ShSig *generate* appropriate identities for their module/hole, TyMod and TySig instead *look up* the pre-computed identities in the package shape $\hat{\Xi}_{\mathsf{pkg}}$. As an example of this, observe what happens when we type the Array module in arrays-a using rule TyMod. The package shape $\hat{\Xi}_{\mathsf{pkg}}$ we pre-computed in the shaping pass tells us that the physical module identity associated with the logical module name Array is ν_{AA}, so we can go ahead and assume ν_{AA} is the identity of Array when typing its implementation. Note that TyMod and TySig call out to typing judgments for Haskell modules and signatures. Like the analogous shaping judgments, these are defined formally in Chapter 7.

Like TyMod and TySig, the rule TyInc also inspects $\hat{\Xi}_{\mathsf{pkg}}$ to determine the pre-computed identities of the modules/holes in the package P being **include**d. The only difference is that an **include**d package contains a whole bunch of subcomponents (rather than only one), so looking up their identities is a bit more involved. It is performed by appealing to a "matching" judgment $\vdash \hat{\Xi}_{\mathsf{pkg}} \leqslant_{\overline{\alpha}} \Xi' \rightsquigarrow \theta$, similar to the one needed for signature matching in ML module systems.[10] This judgment looks up the instantiations of all the **include**d holes $\overline{\alpha}$ by matching Ξ' (the type of the **include**d package P *after* applying the renaming r) against $\hat{\Xi}_{\mathsf{pkg}}$. This produces a substitution θ with domain $\overline{\alpha}$, which then gets applied to Ξ' to produce the type of the **include** binding. For example, when typing the package graph-a, we know after shaping that the identity of the Array module is ν_{AA}. When we **include** structures, the matching judgment will glean this information from $\hat{\Xi}_{\mathsf{pkg}}$, and produce a substitution θ mapping structures' α_A parameter to the actual Array implementation ν_{AA}.

3.4 ELABORATING BACKPACK TO HASKELL

We substantiate our claim to retrofit Haskell with SMD through an elaboration of Backpack, our external language (EL), into a model of GHC Haskell, our internal language (IL). The EL, as we have demonstrated so far, extends across the package, module, and core levels, while the IL defines only module and core levels; effectively the outer, package level gets "compiled away" into mere modules in the IL. Figure 3.6 gives the syntax of the IL[11]; for its semantics, including the typing judgment, see Chapter 9.

Elaboration translates a Backpack package into a *parameterized directory expression* $\lambda\overline{\alpha}.\mathit{dexp}$, which is a mapping from a set of module names f to typed file expressions *tfexp*, parameterized over the identities $\overline{\alpha}$ of the package's holes. We assume an embedding $(-)^\star$ from module identities into IL module names, which respects the equi-recursive equivalence on module identities that the Backpack type system relies on. However, for readability, we will

10 Rossberg *et al.* (2010), "F-ing Modules".

11 This presentation of the IL lacks the *worlds* introduced in Chapter 4. For the full syntax, see the formalization in Chapter 9.

$$\lambda\alpha_P\,\beta_{PL}\,.\left\{\begin{array}{lll}
\alpha_P & \mapsto & \underline{\hspace{4cm}} \\
\beta_{PL} & \mapsto & \underline{\hspace{4cm}} \\
\nu_{AA} & \mapsto & \left(\begin{array}{l}
\text{module } \nu_{AA}\ \texttt{(Arr(MkArr))} \text{ where} \\
\quad \text{import qualified } \alpha_P \text{ as P } \texttt{(List(Nil,Cons))} \\
\quad \texttt{data Arr i e = MkArr ...} \\
\quad \texttt{something = } \text{P.Nil} \texttt{ :: } \text{P.List} \texttt{ (Arr i e)}
\end{array}\right) \\
\nu_{AB} & \mapsto & \left(\begin{array}{l}
\text{module } \nu_{AB}\ \texttt{(Arr(ANil, ...))} \text{ where} \\
\quad \text{import } \alpha_P \text{ as Prelude } \texttt{(List(Nil, Cons))} \\
\quad \texttt{data Arr i e = ANil | ...} \\
\quad \texttt{something = Cons ANil Nil}
\end{array}\right) \\
\nu_{GA} & \mapsto & \left(\begin{array}{l}
\text{module } \nu_{GA}\ \texttt{(G(...))} \text{ where} \\
\quad \text{import } \alpha_P \text{ as Prelude } \texttt{(List(Nil, Cons))} \\
\quad \text{import } \nu_{AA} \text{ as Array } \texttt{(Arr(), something)} \\
\quad \texttt{data G ...}
\end{array}\right) \\
\nu_{GB} & \mapsto & \left(\begin{array}{l}
\text{module } \nu_{GB}\ \texttt{(G(...))} \text{ where} \\
\quad \text{import } \alpha_P \text{ as Prelude } \texttt{(List(Nil, Cons))} \\
\quad \text{import } \nu_{AB} \text{ as Array } \texttt{(Arr(), something)} \\
\quad \texttt{data G ...}
\end{array}\right) \\
\nu_C & \mapsto & \left(\begin{array}{l}
\text{module } \nu_C\ \texttt{(main, } \text{GA}\texttt{.G())} \text{ where} \\
\quad \text{import qualified } \nu_{GA} \text{ as GA } \texttt{(G())} \\
\quad \text{import qualified } \nu_{GB} \text{ as GB } \texttt{(G())} \\
\quad \texttt{main = ... } \text{GA}\texttt{.G ... } \text{GB}\texttt{.G ...}
\end{array}\right)
\end{array}\right\}$$

Figure 3.7: Elaboration of multinst. (For readability, the translation from identities to module names, $(-)^\star$, and the file type annotation on each module file have been omitted. See Figure 3.2 for the latter.)

leave the embedding implicit in the remainder of this subsection. As for the typed file expressions *tfexp*, they can either be defined file expressions (*hsmod* : *ftyp*), which provide both an implementation of a module along with its type, or undefined file expressions (— : *ftyp*), which describe a hole with type *ftyp*. Thus, all components of a *dexp* are explicitly-typed. This has the benefit that the modules in a *dexp* can be typechecked in any order, since all static information about them is specified in their explicit file types.

As a continuation of our running example, Figure 3.7 displays the elaboration of the multinst package, except with the file types stripped off for brevity. First, note that each module identity ν in the physical type Φ_M of multinst (lower-right hand corner of the table in Figure 3.2) corresponds to one of the Haskell modules in the elaboration of the package, and for each ν, its type in Φ_M is (modulo the embedding $(-)^\star$) precisely the file type of ν that we have omitted from Figure 3.7. The concrete module identities in Φ_M map to defined file expressions, while the identity variables α_P and β_{PL} (representing holes) map to undefined file expressions.

The elaboration of packages (marked with shaded text) is almost entirely straightforward, following the typing rules. More interesting is the elaboration of Haskell *modules*, which is appealed to in the second premise of rule TyMod (and formalized in Chapter 7). Offhand, one might expect module elaboration to be the identity translation, but in fact it is a bit more subtle than that.

Consider the ν_C entry in the directory, corresponding to the Client module, as a concrete example.

- *The module header:* Unlike the original EL implementation of Client, which was anonymous, its elaborated IL version has a module header specifying ν_C as its fixed physical name, and `main` and GA`.G()` as its exported entities. More generally, the exported entities should reflect those listed in the module's type τ.

- *The import statements:* Our elaboration rewrites imports of logical names like GA into imports of physical module identities like ν_{GA}, since the physical identities are the actual names of Haskell modules in the elaborated directory expression. We must therefore take care to preserve the logical module names that the definitions in the module's body expect to be able to refer to. For example, the reference GA.G is seen to have provenance $[\nu_{GA}]$G during Backpack typechecking of Client, so in the elaborated IL version of Client we want GA.G to mean the same thing. We achieve this by means of Haskell's "import aliases", which support renaming of imported module names; *e.g.*, the first import statement in ν_C imports the physical name ν_{GA} but gives it the logical name GA in the body, thus ensuring that the reference GA.G still has (the same) meaning as it did during Backpack typechecking. [12]

- *The body:* Thanks to the import aliasing we just described, the entity definitions in the body of ν_C can remain syntactically identical to those in the original Client module.

- *Explicitness of imports and exports:* All imported and exported entities are given as explicitly as the Haskell module syntax allows, even when the original EL modules neglect to make them explicit; *e.g.*, the original code for Graph lists neither its imports nor its exports, but its elaboration (as ν_{GA} and ν_{GB}) does. The primary reason for this explicitness is that it enables us to prove a "weakening" property on IL modules, which is critical for the proof of soundness of elaboration. If modules are not explicit about which core-level entities they are importing and exporting, their module types will not be stable under weakening.

12 The preservation of a module's local "entity environment" under elaboration has been proven as part of the larger proof of soundness of elaboration Chapter 7.

4 CLASS STRUGGLE: TYPE CLASSES IN BACKPACK

Aside from laziness, Haskell's "most distinctive feature" is type classes.[1] Although originally intended to solve "the narrow problem of overloading of numeric operators and equality,"[2] type classes have proven to be a crucial part of the language, expanding into the realm of constraint solving and type-level computation. To give a sense for how pervasive they are in Haskell programs, in GHC version `7.8.4` (released in December 2014), the implicitly imported module providing the essentials from the standard library, Prelude, exposes a whopping 1,375 different type class instances to virtually every Haskell program.[3]

Type classes were a glaring hole in the original presentation of Backpack. For that work, "[we had] left them out of the system deliberately in order to focus attention on the essential features of Backpack that we hope will be broadly applicable, not just to Haskell."[4] (Indeed, Backpack can be seen as a framework to build strong modularity on top of *any* language that only offers weak modularity.) Adding them into the Backpack semantics required new approaches to the module type system and to the proof of soundness. One key result from that effort is a class of side conditions in the metatheory that hint at the antimodularity of type classes (§9.3).

In this chapter, I integrate type classes into the formalization of Backpack. That amounts not to any deviation from Haskell's surface syntax, but to a new specification of their semantics in the (also-new) type system for modules. As we'll see shortly, their integration revolves around a central and straightforward concept of what I call *worlds*, a concept that effectively brings the type class instances known to a module into that module's type.

But first, before introducing type classes into Backpack, I diagnose and analyze an existing problem with Haskell type classes, which I call *misabstraction*, that manifests in the GHC implementation. The worlds semantics for type classes puts the language in firmer, more modular ground for addressing misabstraction. Indeed, one might consider it a modular semantics for a non-modular language feature, *i.e.*, type classes.

4.1 TYPE CLASSES IN HASKELL

The Haskell language spec demands that type classes satisfy a *global uniqueness* restriction: "A type may not be declared as an instance of a particular class more than once in the program."[5] Since this restriction pertains to the *whole* program, it is inherently *non-modular*: Alice and Bob may independently develop modules A and B, which typecheck fine in isolation, but which nevertheless cannot be both imported by a common client C because they define overlapping instances and thus do not compose.

On the one hand, the non-modular nature of Haskell type classes is unfortunate, and as a result, several proposals—such as *named instances*[6] and *modular type classes*[7]—have been put forth in an effort to relax the global uniqueness restriction and support *local* (scoped) instance declarations. On the other hand, adapting Haskell to incorporate such proposals

1 Hudak *et al.* (2007), "A history of Haskell: Being lazy with class," p.17.
2 Ibid., p.8.
3 For the full results of this analysis, presented in §4.3.3, see: `https://gitlab.mpi-sws.org/backpack/class-struggle/blob/v3/data/worlds/prelude-readme.md`
4 Kilpatrick *et al.* (2014), "Backpack: Retrofitting Haskell with Interfaces".
5 Marlow (2010), "Haskell 2010 Language Report," §4.3.2.
6 Kahl and Scheffczyk (2001), "Named Instances for Haskell Type Classes".
7 Dreyer *et al.* (2007), "Modular Type Classes".

would constitute a serious breaking change to the language and its ecosystem because the global uniqueness restriction is deeply baked into them.

As an example, to be explored in more detail in §4.2, the abstract data type `Set a` (defined in GHC's `Data.Set` module) only behaves correctly under the assumption of global uniqueness. In particular, if A and B were permitted to locally define distinct instances of `Ord T` (for some `T`) but hide those instances from their common client C, local invariants maintained in `Data.Set` could be unwittingly violated by C, resulting in anomalous behavior. Consequently, it seems fair to assume that the global uniqueness restriction is not going away anytime soon.

The question remains: how should global uniqueness be implemented? Even though it is a non-modular property, we still need a way to implement it in a modular fashion if we wish to typecheck a program one module at a time. However, the Haskell language spec provides no guidance here, and the answer to this question is controversial. For instance, GHC attempts to enforce global uniqueness modularly by checking that each instance declaration does not overlap with any existing instances in scope at that point, and by checking for instance ambiguity at use sites. But, as I explain in §4.2, these checks do not in fact guarantee global uniqueness. Some in the Haskell community have identified so-called *orphan instances* as the culprit and argued for a prohibition on them (implemented by GHC as an optional compiler flag). But prohibiting orphan instances also rules out perfectly valid Haskell programs, and what's more, it *still* fails to ensure global uniqueness.

Developing a satisfactory answer to this question is important for Haskell, but even more crucial in the context of Backpack, a system in which (formally modeling) modularity is essential.

In this chapter, I propose a new answer to this question, based on the simple (but previously unarticulated) idea of *worlds*. A world describes the set of instances that are in scope when typechecking a given module. A world is *consistent* if these instances are non-overlapping. When multiple modules (say, A and B) are imported into the scope of a client module C, their respective worlds are joined together via a partial merge operation, which succeeds if and only if the union of their worlds is consistent. In effect, merging checks for global uniqueness *eagerly* rather than lazily (as GHC currently does), and I argue this is both necessary and desirable.

4.2 WHAT'S WRONG WITH TYPE CLASSES TODAY

Figure 4.1 presents our first example of a Haskell program that witnesses the modular ambiguity of type classes. This program is perfectly valid, compilable, and executable in the Glasgow Haskell Compiler (GHC), the *de facto* implementation of Haskell. It demonstrates the potential for bad behavior when mixing modularity and type classes; how the Haskell spec ostensibly rules out this bad behavior; how the GHC implementation (attempts to) follow the spec; and how GHC nonetheless *allows* (some of) the bad behavior.

4.2.1 *Bad behavior through modularity*

The very first module of the program, Data.Set, defines an abstract data type (ADT) for ordered sets, represented as binary trees.[8] The remaining modules form a classic diamond dependency. FunnyTop defines a type `I`, a straightforward wrapper around `Int`. FunnyLeft and FunnyRight define opposite orderings on `I` in their respective `Ord I` instances, along with instantiations of `Set`'s (generic) `insert` operation that bake in the respective orderings. Finally, FunnyBottom defines a set value `funnySet` by using both instantiations to insert values into an empty set, and then (purportedly) prints out the unique elements of that set, as a list, in ascending order. Perversely, however, the elements of the printed list are neither unique nor ordered. Something is wrong with this program.

8 Defined in the containers package: http://hackage.haskell.org/package/containers-0.5.6.3/docs/Data-Set.html

```
-- The standard Set module in Hackage, summarized here for illustration
module Data.Set(Set(), ...) where
  data Set a = ...

  empty :: Set a = ...
  insert :: Ord a => a -> Set a -> Set a = ...
  toAscList :: Set a -> [a] = ...

-- Top of the diamond, defining the type I for which we define orphan instances
module FunnyTop where

  -- simple wrapper around Int
  data I = I Int

  instance Eq I where ...
  instance Show I where ...

-- Left side of the diamond, defining Ord I one way
module FunnyLeft where
  import Data.Set
  import FunnyTop

  -- ⩽ ordering on wrapped Int; orphan instance
  instance Ord I where ...

  insL :: I -> Set I -> Set I
  insL = insert -- applies and "bakes in" this ⩽ instance

-- Right side of the diamond, defining Ord I the other way
module FunnyRight where
  import Data.Set
  import FunnyTop

  -- ⩾ ordering on wrapped Int; orphan instance
  instance Ord I where ...

  insR :: I -> Set I -> Set I
  insR = insert -- applies and "bakes in" this ⩾ instance

-- Bottom of the diamond
module FunnyBottom where
  import Data.Set
  import FunnyTop
  import FunnyLeft
  import FunnyRight

  funnySet :: Set I
  funnySet = insR (I 1) (insL (I 1) (insL (I 2) empty))
  --            ^           ^           ^-- uses ⩽
  --            ^           ^-- uses ⩽
  --            ^-- uses ⩾

  -- prints "[I 1, I 2, I 1]" instead of "[I 1, I 2]"
  main = print (toAscList funnySet)
```

Figure 4.1: Example of constructing an abstraction-unsafe Set.

The programed defined by the Funny modules exhibits a particular kind of bad behavior that we'd like to rule out—specifically, that the implementor of Data.Set would like to rule out in any potential client programs. Since `funnySet` was created with two opposite orderings, its internal representation as a binary tree (according to a single ordering) has been munged, as evidenced by the non-unique, non-ascending list that is printed out. In other words, this program breaks *abstraction safety*, *i.e.*, "the ability to locally establish representation invariants for [ADTs],"[9] like `Set`, which clients cannot possibly break when using the ADT operations, like `insert`.

Although never stated as a desired property for Haskell, abstraction safety should nonetheless hold for any type system that supports ADTs and modularity, arguably as much an imperative as traditional (syntactic) type safety, as has been argued by Dreyer (2014). But the lack of abstraction safety demonstrated here can be recast in terms specific to Haskell (or any language with type classes): the lax enforcement of global uniqueness of type class instances. Developers often rely on an assumption that any ground type class constraint $C\ \overline{typ}$ can be satisfied by at most one particular instance across the whole program. Since FunnyLeft and FunnyRight define distinct instances for `Ord I`, this program breaks the global uniqueness property. Unfortunately, however, GHC still allows it.

If such programs without global uniqueness of instances should be rejected, then on what grounds? Most literally, on the grounds that they violate the Haskell spec's *global uniqueness property*:[10]

Global Uniqueness Property. *A type may not be declared as an instance of a particular class more than once in the program.*

Clearly, the Funny program should *not* be a valid Haskell program, according to this definition, since it defines two instances for `Ord I`.

Despite what the spec says, this program is accepted by GHC, which therefore does not fully implement the Haskell spec when it comes to the intersection of modularity and type classes. (But we shall see shortly how GHC can be tuned to *partially* implement the spec.) The reason for this deviation is that no code in the program actually witnesses any type class *incoherence*:[11] in every module, whenever the typechecker requires a particular type class instance, like `Ord I`, there is (at most) a single choice of instance.

In contrast, if the program included the FunnyClient module below,

```
-- (X) rejected module
module FunnyClient where
  import Data.Set
  import FunnyTop
  import FunnyBottom
  funnySet2 = insert (I 5) funnySet -- ill-typed!
```

then GHC would reject it since the call to `insert` demands a single `Ord I` instance but the module knows of two—the one from FunnyLeft and the one from FunnyRight, which were both transitively imported into this module via FunnyBottom. Put simply, this module witnesses *incoherence* of type class instances. Typechecking it fails with an error reporting these two overlapping instances.

To the author of FunnyClient, it might seem strange to encounter an error that concerns the composition of FunnyLeft and FunnyRight. After all, these two modules were composed—*i.e.*, imported into the same module—upstream in FunnyBottom. (If the code that witnesses the two overlapping `Ord I` instances were defined not in a direct client of FunnyBottom but in a *client of a client*, then the reported error would be even more abstruse than before.) GHC therefore assigns blame not *eagerly*, to the (potentially upstream) module that composes two modules with overlapping instances, but *lazily*, to the (potentially downstream) module whose code witnesses the overlap.

9 Rossberg *et al.* (2014), "F-ing modules".
10 Marlow (2010), "Haskell 2010 Language Report," §4.3.2.
11 Jones (1993) and Peyton Jones *et al.* (1997a).

4.2.2 *Misdiagnosing the problem*

Although GHC accepts the erroneous Funny program by default, it does offer command-line options that, when combined in the right way, reject the program. On which grounds? Not because of adherence to the Haskell spec, but for the sake of compilation performance! By failing at the definition sites of so-called *orphan instances*,[12] GHC can avoid some superfluous file accesses when loading interfaces for depended-upon modules.[13] And it just so happens that by (optionally) rejecting orphan instances, GHC would consequently reject some programs that break the global uniqueness property—but not all of them, as we shall see shortly.

But why should ruling out orphan instances also ensure (in some cases) the global uniqueness of instances? If instance orphanhood is a mere implementation concern, what does it have to do with the Haskell spec? Surprisingly, it seems to be a coincidence—a coincidence that has led many in the Haskell community to misdiagnose the problem with programs that lack global uniqueness.

Roughly speaking, an *orphan instance* is a type class instance defined in a module that defines neither the type class nor any of the type constructors in the instance head. For example, the two instances defined in FunnyTop are *non*-orphans because they're defined in the same module as the type `I`, but the two instances defined in FunnyLeft and FunnyRight are indeed orphans. Non-orphanhood constitutes a further syntactic restriction on top of the Haskell spec's existing[14] syntactic restriction on instance declarations: each instance must have the form $\mathsf{C}\ (\mathsf{T}\ \alpha_1\ \cdots\ \alpha_n)$ (*i.e.*, a single type parameter, which must be a type constructor T applied to zero or more distinct type variables), and either C or T (or both) must be defined in the same module as this instance. With such an onerous restriction on instance declarations, the only way to break global uniqueness is to define two such instances in the exact same module—a localized problem that is already prevented by GHC. As a result, by rejecting all orphan instances and only accepting spec-approved instance declarations, GHC preserves the global uniqueness property.

Rejecting orphan instances is not the default mode of compilation in GHC, but avoiding them is quite conventional in practice. Indeed, the real-world analysis presented in §4.3 reveals that only 3% of all instances defined in common Haskell pckages are orphans.

4.2.3 *Failure of the diagnosis*

Hitching the assurance of global uniqueness to non-orphanhood might seem feasible for instances allowed by the Haskell spec, but in real-world Haskell programming this strategy does not suffice: some programs lack orphans yet break global uniqueness, and some programs contain orphans yet preserve global uniqueness.

First, a bad program without orphans. Figure 4.2 presents a program that, like the Funny program, defines conflicting instances in the left and right sides which are then imported together in the bottom, breaking global uniqueness of instances. The type class `Coercible`, defined at the top, is modeled after the somewhat magic type class for GHC described in (Breitner *et al.*, 2014) and more recently implemented as part of the Haskell standard library package, `base`.[15] The two conflicting instances, in NonorphanLeft and in NonorphanRight, are *not* orphans since each mentions a locally defined type, `BL` and `BR`, respectively. Taken together, the program defines two applicable instances for the class constraint `Coercible (BL Int) (BR Int)` but is accepted by GHC nonetheless.

12 The documentation of a command-line option to generate warnings (and thus, potentially, errors) at orphan instance declarations dates back to GHC 6.4, released in March 2005: `https://downloads.haskell.org/~ghc/6.4/docs/html/users_guide/options-sanity.html`

13 The documentation of orphans as a compiler performance concern dates back to GHC 5.00, released in April 2001: `https://downloads.haskell.org/~ghc/5.00/docs/set/separate-compilation.html`

14 The restrictive syntax of class and instance declarations laid out in the 2010 Haskell spec (Marlow, 2010) remains essentially unchanged from the original "Haskell 98" report from 2002.

15 `http://hackage.haskell.org/package/base-4.11.1.0/docs/Data-Coerce.html`

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
module NonorphanTop where
  -- a is coercible to b
  class Coercible a b where
    coerce :: a -> b

  instance Coercible Int Int where
    coerce = id

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module NonorphanLeft where
  import NonorphanTop

  -- a box type
  data BL a = BL a

  -- lift coercibility of a type *out of* its box type
  instance Coercible a b => Coercible (BL a) b where
    coerce :: BL a -> b
    coerce (BL xa) = coerce xa

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module NonorphanRight where
  import NonorphanTop

  -- another box type
  data BR b = BR b

  -- lift coercibility of a type *into* its box type
  instance Coercible a b => Coercible a (BR b) where
    coerce :: a -> BR b
    coerce xa = BR (coerce xa)

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module NonorphanBottom(bl, n, BL, BR) where
  import NonorphanTop
  import NonorphanLeft
  import NonorphanRight

  bl :: BL Int
  bl = BL 5

  n :: Int
  n = coerce bl

  -- this would be ill-typed because there are two possible
  -- instances for Coercible (BL Int) (BR Int):
  -- br :: BR Int
  -- br = coerce bl
```

Figure 4.2: Example of an orphan-free program that breaks global uniqueness of instances. Global uniqueness is broken because in NonorphanBottom (and its clients) there are two applicable instances for `Coercible (BL Int) (BR Int)`. Note that `FlexibleInstances` is required since the instance definitions each contain a naked type variable.

```
module FooData where
  data FooData = ...

module Fooable where
  import FooData
  class Fooable a where
    toFoo :: a -> FooData
```

```
name: foolib
build-depends:
  base,
  containers,
  transformers
exposed-modules:
  FooData, Fooable
```

```
module Fancy where
  data FancyT = ...
```

```
name: fancylib
build-depends:
  base, text
exposed-modules:
  Fancy
```

```
module FooFancy where
  import FooData
  import Fooable
  import Fancy

  instance Fooable FancyT where
    toFoo fancy = ...
```

```
name: foofancy
build-depends:
  base,
  foolib,
  fancylib
exposed-modules:
  FooFancy
```

Figure 4.3: Example of a program, consisting of three packages, that has an orphan instance but maintains global uniqueness of instances. To write this instance as a non-orphan, one of the first two packages would need a superfluous dependency on the other.

GHC's acceptance of the Nonorphan program requires a couple common, real-world language extensions, as the program's type class and instances do not adhere to the stringent syntax of the Haskell spec. Because `Coercible` has two type parameters, it requires `MultiParamTypeClasses`; and because the conflicting instance declarations have naked type variables as class parameters (*i.e.*, `a` in the left instance and `b` in the right instance), they require `FlexibleInstances`.

With these ubiquitous extensions to the Haskell spec's aging type classes, non-orphanhood no longer ensures global uniqueness of instances.

Second, a good program with orphans. Figure 4.3 describes a program split across three packages: `foolib` defines a class `Fooable` for serialization into its `FooData` type; `fancylib` defines some fancy new type `FancyT`; and `foofancy` merely defines the instance `Fooable FancyT`, an orphan instance, for serializing fancy types into `FooData`. Though seemingly contrived, this configuration arises often in practice.

Each of the first two packages adds its own utility to the Haskell ecosystem, but neither actually depends on the other—perhaps they were authored in total isolation. However, a third party (such as `foofancy`'s author) might like to combine both of them into greater utility by defining the missing type class instance, but this third party would *necessarily* define the instance as an orphan. To avoid the orphan, exactly one of the first two packages must depend on the other and then define the instance along with its own class or type. Not only does this introduce a synchronization problem between the two authors; it also introduces a superfluous package dependency—compilation bloat that package authors generally wish to avoid.

4.2.4 *New diagnosis: deficient type system*

The Haskell spec describes a key property that must hold for valid programs: the so-called global uniqueness of type class instances. And many Haskell programs are written with this property in mind, to the extent that the correctness of their downstream clients depends on it, as the Funny example shows.

However, the spec's terse description of global uniqueness doesn't sufficiently instruct Haskell implementors (or semanticists) how to faithfully implement (or formalize) it.[16] It's a non-local property in the sense that, when checked on a per-module basis, it's not necessarily preserved under module composition (*i.e.*, the import of modules into the same client module): a program might define multiple conflicting type class instances without any module witnessing their incoherence. Moreover, it's a property not of a module's own apparent instances but of its "in-scope" instances, some of which trickle down from upstream dependencies that the composing module knows nothing about.

The real problem with the spec is that it defines a non-local property with no explanation of how to decompose it into modular properties. In other words, it lacks a type system.

A type system for Haskell modules should encode and enforce modular properties that enable the determination of non-local properties like global uniqueness. With a type system, client modules could "look at the types" to determine whether the import of two modules preserves such properties.[17] And as with a conventional typechecker, said types would be stored, reused, and perhaps substituted for further analysis at other usage sites.

In contrast to a type system, with the (insufficient) non-orphanhood restriction many Haskell programmers rely on, client modules have no language for determining whether particular modules maintain global uniqueness after composition. Somewhat paternalistically, programmers are simply unable to define instances that might potentially, but not necessarily, break global uniqueness.

4.3 WORLD SEMANTICS FOR TYPE CLASSES

As described in the previous section, global uniqueness of type class instances is a key property for Haskell. However, defined at the granularity of instance definitions ranging over the whole program, rather than at the granularity of modules, global uniqueness doesn't lend itself naturally to modular enforcement and composition. Consequently, GHC attempts to enforce the property by checking only *instance definition* sites[18] and not *module import* sites, an attempt which the previous section revealed to be inadequate.

As an example of how the granularity of the global uniqueness property matters for modularity, consider the following question: Do the already-typechecked modules A and B maintain global uniqueness when composed? With the property defined at the granularity of instance definitions ranging over the whole program, one should look at all instance definitions in A, B, and their upstream dependencies and check for conflicts. With a more modular formulation of global uniqueness, one might answer the question by analyzing only the already-synthesized static knowledge of the two modules A and B—essentially, their types.

In order to modularize global uniqueness, we need a semantical component that describes the "static knowledge" that each module has about type class instances: *worlds*. Roughly speaking, a world maps a type class constraint like `Eq Int` to the module that defines an instance for that constraint, while the world *inhabited* by a module determines the in-scope instances of that module.

In §4.4 worlds will get fleshed out in technical detail. For now, since worlds are a fairly straightforward idea, we'll jump right into some examples.

4.3.1 *World tour*

In this first example, modules successively depend on each other and add additional instances, none of which conflict with each other.

16 The semantics of recursive modules (Marlow, 2010, Ch. 5) is similarly left open for others to flesh out (Diatchki *et al.*, 2002; Kilpatrick *et al.*, 2014).

17 Type systems for ML modules similarly evolved out of a need to express, in the types of modules, relationships between abstract type components. Consider, for example, the introduction of "manifest types" (Leroy, 1994) and "translucent signatures" (Harper and Lillibridge, 1994).

18 When typechecking a module, GHC checks that each new instance definition does not conflict with any instances defined upstream or with any other instances in the same module.

```
module A where
  class Foo a where ...

module B where
  import A
  data TB = ...
  instance Foo TB where ...

module C where
  import A
  import B
  data TC = ...
  instance Foo TC where ...
```

Each module's world is determined by (1) the worlds of imported modules and (2) the module's own instance definitions; both are combined according to a *merge* operation on worlds, $\oplus$. In the absence of conflict, that merge operation acts essentially like a set union, but because merging fails in the case of two worlds with conflicting instances, the operation is partial. The world inhabited by some module is defined as $\omega = \omega_1 \oplus \ldots \oplus \omega_n \oplus \omega_M$, where ω_i is the world inhabited by the ith imported module, and ω_M is the set of additional instances defined in this module. For example, the worlds in the trivial example above are:

$$\omega_A = \emptyset$$
$$\omega_B = \omega_A \oplus \{\, \texttt{Foo TB} \mapsto \mathsf{B} \,\}$$
$$\omega_C = \omega_A \oplus \omega_B \oplus \{\, \texttt{Foo TC} \mapsto \mathsf{C} \,\}$$

As a consequence of the definition of a module's world, every downstream module knows *at least as much* as its upstream modules. In particular, the current module's world ω *extends* each of the imported modules' worlds ω_i, where world extension, written $\omega' \sqsupseteq \omega$, means that ω' is equal to the merging of ω with some additional knowledge. For example, $\omega_C \sqsupseteq \omega_B \sqsupseteq \omega_A$. In §4.4 we will see how world extension factors into *matching* in Backpack extended with type classes.

Returning to the Funny example from Figure 4.1, we see what happens when worlds collide, *i.e.*, when two worlds cannot be merged because they have *mutually inconsistent* knowledge.

Supposing that Data.Set defines no instances, what are the worlds inhabited by the five modules in the program? For all but FunnyBottom, it's straightforward:

$$\omega_T = \{\, \texttt{Eq I} \mapsto \mathsf{FunnyTop},\ \texttt{Show I} \mapsto \mathsf{FunnyTop} \,\}$$
$$\omega_L = \omega_T \oplus \{\, \texttt{Ord I} \mapsto \underline{\mathsf{FunnyLeft}} \,\}$$
$$\omega_R = \omega_T \oplus \{\, \texttt{Ord I} \mapsto \underline{\mathsf{FunnyRight}} \,\}$$

FunnyBottom, however, is different. Because it imports the left and right modules, its world is defined, in part, as the merging of ω_L and ω_R. And, intuitively, this merging should be undefined since the left and right instances conflict. Indeed, each of the two worlds ω_L and ω_R maps the class constraint `Ord I` to a different module defining the instance; $\omega_L \oplus \omega_R$ is thus obviously undefined. Moreover, because the FunnyBottom module composes FunnyLeft and FunnyRight, and thus $\omega_B = \ldots \oplus \omega_L \oplus \omega_R$, we say that FunnyBottom inhabits an "inconsistent" world $\omega_B = \bot$.[19]

19 We abuse notation by saying that $\omega_B = \ldots \oplus \omega_L \oplus \omega_R$ is both undefined and equal to $\bot$, as worlds can be understood also as a *total* commutative monoid with $\bot$.

4.3.2 *Redefining type class semantics with worlds*

Not simply a new layer sitting on top of Haskell, the worlds semantics described above is intended to subsume the existing semantics of Haskell type classes—at least at the level of the *module* system, *i.e.*, import and export semantics. Although worlds determine the in-scope instances available to a module, they say nothing about instance resolution, constraint solving, or type inference—type class semantics at the level of the *core* language of types and values.

Haskell's module system treats instances as core-level program entities, like types and values, that are imported and exported between modules. Unlike types and values, however, instances are imported and exported somewhat magically: *"All instances in scope within a module are always exported and any import brings all instances in from the imported module. Thus, an instance declaration is in scope if and only if a chain of import declarations leads to the module containing the instance declaration."*[20] Compared to the spec's definition, the worlds semantics of type classes offers an alternative, cleaner characterization: an instance is in scope within a module if and only if that module's world knows about it.

Beyond the module-level semantics of type classes, the world semantics enables a clear re-formulation of global uniqueness:

World Consistency Property. *Every module in the program must inhabit a consistent world.*

Because consistent worlds characterize exactly those modules which see no conflicting instance definitions, the world consistency property subsumes the global uniqueness property for a complete program. Even better, it's a more modular property. As a quantification over modules, world consistency decomposes cleanly into a property that may be verified at the granularity of a module—with a world ω acting as evidence—and then stitched back together for a whole program.

Unfortunately, in the more modular system of Backpack, where modules can depend on signatures which can then be implemented by other modules, world consistency *does not suffice to guarantee global uniqueness*, at least not at the level of *modules*. We'll return to this point in §4.4, but for now we'll continue with the idea of world semantics *for Haskell*.

Both global uniqueness and world consistency have been stated as *ad hoc* properties that should be true of any valid Haskell program. Instead, why not formally encode such a property as part of the type system of modules? Then the truth of the property would be established by the well-typedness of a program's modules. Our more modular formulation of the property, world consistency, quite naturally facilitates such an encoding since worlds are determined during module typechecking and since module well-typedness is established within those worlds. In §4.4 I will detail the encoding of worlds into module types—though in the richer system of Backpack.

Much later in this dissertation, I will show that the Internal Language of Backpack—more or less a formalization of plain Haskell—satisfies world consistency as a consequence of a program being well-typed. Indeed, the idea of embedding world consistency *into the type system* is evidenced in that system; see §9.3.6.

* * *

Embedding type classes into the module type system seems like a win-win situation: make formal and "typey" an otherwise informally specified and mostly unchecked requirement of Haskell. What can go wrong? As it turns out, when world semantics enter the type system for Haskell's modules, that type system loses conventional metatheoretic, structural properties. In rough terms that's the flip side of embedding worlds—which in a sense act as barriers to modular composition—into the type system.

Take *Weakening*, for example. In most type systems with a typing judgment of the form $\Gamma \vdash e : \tau$, Weakening states that if such a typing derivation can be formed, then so can $\Gamma' \vdash e : \tau$, where Γ' has more, or richer, assumptions than Γ. This property can then be proven in

20 Peyton Jones (2003), "Haskell 98 Language and Libraries: the Revised Report," §5.4.

terms of the particular definitions of the judgment. With type classes and world semantics, however, imported modules in that richer Γ' possibly define new type class instances that cause conflicts downstream in e—to the point that e is no longer well-typed at all, let alone with type τ!

In my formalization of Backpack's Internal Language (Chapter 9), I have stated and proved several such metatheoretic properties that are conventional for typing derivations in type systems, *e.g.*, *Weakening* and *Cut*. In all such cases the properties stipulate an onerous side condition I call *world preservation*: the concerning module expression must inhabit a consistent world even after the structural modification to the typing derivation. Indeed, the proofs of these properties are stuck without this additional assumption. Looking at these painful side conditions to otherwise conventional type systems metatheory sheds new light on the oft-heard folklore assessment that type classes are *antimodular*.

4.3.3 *World consistency in the wild*

I have argued that the world semantics is the right semantics for (the module level's interaction with) type classes in Haskell. And I have argued that world consistency, the modularization of global uniqueness, is a crucial property that should hold of every Haskell program. But do Haskell programs in the wild actually inhabit consistent worlds?

To answer this question I have developed a *post hoc*, prototype implementation[21] of world semantics and tested it on over 100 common Haskell libraries (and almost 200 more when including their dependencies). With my protoype I have found that, *yes*, the vast majority of common Haskell libraries inhabit consistent worlds.

PROTOTYPE The prototype synthesizes the world inhabited by every module in a package by processing those modules' "binary interface files" (*i.e.*, the `.hi` files) which result from locally compiling, with GHC 7.8.4, their source files. Each binary interface file lists the locally defined type class instances and the names of imported modules.[22] Therefore, by processing these interface files, one can synthesize worlds from compiled modules. The full process for analyzing worlds in some package P goes as follows:

1. Install package P in a local (sandbox) repository on our test machine. P must be a "Cabalized" package that defines a library, not merely an executable. Installation involves compiling every (exposed and hidden) module in the package—with default Cabal settings—and locally storing their binary interface files.

2. For each module M in P, use the GHC API to read in M's binary interface file and parse it for its locally defined type class instances and imported modules.

3. Recursively[23] synthesize a world for each of M's imports, and then merge these worlds with M's local instances to form M's world.

PACKAGE SELECTION I selected packages from two sources that together constitute some notion of "common": first, the 53 packages included in the Haskell Platform 2014 release, and second, the 80 most-downloaded packages from Hackage, defined as all those with at least 1000 downloads at the time of my testing.[24] From both sources I only considered packages that define libraries and not just executables (*e.g.*, `alex` and `cabal-install`), as binary interface files of modules from Cabal executables are not installed locally. In total, 118 packages

21 Code and data available at `https://gitlab.mpi-sws.org/backpack/class-struggle/tree/v3`.

22 Technically, each interface file lists not the directly imported modules but some subset of the *transitively* imported modules that contains at least the modules from which it imports instances.

23 In the case of recursive modules, GHC requires that the import cycle be broken with a "boot file." Because my prototype treats a boot file for M as a separate module from M itself, the prototype also visits and synthesize worlds for boot files (which themselves have binary interface files, `.hi-boot` files). Saving the boot files' interfaces when installing a package with recursive modules required a straightforward modification to Cabal; see instructions at my fork of Cabal 1.22: `https://gitlab.mpi-sws.org/backpack/cabal-hi-boot/tree/class-struggle`

24 Package lists with sources: `https://gitlab.mpi-sws.org/backpack/class-struggle/blob/v3/data/pkgs`

Which Packages	# Consis.	# Inconsis.	# Total	% Consis.
Selected from *Platform*	53	0	53	100%
Selected from *Popular*	59	21	80	74%
All Selected	97	21	118	82%
All Analyzed	263	31	294	89%

Table 4.1: Calculation of world consistency in the wild.

Analyzed Package	Selected?	A	B
`ReadArgs-1.2.2`		1	2
`aeson-0.8.0.2`	YES	3	20
`egison-3.5.6`	YES	1	0
`ghc-7.8.4`		2	3
`haskeline-0.7.1.2`		4	1
`pandoc-citeproc-0.6`	YES	2	1
`persistent-2.1.2`	YES	2	7
`persistent-template-2.1.1`	YES	1	4
`shelly-1.6.1.2`	YES	3	0
`yesod-1.4.1.4`	YES	1	0
`yesod-persistent-1.4.0.2`		1	3

Table 4.2: Sources of world *inconsistency* in the wild. Column A denotes the number of modules in the package that create inconsistency, *i.e.*, where the locally defined instances, the imported instances, or the merging of both such groups lead to overlap. Column B denotes the number of *additional* packages analyzed that inherit inconsistent worlds from each such package.

were selected from the two sources. Since I needed to install (and process) not just those packages but also all their dependencies, a total of 294 packages were analyzed.

Because these 118 packages were selected only by their names, like `containers`, I also needed to select a particular *version* of each, like `0.5.5.1`, in order to uniquely identify some code to compile. For this selection I used the Stackage snapshot of "safe" package configurations.[25]

RESULTS The results of the calculation are presented in Table 4.1. Although 31 packages were found to define modules with inconsistent worlds, only 11 of all 294 analyzed packages (4%) define any modules that "create" inconsistency, *i.e.*, modules that either define instances that conflict with imported worlds or whose imported worlds are mutually inconsistent. 20 additional packages "inherit" their inconsistency from these 11. The 11 packages creating inconsistency are listed in Table 4.2.[26] Additionally, Table 4.3 presents an enumeration of the orphan instances defined within the same packages. Interestingly, the percentage of orphans among all instances defined in each category of packages is roughly 3%.

25 Stackage LTS 2.1: `http://www.stackage.org/snapshot/lts-2.1`
26 See the project `README` for more information about how to analyze and interpret these results: `https://gitlab.mpi-sws.org/backpack/class-struggle/tree/v3/`

Which Packages	# Orphans	# Total	% Orphans
Selected from *Platform*	171	5,136	3%
Selected from *Popular*	345	10,282	3%
All Selected	426	14,083	3%
All Analyzed	651	20,332	3%

Table 4.3: Enumeration of orphan instances in the wild.

The worst offender in the sources of inconsistency is the `aeson-0.8.0.2` package, a common library for dealing with JSON in Haskell.[27] But instead, given that it's the implementation of the compiler itself, `ghc-7.8.4` is what I'll focus on in more detail.[28]

This package, the GHC API, is one of the 11 packages causing inconsistency. Within this package one module, `CmmExpr`, defines two sets of conflicting instances (three in one set, four in the other), and in a downstream module, `CmmNode`, one of those instances conflicts with a locally defined one, making 8 total conflicting instances.[29] All instance conflicts in this package stem from two type classes for which there exist overly generic instances with naked type variables in their parameters:

- `CmmExpr.DefinerOfRegs r r`
- `CmmExpr.UserOfRegs r r`

Each one of these two instances clashes with every other instance of the form C `a` $(\cdots$ `b` $\cdots)$, like `CmmExpr.DefinerOfRegs r [a]`, which is unifiable with the overly generic instance above.

The inconsistency from one or both of these modules trickles down into the worlds of 260 other modules. That's over half of the 448 total modules defined in the `ghc` package. All told, the package's combined world contains 3,921 instances, of which 1,472 are defined within the package itself. And of those 1,472 instances only 8 are conflicting, thereby producing inconsistency. Finally, to further hammer home the point that orphans aren't the problem, none of these 8 instances is one of the 56 orphan instances defined in the package. Full results can be found in the data files in the accompanying repository.[30]

* * *

To summarize, my prototype demonstrates that, among hundreds of common Haskell packages, modules that don't inhabit consistent worlds are only a small minority. My analysis suggests that the Haskell ecosystem could remain largely unchanged if it were typechecked against the world semantics of type classes.

Moreover, since world consistency was designed as a modularization of the Haskell spec's global uniqueness property, the analysis shows which of these common packages adhere to the spec (with respect to type class instances): exactly those packages that inhabit consistent worlds. This is a contribution in itself, regardless of world semantics.

27 Package description in Hackage: http://hackage.haskell.org/package/aeson-0.8.0.2
28 Source code: https://github.com/ghc/ghc/tree/ghc-7.8.4-release
29 The `CmmExpr` and `CmmNode` modules define data and functionality for GHC's internal C-- representation of programs. Both modules use the extensions `FlexibleContexts` and `UndecidableInstances`, thereby allowing GHC to accept them as well-typed despite defining conflicting instances. See the GHC documentation on the latter language extension: https://downloads.haskell.org/ghc/7.8.4/docs/html/users_guide/type-class-extensions.html#undecidable-instances
30 A slightly modified output from the prototype that demonstrates the inconsistency and worlds in the `ghc-7.8.4` package can be found at the following URL, with more info at the project `README`: https://gitlab.mpi-sws.org/backpack/class-struggle/blob/v3/data/worlds/islands-ghc.txt#L736-797

package a-sig **where**

```
       ┌  data Set a
       │ empty :: Set a
       │ insert :: Ord a => a -> Set a -> Set a
A ::   │ class Foldable t where
       │   foldr :: (a -> b -> b) -> b -> t a -> b
       └ instance Foldable Set
```

package client **where**

include a-sig

```
       ┌  import A
       │ export (main, Foldable)
       │ data T = T Int
B =    │ xs = insert (T 5) empty
       │ main = print (foldr ... xs)
       │ instance Eq T where ...
       └ instance Ord T where ...
```

package a-impl **where**

```
       ┌  data Set a = ...
       │ empty = ...
       │ insert = ...
A =    │ class Foldable t where
       │   foldr :: (a -> b -> b) -> b -> t a -> b
       │ instance Foldable Set where ...
       └ instance Eq a => Eq (Set a) where ...
```

package main **where**

include client

include a-impl

Figure 4.4: Revised example of modular abstraction, now with type classes.

4.4 RETROFITTING BACKPACK WITH TYPE CLASSES

Now that the reader has a grasp of the world semantics for type class instances in Haskell, we return our attention to Backpack. Figure 4.4 contains a few Backpack packages whose modules and signatures contain type classes and instances. We will now walk through how type classes, instances, and worlds manifest in this example, along the way discussing how Backpack's semantics should be updated—with worlds—to capture that change.

4.4.1 *Type classes as core entities*

Most immediately, we see the need for representing type classes (like `Foldable`) and instances (like that of `Eq I`) as definable core entities like values and types. However, only classes, not instances, should be imported and exported like the other core entities (*e.g.*, `Foldable` in B); instances should instead obey the world semantics discussed in §4.3. We therefore need to augment Backpack's core names:

Class Names	C	$\in$	*ClassNames*
Defined Entities	*dent*	$::=$	$\ldots \mid \mathsf{C}(\overline{x})$

Instance names, however, don't exist in the surface syntax. They are implicitly generated, derived from the types and classes in the instance definitions.

Module types need to capture all the typed information about core entities (defined in those modules). So they'll also need to know the types of everything related to class definitions and instance definitions. For example, the type of A contains the following specification of the type class definition for `Foldable`:

```
class Foldable (t :: * -> *) {} where
    foldr :: (a -> b -> b) -> b -> t a -> b
```

The empty curly braces represent the fact that this class has no superclass constraints. Likewise, the specification of the instance definition for the `Eq` instance in `a-impl`'s A module is:

$$\texttt{instance}\ (\texttt{a} :: *)\ \{[\nu_0]\texttt{Eq a}\}\ [\nu_0]\texttt{Eq}\ ([\nu_A]\texttt{Set a})$$

This specification contains all the information to uniquely identify each name and type in the instance definition: the originating module identity, in square brackets, along with that entity's syntactic name, together forming a *physical name* (*phnm*). Note that the `Set` type is defined in this module itself, which has identity ν_A.

To account for these new kinds of entity definitions, the semantic object that describes the typed specifications of entities needs to be updated. As a prerequisite, we also need a semantic representation of class constraints like $[\nu_0]\texttt{Eq}\ ([\nu_A]\texttt{Set a})$, just as Backpack needed a semantic representation of types (§3.1):

Semantic Class Constraints $\quad cls ::= [\nu]\mathsf{C}\ \overline{typ}$

Defined Entity Specs

$$\begin{aligned} dspc ::=\ & \ldots \\ \mid\ & \texttt{class}\ \mathsf{C}\ kenv\ \{\overline{cls}\}\ \texttt{where}\ \overline{\texttt{x} :: typ} \\ \mid\ & \texttt{instance}\ kenv\ \{\overline{cls}\}\ cls \end{aligned}$$

Like type constructors, classes are uniquely identified with physical names as well, like $[\nu]\mathsf{C}$. In the above example of the specification of A's `Eq` instance, there are multiple mentions of the physical name of that class, $[\nu_0]$`Eq`.[31]

The multiple occurrences of *typ* in the semantic object for class constraints, *cls*, highlights a departure from the Haskell spec: type classes in Backpack are inherently multi-parameter. That's because the complexity of multi-parameter type classes is entirely orthogonal to *the module level*; nothing in Backpack's definitions has any need for there to be only a single type parameter on classes.[32]

4.4.2 *Worlds in module typing*

The instances known to each module are determined not via the import and export resolution of entity names, but via the world semantics (§4.3). For reference, the definition of worlds in Backpack, as semantic objects, is provided in Figure 4.5. The full definition will be provided in §7.2 (Figure 7.5).

The mechanics of worlds in Backpack essentially match what we saw in the previous section: worlds form a partial commutative monoid (PCM) on finite maps from class constraints to module identities.

31 We assume that the class `Eq` originates in some implicit, builtin module with identity ν_0, as does the type `Int`. We shall also assume that the same module defines instances like `Eq Int`.

32 Indeed, the problem with orphanhood presented in §4.2 showed that such *syntactic* restrictions (*i.e.*, prohibiting orphan instances) were insufficient as compared to *semantic* restrictions (*i.e.*, requiring consistent worlds). The complexity for multi-parameter type classes exists instead at *the core level*; as such, the complexity is shoved under the rug of the (assumed) core type unification ($\mathsf{unify}(typ; typ)$) used as part of instance head avoidance ($head\ \#\ head$) defined in Figure 4.5.

$\boxed{\omega}\ \boxed{\mathit{fact}}\ \boxed{\mathit{head}}$

$$\begin{array}{llcl} \text{Instance Heads} & \mathit{head} & ::= & \mathit{kenv}.\mathit{cls} \\ \text{Instance Facts} & \mathit{fact} & ::= & \mathit{head} \mapsto \nu \\ \text{Worlds} & \omega & ::= & \{\!|\, \overline{\mathit{fact}} \,|\!\}^{\dagger} \end{array}$$

$\boxed{\mathit{head} \mathbin{\#} \mathit{head}}$

$$(\mathit{kenv}_1.([\nu_1]\mathsf{C}_1\ \overline{\mathit{typ}_1})) \mathbin{\#} (\mathit{kenv}_2.([\nu_2]\mathsf{C}_2\ \overline{\mathit{typ}_2})) \overset{\mathsf{def}}{\Leftrightarrow} \begin{cases} \text{true} & \nu_1 \neq \nu_2 \vee \mathsf{C}_1 \neq \mathsf{C}_2 \\ \forall i : \neg\mathsf{unify}(\mathit{typ}_{1,i}; \mathit{typ}_{2,i}) & \text{otherwise} \end{cases}$$

$\boxed{\mathsf{consistent}(\omega)}$

$$\mathsf{consistent}(\omega) \overset{\mathsf{def}}{\Leftrightarrow} \forall \mathit{fact}_1, \mathit{fact}_2 \in \omega : \mathsf{head}(\mathit{fact}_1) \mathbin{\#} \mathsf{head}(\mathit{fact}_2) \vee \mathit{fact}_1 = \mathit{fact}_2$$

$\boxed{\omega \oplus \omega}$

$$\omega_1 \oplus \omega_2 \overset{\mathsf{def}}{=} \omega_1 \cup \omega_2, \quad \text{if}$$
$$\forall \begin{pmatrix} \mathit{fact}_1 \in \omega_1, \\ \mathit{fact}_2 \in \omega_2 \end{pmatrix} : \mathsf{head}(\mathit{fact}_1) \mathbin{\#} \mathsf{head}(\mathit{fact}_2) \vee \mathit{fact}_1 = \mathit{fact}_2$$

$\boxed{\omega \sqsupseteq \omega}$

$$\omega' \sqsupseteq \omega \overset{\mathsf{def}}{\Leftrightarrow} \exists \omega_F : \omega' = \omega \oplus \omega_F$$

$\boxed{\mathsf{apply}(\theta; \omega)}$

$$\mathsf{apply}(\theta; \{\!|\, \overline{\mathit{fact}} \,|\!\}) \overset{\mathsf{def}}{=} \{\!|\, \overline{\theta \mathit{fact}} \,|\!\}, \quad \text{if}$$
$$\forall \mathit{fact}_1, \mathit{fact}_2 \in \overline{\mathit{fact}} : \theta\mathsf{head}(\mathit{fact}_1) \mathbin{\#} \theta\mathsf{head}(\mathit{fact}_2) \vee \theta\mathit{fact}_1 = \theta\mathit{fact}_2$$

Figure 4.5: Definition of world semantic objects in Backpack. † refers to the fact that, although worlds have the presented syntax, they additionally carry the invariant $\mathsf{consistent}(-)$.

As a first example, consider the world of the A module defined in a-impl. Its world is determined exactly as in the previous section: merge the worlds of its imports (empty, in this case) with the world consisting only of locally defined instances. That world is:

$$\omega'_A = \left\{ \begin{array}{lll} [\nu_A]\texttt{Foldable}\ [\nu_A]\texttt{Set} & \mapsto & \nu_A, \\ (\mathsf{a} :: *).\ [\nu_0]\texttt{Eq}\ \ ([\nu_A]\texttt{Set a}) & \mapsto & \nu_A \end{array} \right\}$$

There are three key observations about this world. First, rather than syntactic instance names, we simply map each class constraint to the module identity that locally defines the instance for that constraint. This suffices to identify a particular instance with a particular class constraint because each module can only define a single instance producing that class constraint; otherwise, such a module would define two conflicting instances.

Second, all class and type names occurring in the class constraints are uniquely identified with their defining module identities, like ν_A for `Foldable`, this module's identity as determined earlier in the section.

Third, the `Eq` instance is polymorphic and thus requires a kind environment (*kenv*) to make sense of the type variable a. The latter addition—polymorphic class constraints in worlds—is not unique to Backpack; Haskell would require the same mechanism with a worlds semantics, even though the discussion in §4.3 didn't necessitate it. Not just syntactic noise, the possible polymorphism in instance definitions directly concerns the *true* definition of the partial merge operation on worlds (Figure 4.5): two distinct instances conflict—*i.e.*, overlap—if their class constraints are unifiable in core-level type variables a (not in module identity variables), hence the unify operation in the definition of merge.

Two instances defined in *the same module* would of course conflict even when their class constraints are unifiable. That's a classic case of overlapping instances. For example, if the A module in a-impl also defined a specialized instance for `Eq (Set Int)`, that instance would impose the following additional fact in the world ω'_A:

$$().\ [\nu_0]\texttt{Eq}\ \ ([\nu_A]\texttt{Set}\ [\nu_I]\texttt{Int}) \ \mapsto\ \nu_A$$

That semantic class constraint absolutely unifies (in core type variables) with that of the other `Eq` instance in the world, using the core type substitution $\mathsf{a} := [\nu_I]\texttt{Int}$. In other words, the statement

$$\mathsf{unify}\Big(\ (\mathsf{a} :: *).\ [\nu_0]\texttt{Eq}\ \ ([\nu_A]\texttt{Set a})\ \ ;\ \ ().\ [\nu_0]\texttt{Eq}\ \ ([\nu_A]\texttt{Set}\ [\nu_I]\texttt{Int})\ \Big)$$

holds, but we don't actually want to treat these two world *fact*s as compatible. As a result, the world that contains both instance *fact*s would simply not be a valid semantic object. (Again, alternatively, one could say such a world exists, but it's the inconsistent world, $\bot$.)

This partiality of merge that requires the facts in the two worlds to be consistent with each other—*i.e.*, non-overlapping—is not just part of the world merging definition. It's actually a key invariant of worlds themselves, expressed as $\mathsf{consistent}(\omega)$ in Figure 4.5. Worlds as semantic objects maintain this invariant in Backpack: in the formalization's metatheory, everywhere a world semantic object is synthesized or denoted, the consistency invariant must be proven. As far as the type system goes, this manifests in the consistency condition of definedness for partial operations denoting worlds, like $(\omega_1 \oplus \omega_2)$ and $\mathsf{apply}(\theta; \omega)$.

Worlds of modules are determined just as presented earlier in this chapter for Haskell, but where are they actually located in *Backpack's* semantics? The approach of inserting them into the module level is meant literally—I augment module contexts and judgments with world objects, as "@ ω" (see §7.2 for full definitions):

$$\begin{array}{llll} \text{Physical Shape Ctxts} & \hat{\Phi} & ::= & \overline{\nu{:}\hat{\tau}^m @ \hat{\omega}} \\ \text{Physical Type Ctxts} & \Phi & ::= & \overline{\nu{:}\tau^m @ \omega} \end{array}$$

$$\begin{array}{ll}\text{Module Shaping} & \widehat{\Gamma};\, \nu_0 \Vdash M \Rightarrow \hat{\tau} @ \widehat{\omega} \\ \text{Module Typing} & \Gamma;\, \nu_0 \vdash M : \tau @ \omega\end{array}$$

As an example, the shape of the A implementation in a-impl is

$$\hat{\tau}'_A = \left\langle \begin{array}{l}\texttt{Set(..)},\ldots, \\ \texttt{Foldable(foldr)}\end{array} ; \begin{array}{l}[\nu_A]\texttt{Set(..)},\ldots, \\ [\nu_A]\texttt{Foldable(foldr)}\end{array} \right\rangle \;@\; \omega'_A.$$

Note that this shape does not mention, among its *defined entity names* in the first component, either of the two instances defined in this module. In a hypothetical Haskell with syntactic instance names, they would occur here, but for now, the module shape has no use for them. During the *typing* pass, however, the module *type* will indeed mention the specs for the two instances:

$$\begin{array}{llll}\texttt{instance} & (a :: *) & \{[\nu_0]\texttt{Eq}\ a\} & [\nu_0]\texttt{Eq}\ ([\nu_A]\texttt{Set}\ a) \\ \texttt{instance} & & \{\} & [\nu_A]\texttt{Foldable}\ [\nu_A]\texttt{Set}\end{array}$$

4.4.3 *Worlds for signatures too*

The very first package already confronts us with a scenario we didn't see in the previous section—signatures, and they inhabit worlds just like modules do.

The world of a signature like A in a-sig is determined exactly as if it were a module: merge the worlds of its imports with the world consisting only of locally defined (rather, *declared*) instances. Thus the world of this signature only knows about the single instance for `Foldable Set`. What is the module identity that defines this instance?

In the original semantics section for Backpack (§3.2), we saw that signatures yield a module identity variable α, to be substituted by the identity of the implementation, and also n variables β_i, each to be substituted by the identity of the module that provides the i-th entity declared in the signature. Concretely, in a-sig, the identity of A is α_A and the identities of (the modules that will implement) its declared entities are β_S, β_e, β_i, β_F, and β_{FS}, all of them fresh and distinct. Padding the class, instance, and type names with their defining identities, we see that A's world is the singleton

$$\omega_A = \{[\beta_F]\texttt{Foldable}\ [\beta_S]\texttt{Set} \mapsto \beta_{FS}\}.$$

Although this world mentions module identity variables, it's still a perfectly valid world.

We have now demonstrated the worlds of both the implementation of A (ω'_A) and the hole of A (ω_A), as they exist in their respective packages. Notably, the former is not related to the latter because the class constraint and defining module identity in the hole are entirely distinct from those in the implementation: `Foldable`, `Set`, and the instance itself all come from module ν_A in the implementation but from modules β_F, β_S, and β_{FS} in the hole. However, this does not present a problem for the well-typedness of main, as it will see *unified* identities in the respective worlds.

4.4.4 *Worlds when linking*

During Backpack's shaping pass (§3.2), when an implementation is linked for a hole, the identities contained in the implementation are unified with their corresponding identities (as variables) in the hole. Then, during the typing pass, the module type of the implementation must be a subtype of the now-unified module type of the hole.

The same unification-followed-by-subtyping occurs with worlds as well; *i.e.*, the world of the implementation must *extend* the world of the hole. In our example, that means $\theta(\omega'_A) \sqsupseteq \theta(\omega_A)$, where θ is the unifying substitution that results from linking.

However, we must slightly adjust Backpack's linking semantics to account for worlds. In Backpack, the identities to unify between two module types are looked up according to the exported entities list of the two types: if the implementation exports $[\nu]\mathsf{x}$ and the hole exports $[\beta]\mathsf{x}$, then we know to unify ν and β. In Backpack extended with worlds, the identities to unify must be looked up not just in these export specs but also in the two modules' worlds.

For example, when unifying the shapes of the A implementation and hole, the identities for the exported `Foldable` and `Set` in the implementation (both ν_A) are unified with their counterparts in the hole (β_F and β_S)—just as expected from Backpack. However, type class instances demand an extra unification. After the previous unification, the implementation and the hole both know an instance for the constraint $[\nu_\mathsf{A}]$`Foldable` $[\nu_\mathsf{A}]$`Set`, but the two worlds map this constraint to distinct module identities: ν_A in ω'_A and β_FS in ω_A. Therefore we must unify *the defining identities* of these instances, ν_A and β_FS, that have equal class constraints. Without this additional unification on the instances' defining module identities, the linking substitution θ would not yield $\theta(\omega'_\mathsf{A}) \sqsupseteq \theta(\omega_\mathsf{A})$.

Note however that substitution on worlds is actually *partial*, as the consistency invariant must be maintained. (See the definition of $\mathsf{apply}(\theta;\omega)$ in Figure 4.5.) Why might substitution on a world produce an inconsistent world? The example in §4.5.3 shows exactly that.

In Backpack a world of an implementation must *extend* the world of a signature it matches. That's because a downstream module that imports the signature might rely on some fact exposed by that signature; after all, signatures have worlds constructed from imported worlds and locally declared instances just like modules do. Then in order to make sure that the downstream module continues to observe the same fact after linking, the world of the implementation must be checked to contain all the facts known to the world of the signature. In §4.5.1 an example highlights this further.

4.4.5 *Lifting world consistency to packages*

In §4.3.2 it was hinted that the World Consistency Property, stated as a modular reformulation of the Global Uniqueness Property in Haskell, would not suffice in the more modular context of Backpack. To illustrate why, let's reconsider the motivating Funny example from Figure 4.1. First, we rewrite it as a package in Backpack, and second, we replace the FunnyRight module with a signature that hides the problematic instance. The resulting package, funny-partial, is defined in Figure 4.6.

Within funny-partial, there is only one instance for `Ord I` and therefore no concerns about consistency. However, what about the package funny-complete that links in the same problematic, conflicting instance? Statically, during the typing of funny-partial, the world inhabited by FunnyBottom is consistent. And in funny-complete, the module hasn't changed and neither has its world. The linking of the FunnyRight implementation does not cause its world to percolate retroactively into that of FunnyBottom.

But, as in Haskell, the FunnyBottom program from funny-complete would see the *misabstraction* and therefore the erroneously constructed list. That's because, regardless of static typing of worlds, the elaboration of this package will be the exact same modules as in the original Haskell example. Though worlds aid the type system in statically ruling out misabstraction, they offer the dynamic execution of a program no such benefits.

What this example shows is that satisfying the World Consistency Property in Backpack ("every module inhabits a consistent world") does not guarantee in the elaborated Haskell program the Global Uniqueness Property ("there are no two conflicting instances defined in the program"). Since the latter is the (admittedly heavyweight) property that rules out misabstraction, we need a way to recover it. Concretely, we need the type system of Backpack to prohibit the funny-complete package.

To recover Global Uniqueness in the elaboration, we add a new requirement to Backpack's type system:

Package-Level Consistency Property. *The world of a package, obtained by merging together the worlds inhabited by all modules in that package's type, must be consistent.*

```
package funny-partial where
  include containers (Data.Set)
                   ⎡  import Data.Set                  ⎤
                   ⎢                                   ⎥
  FunnyTop     =   ⎢ data I = I Int                    ⎥
                   ⎢ instance Eq I where ...           ⎥
                   ⎣ instance Show I where ...         ⎦
                   ⎡  import Data.Set                  ⎤
                   ⎢ import FunnyTop                   ⎥
                   ⎢                                   ⎥
  FunnyLeft    =   ⎢ instance Ord I where ...          ⎥
                   ⎢ insL :: I -> Set I -> Set I       ⎥
                   ⎣ insL = insert                     ⎦
                   ⎡  import Data.Set                  ⎤
                   ⎢ import FunnyTop                   ⎥
  FunnyRight   ::  ⎢                                   ⎥
                   ⎣ insR :: I -> Set I -> Set I       ⎦
                   ⎡  import Data.Set                                  ⎤
                   ⎢ import FunnyTop                                   ⎥
                   ⎢ import FunnyLeft                                  ⎥
                   ⎢ import FunnyRight                                 ⎥
  FunnyBottom  =   ⎢                                                   ⎥
                   ⎢ funnySet :: Set I                                 ⎥
                   ⎢ funnySet =                                        ⎥
                   ⎢ insR (I 1) (insL (I 1) (insL (I 2) empty))        ⎥
                   ⎣ main = print (toAscList funnySet)                 ⎦
-- is this well-typed?
package funny-complete where
  include funny-partial
                 ⎡  import Data.Set                ⎤
                 ⎢ import FunnyTop                 ⎥
                 ⎢                                 ⎥
  FunnyRight  =  ⎢ -- reverse ordering             ⎥
                 ⎢ instance Ord I where ...        ⎥
                 ⎢ insR :: I -> Set I -> Set I     ⎥
                 ⎣ insR = insert                   ⎦
```

Figure 4.6: Revisiting the Funny example from Figure 4.1 with modular abstraction.

By lifting the consistency requirement to the level of entire packages, we can truly rule out any violations of Global Uniqueness—which, by replacing mentions of "global" and "the program" with "the package" is essentially the same property. As with our original motivation for worlds, however, Package-Level Consistency retains the footing in the Backpack type system: it's a property expressible with the semantic objects of module types and in particular it's a clearly stated property about a package's "type," so to speak; see Property A.3, for example.

Returning to the funny-full package, we can see how it would be considered ill-typed. The include binding has a package type Ξ_1 that contains within it a module typing $\nu_L{:}{\tau_L}^{+} @ \omega_L$, while the FunnyRight binding has a package type Ξ_2 that contains the single module typing $\nu_R{:}{\tau_R}^{+} @ \omega_R$, with $\omega_L \not\oplus_? \omega_R$ since they define conflicting instances for $[\nu_0]\mathtt{Ord}\ [\nu_T]\mathtt{I}$. Therefore, by augmenting the definition of the partial merging of package types to require that their combined worlds are consistent with each other, we cannot prove $\Xi_1 \ \oplus_? \Xi_2$ as required for the well-typedness of module bindings, and thus the package is ill-typed.

Package-Level Consistency is unfortunately quite a blunt instrument. By requiring that all worlds across an entire package be mutually consistent, we ensure that the desired World Consistency property holds for each module—but at the cost of ruling out possibly valid packages that define conflicting instances that are never witnessed by any particular module.

The funny-full example demonstrates the trickiness of designing a semantics for type classes in the presence of abstraction as in Backpack. The blunt instrument isn't necessarily the best way to rule out such examples, but it's the present way that I have chosen for Backpack.

4.4.6 *Elaborating Backpack with worlds*

The internal language (IL) of Backpack, into which Backpack packages are elaborated, was designed with a principal goal: to model "plain Haskell" as it exists today. When Backpack is augmented with type classes and worlds, the IL needs to be augmented in turn.

Since worlds in Haskell are a semantical concept used to model existing Haskell behavior of type class instances, worlds in the IL are likewise only a peripheral ingredient to IL's semantics. In particular, the world semantics for plain Haskell, as presented at the beginning of this section, is integrated into the IL. The modified syntax and semantic objects of the IL are presented below:

$$\begin{array}{llll}
\text{(Directory Expressions)} & \mathit{dexp} & ::= & \{\overline{\mathsf{f} \mapsto \mathit{tfexp} @ w}\} \\
\text{(World Facts)} & \mathit{fact} & ::= & \mathtt{instance}\ \mathit{kenv}\ \mathring{\mathit{cls}}\ \mathtt{from}\ \mathsf{f} \\
\text{(Worlds)} & w & ::= & \{\!|\overline{\mathit{fact}}|\!\} \\
\text{(File Environments)} & \mathit{fenv} & ::= & \overline{\mathsf{f} : \mathit{ftyp}^{\mathsf{m}} @ w}
\end{array}$$

This integration of IL worlds (w) directly resembles that of worlds into Backpack's semantic objects of module contexts. It is a straightforward translation from the EL concept into the ingredients of the IL, *e.g.*, with source file names f replacing module identities ν. The typing judgment of the IL must also be augmented to yield the world inhabited by the concerning module source file (*hsmod*), and the merging operation on both directory expressions (*dexp*) and file environments (*fenv*) must be augmented to handle worlds as they are in the EL.

One noteworthy exception is the lack of any kind of Package-Level Consistency Property in the IL. No where in the IL are the various worlds of a directory or environment aggregated and required to be consistent.[33] Instead, as in the presentation of worlds for Haskell, they are only ever analyzed and checked for consistency at the granularity of individual modules. More accurately, they are consistent in the EL and the elaboration preserves that

33 Worlds in the EL are semantic objects and not present in the syntax of the expression language at any level. Worlds in the IL, however, are part of the syntax of the directory expression (*dexp*) via the typed file expression (*tfexp*). On that basis and because the latter better models Haskell, I decided to impose consistency as an implicit invariant on EL worlds but not IL worlds.

consistency; that's one of the consequences of Backpack's primary Elaboration Soundness Theorem (§9.6).[34]

The reader might wonder, why augment the IL with worlds at all, given that the mechanics of (fully closed, complete programs in) the IL *still* just models that of the Haskell spec? The answer is simply that their presence aids technical definitions. Moreover, their presence in directory expressions acts as a certificate of global uniqueness: since each world is consistent by definition, and since a well-typed directory expression *dexp* has for each of its module source files the world that the module inhabits, each module in *dexp* inhabits a consistent world—no module in the program can witness (or execute code from) two conflicting instances.

As with the rest of this chapter, full technical details are provided in the formalization in Chapters 6, 7, and 8.

4.5 SUBTLETIES OF TYPE CLASSES AND ABSTRACTION

The interaction between type classes and modular abstraction introduces a number of subtleties that, in the course of the development of type classes for Backpack, proved quite challenging to pin down. However, once I settled on the world semantics (rather than treating instances as implicitly imported and exported entities, as the Haskell spec does), the subtleties became more holistically comprehensible. We now walk through some examples that illustrate these subtleties.

4.5.1 *Signature matching and world extension*

Earlier it was revealed that worlds of implementations must extend worlds of signatures they match. At first glance, that might seem backwards and overly restrictive, according to one interpretation of substitutability. In that interpretation, the more instance facts a module's world contains, the *less* composable that module is with others, for each of its instances might clash with one in a prospective composed module. Therefore the implementation, which has a larger world, can be used in fewer places than can the signature it matches against, which has a smaller world—a violation of intuition about substituting an implementation for a signature.

A practical example illuminates Backpack's imposed order of extension along signature matching. Consider the following package providing a signature for some module X:

```
package x-sig-1 where
  include prelude-sig
         ┌   import Prelude              ┐
         │                               │
         │ data T                        │
  X ::   │ value :: T                    │
         │ power :: T -> Int -> T        │
         └ toStr :: T -> String          ┘
```

Because this signature imports Prelude (in order to know about `Int` and `String`), its world is as large as Prelude's. Any prospective implementation needs to have a world that extends that world. That might seem like too hefty a requirement.

But in order to match the signature, that implementation needs to know about `Int` and `String`, if not by importing Prelude directly, then by importing something that imported them. And if that's the case then it must know everything Prelude did along with any intervening

34 Elaboration Soundness proves that a well-typed EL package definition elaborates to a *dexp* such that, according to the Soundness Relation (§9.5), each world of a module typing in the EL elaborates directly to a world of a file in the IL. Specifically, for the EL world $\{\!\!|\, \mathit{fact}_1, \ldots, \mathit{fact}_n \,|\!\!\}$ of module ν, the elaboration of ν is a module named $\nu^\star$ with IL world $\{\!\!|\, \mathit{fact}_1{}^\star, \ldots, \mathit{fact}_n{}^\star \,|\!\!\}$, where $(-)^\star$ is just the straightforward translation of module names.

modules. Its world is naturally larger than X's; the seemingly backwards world extension requirement no longer seems so restrictive.

Now let's consider the case that the X signature also imported some particular concrete implementation, for example, because the programmer had some unnecessary imports:

```
package some-lib where
  include prelude-sig
  include containers-sig
  include mtl-sig
             ⎡ import Prelude
             ⎢ import Data.Set
             ⎢ import Control.Monad.FancyStuff
             ⎢
  SomeLib =  ⎢ data Thing a = Blah Blah Blah
             ⎢ thing = sadf
             ⎢
             ⎣ instance Monad Thing where ...
```

```
package x-sig-2 where
  include prelude-sig
  include some-lib
        ⎡ import Prelude
        ⎢ import SomeLib
        ⎢
        ⎢ data T
        ⎢ value :: T
  X ::  ⎢ power :: T -> Int -> T
        ⎢ toStr :: T -> String
        ⎢
        ⎢ data U = MkU Thing
        ⎣ other :: T -> U -> Thing
```

At this point, just by importing SomeLib, X's world in x-sig-2 knows everything from Prelude, Data.Set, Control.Monad.FancyStuff, and the additional instance in SomeLib. Now the burden on an implementation is quite a bit higher! It too has to import that exact same SomeLib, for example. That seems unfortunate. If only the author of X had split it into X and Y, with only Y importing that module, this additional burden of implementation *of X* wouldn't have existed.

That consideration of granularity of dependency (on holes) is exactly the kind of consideration of how best to modularize and parameterize her code that the author of X needs to make under Backpack.

4.5.2 *Coherence and superclass constraints*

The second example concerns not only the visibility of instances within signatures, but also the status of superclass instances required for instance definitions.

```
package superclass-coherence where
  A = [ data T = T Int ]
  B = [   import A
        instance Eq T where ... ]
  C :: [   import A, B
         -- requires an Eq T instance
         instance Ord T ]
  D = [   import A, C
        -- which Eq T instance applied?
        main = print ((T 0) == (T 1)) ]

-- (×) ill-typed package
package incoherent-attempt where
  include superclass-coherence
  B' = [   import A
         instance Eq T where ... ]
  C  = [   import A, B'
         instance Ord T where ... ]
```

In module D which instance of `Eq T` is (statically) chosen for the equality check? Does the C implementation match the signature since they each used a different `Eq T` to validate their `Ord T` instance?

The first question is interesting because it demonstrates a certain kind of *coherence* (*i.e.*, determinism of instance resolution) in the presence of abstract instance declarations in Backpack. In module D, there are two potential ways to construct an instance for the required constraint `Eq T`: (1) by directly applying B's `Eq T` instance known to this module (thanks to the design decision for signatures to obey world semantics just like modules do), or (2) by extracting it, as superclass evidence, from the (abstractly declared) `Ord T` from the hole C.[35] At this point, we don't actually know which `Ord T` instance will come with the implementation of C, so there's no telling which `Eq T` instance will have been used as evidence of its superclass constraint.

Fortunately, as it turns out, the two methods of constructing evidence *must be the same*. The answer to the second question, regarding the alleged implementation of C in incoherent-attempt, shows why. Consider the worlds of the modules among the two packages above:

$$
\begin{aligned}
\omega_A &= \emptyset \\
\omega_B &= \{[\nu_0]\mathtt{Eq}\ [\nu_A]\mathtt{T} \mapsto \nu_B\} \\
\omega_C &= \{[\nu_0]\mathtt{Eq}\ [\nu_A]\mathtt{T} \mapsto \nu_B,\ [\nu_0]\mathtt{Ord}\ [\nu_A]\mathtt{T} \mapsto \beta_{0T}\} \\
\omega_D &= \{[\nu_0]\mathtt{Eq}\ [\nu_A]\mathtt{T} \mapsto \nu_B,\ [\nu_0]\mathtt{Ord}\ [\nu_A]\mathtt{T} \mapsto \beta_{0T}\} \\
\omega'_B &= \{[\nu_0]\mathtt{Eq}\ [\nu_A]\mathtt{T} \mapsto \nu'_B\} \\
\omega'_C &= \{[\nu_0]\mathtt{Eq}\ [\nu_A]\mathtt{T} \mapsto \nu'_B,\ [\nu_0]\mathtt{Ord}\ [\nu_A]\mathtt{T} \mapsto \nu'_C\}
\end{aligned}
$$

(where β_{0T} is the variable for the identity that implements the abstractly declared `Ord T` instance in the hole C). As discussed earlier, the world of an implementation must extend the world of a hole—a natural extension of Backpack's signature matching. But in the case of C in incoherent-attempt, this is not the case: the world of the hole, ω_C, contains an instance for `Eq T` from module identity ν_B, while the world of the hole, ω'_C, contains an instance for the

35 Proof systems for type class constraint solving often include a rule for proving such superclass constraints, *e.g.*, (Peyton Jones *et al.*, 1997b; Sulzmann, 2006).

same constraint but from a *different* module identity, ν'_B, *i.e.*, the instance defined in B', not B. Then the implementation does not match the hole and this package, incoherent-attempt, is therefore *ill-typed*. Returning to the coherence issue, this shows that a valid implementation of C *must* have known the same `Eq T` instance that the hole knew, thus ruling out incoherence stemming from the abstract instance declaration.

The fastidious reader might wonder what makes these example packages tricky, for Package-Level Consistency rules incoherent-attempt ill-typed as soon as it merges (the types of) a binding for module B', which has an instance for $[\nu_0]$Eq $[\nu_A]$T, and the included binding for module B, which has a *different* instance for the same class constraint. Here, yes, *the blunt instrument of Package-Level Consistency makes the example quite moot*. But the example nonetheless highlights the subtleties behind mixing type classes and modular abstraction—possibly for a future attempt with a different semantics.

4.5.3 *Instances on abstract types*

The final subtle example highlights what happens when instances are defined on abstractly declared types from signatures.

```
package uncertain where
  X₁ :: [data T]
  X₂ :: [data T]
          ┌  import qualified X₁, X₂  ┐
  Y   =   │ instance Eq X₁.T where ... │
          └ instance Eq X₂.T where ... ┘

-- (×) ill-typed package
package certainly-bad where
  include uncertain
  X   = [data T = MkT Int]
  X₁  = X
  X₂  = X
```

Is module Y well-typed, considering that it doesn't completely know the identities of the two T types? If uncertain is well-typed, then what happens in certainly-bad when X_1 and X_2 are bound to be aliases?

The world inhabited by the Y module knows only about its two locally defined instances:

$$\omega_Y = \left\{ \begin{array}{lcl} [\nu_0]\mathsf{Eq}\ [\beta_1]\mathsf{T} & \mapsto & \nu_Y, \\ [\nu_0]\mathsf{Eq}\ [\beta_2]\mathsf{T} & \mapsto & \nu_Y \end{array} \right\}$$

(where β_1 and β_2 are the identities of T in X_1 and in X_2 respectively). There's something fishy about these two instances. They *almost* overlap, as they are defined on two Haskell types, $[\beta_1]$T and $[\beta_2]$T, that are distinct but unifiable in their free module identity variables. At the moment, within uncertain, there's no reason to worry since they don't actually overlap; the world of Y is consistent and this package is well-typed.

But the next package, certainly-bad, is *not* well-typed. In this package, the two holes X_1 and X_2 are linked together (with an *alias* binding, §2.5), and thus β_1 and β_2 will be unified by some linking substitution θ. The application of θ to the world ω_Y would be undefined since it would unify two distinct instances' class constraints. In other words, applying a substitution θ to a world ω is a partial function that is only defined when θ is an injection on ω's class constraints; indeed, that's how substitution maintains the implicit consistent$(-)$ invariant on worlds (Figure 4.5).

Let's put that technical description of substitution's definedness criterion in simpler words. This example show that linking requires we check existing module worlds, in the module context (Φ), to make sure they are still consistent after substitution. It shouldn't be so surprising that linking could potentially result in inconsistent worlds, however. In §4.2 we demonstrated that module composition in Haskell—expressed as the import of two modules into the same client module—does not necessarily preserve world consistency. Since Backpack offers a much richer system of modularity, it offers a richer language for composing modules. And, as is the case in Haskell, composing modules does not necessarily preserve world consistency. That's why, with Backpack, we aimed to integrate world consistency into the well-typedness of modules in the first place.

Part II

FORMALIZATION OF BACKPACK

5 OVERVIEW OF THE FORMALIZATION

In Part I of this dissertation I've presented intuition and examples for how to make sense of programs in Backpack. What we haven't seen yet is a technical definition for that intuition; that's the subject of Part II over the course of the next several chapters. In this chapter I describe the formalization of Backpack at a high level. Not only does the formalization fully and precisely define Backpack *as a type system* spanning both an External Language (EL) and Internal Language (IL)—along with the elaboration of the former into the latter—it also includes metatheory that validates that definition. Figure 5.1 presents a diagram of the formalization.

The principal, outermost typing (and elaboration) judgment of Backpack is as follows:

$$\Delta \vdash \mathrm{D} : \forall\overline{\alpha}.\Xi \rightsquigarrow \lambda\overline{f}.\mathit{dexp}$$

A package definition D has package type $\forall\overline{\alpha}.\Xi$ and elaborates to the parameterized IL expression $\lambda\overline{f}.\mathit{dexp}$. Explaining the definition of—and justification for—that judgment and its constituent parts is the goal of this part of the dissertation, making more precise and concrete what was informally presented in Chapter 3. The package level is covered in Chapter 8.

That judgment is part of the outermost level of Backpack: *the package level*. Backpack is formalized not just as this new layer atop Haskell, but as an entire three-layer cake. Below the package level is *the module level*, where the two principal typing (and elaboration) judgments are:

$$\Gamma; \nu_0 \vdash \mathrm{M} : \tau@\omega \rightsquigarrow \mathit{hsmod}$$
$$\Gamma; \rho \vdash \mathrm{S} : \sigma@\omega \mid \Phi_{\mathsf{sig}}$$

Both judgments describe how a module or signature expression is typed in some module context, with the module expression additionally elaborating to the IL expression (read: Haskell module definition) *hsmod*. This module level of Backpack is mostly imported wholesale from Haskell, retaining all its complexity while adding the additional notion of signatures. This level is completely defined in the formalization, all in Chapter 7, with some spillover from Chapter 6 as well.

Finally, below the module level is the innermost level of Backpack: *the core level*. "Here be dragons," as the saying goes. This is where the main focus of the world of Haskell has always existed; it's the domain of values, types, functions, data constructors, type classes and instances, *etc.* For that reason, it's actually the least interesting to the Backpack formalization. There the principal typing judgments are:

$$\Phi; \nu_0; \mathit{eenv}; \omega \vdash \mathit{defs} : \mathit{dspcs}$$
$$\Phi; \overline{\nu}; \mathit{eenv}; \omega \vdash \mathit{decls} : \mathit{dspcs}$$

These judgments describe core-level definitions (in modules) and—new compared to Haskell—declarations (in signatures) are typed with respect to contextual objects determined at the module level. This level, unlike the other two, is *not* fully defined in the Backpack formalization. Instead, I assume the existence of these (invented) judgments and axiomatize some properties expected of them. The core level is covered first, in Chapter 6.

Altogether, the syntax, semantics, and metatheory of the core level, then module level, and then package level *of the EL* will be presented in turn. Following that will be the syntax, semantics, and metatheory *of the IL*—essentially a model of Haskell's core and module

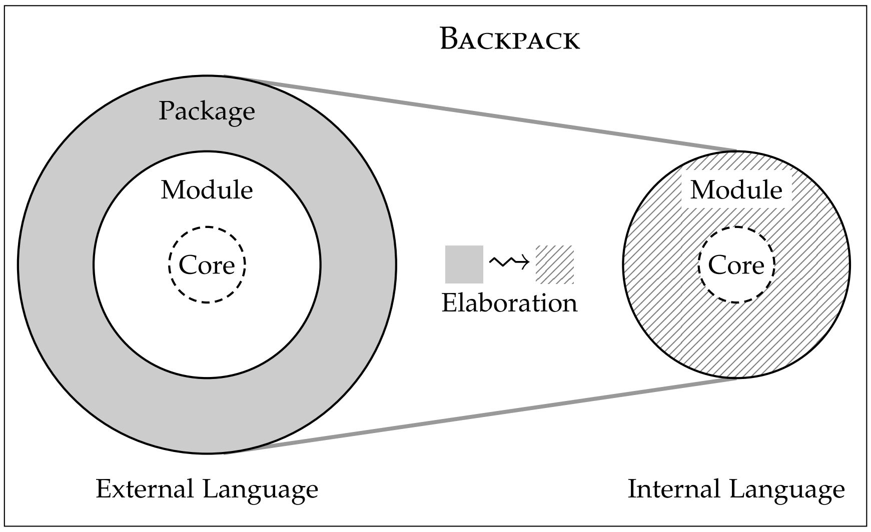


Figure 5.1: Diagram of the entire Backpack formalization.

levels. Finally, the elaboration semantics will be presented, along with the core Elaboration Soundness Theorem.

* * *

The formalization of Backpack, or any other language for that matter, is hardly a deterministic consequence of its intended design and semantics. It is instead, like Backpack itself, an object designed according to certain goals and trade-offs.

GOAL: PRECISE DEFINITION AS TYPE SYSTEM

Expressions in Backpack are organized into a type system: well-formed expressions are classified by *types* according to inductively defined typing judgments. These typing judgments are defined almost entirely deterministically—exceptions will be discussed later —and thus collectively serve, along with shaping, as a *typechecking algorithm* for Backpack. Altogether, this typechecking algorithm realizes the intuition presented in the earlier chapter.

NESTED LEVELS Backpack's type system is quite large. To manage the complexity, its syntax and semantics are stratified across three levels: package, module, and core, which correspond respectively to the new package constructs; modules, signatures, and import/export resolution; and Haskell types, values, and typechecking.[1] Moreover, this stratification serves as a "parameterization" of the package and module levels over the core level, effectively abstracting Backpack away from the specifics of Haskell types, values, and typechecking. For exactly these reasons, such stratification is common in formalizations of type systems for ML modules—minus the additional package level, however.

SEMANTIC OBJECTS Unlike many type systems, including some earlier ones for ML modules, Backpack's types do not manifest in the syntax of expressions: there are no "syntactic signatures." For example, the types of module expressions (M) have no representation as expressions; they are not signature expressions (S). Instead, Backpack's types are defined as—and in terms of—*semantic objects*.

Some of Backpack's semantic objects carry additional structure beyond their syntactic representation. For example, given the prevalence of merging operations (due to mixin composition), many of them are idempotent *partial commutative monoids* (PCMs). The formalization

1 While they together constitute a single type system, I sometimes refer to them as three different ones.

defines the syntactic elements of such objects, like module types (τ), along with a (partial) merging operation ($\oplus$) and identity element ($\cdot$) that satisfy laws of idempotence, associativity, commutativity, and identity. As an example of other structure, a physical module context (Φ) acts as a finite mapping from module identities (ν) to polarities (m), module types (τ), and worlds (ω). As such, two contexts are equivalent as mappings, despite the syntactically written order of their constituent objects. Moreover, physical module contexts are idempotent PCMs with a standard merging definition for finite mappings.

WELL-FORMEDNESS Whereas some semantic objects satisfy properties which are "baked in," like being an idempotent PCM, most semantic objects have an additional notion of *well-formedness* in some context. Well-formedness is captured by an additional judgment that ascertains whether a syntactically valid semantic object satisfies certain properties with respect to a context. For example, if a module type τ mentions some Haskell type $[\nu]\mathsf{T}$, then τ would not be well-formed in context Γ if Γ did not contain a module ν that defines type T.

METATHEORY Not just a type system, the Backpack formalization also includes metatheory, *i.e.*, theorems about the type system (and elaboration, as we'll see shortly) that support and validate the definitions. The metatheory of type systems based on the lambda calculus often include so-called structural properties like *weakening* and *substitution*. Indeed, the internal language of Backpack relies on such properties in order to prove the central elaboration soundness theorem. As another example, typing judgments on Backpack expressions satisfy a regularity property: when an expression is classified by a type in some context, then that type is itself well-formed in the same (or some related) context. Then if the judgment $\Delta;\Gamma;\hat{\Xi}_{\mathsf{pkg}} \vdash B : \Xi$ is derivable ("package binding B has type Ξ") then so is $\Gamma \vdash \Xi$ wf ("package type Ξ is well-formed in context Γ").

GOAL: ELABORATION INTO MODULES

The formalization actually contains two languages: Backpack, the *external language* (EL), and plain Haskell, the *internal language* (IL). Each of the two languages is specified with its own type system (and accompanying metatheory), and they are linked together with an elaboration from EL to IL (and its accompanying metatheory).

INTERNAL LANGUAGE "FILESYSTEM" As a model of plain Haskell, the IL largely resembles the fragment of the EL restricted to (1) the module and core levels and (2) concrete modules, *i.e.*, no abstraction. This resemblance exists in the syntax, in the type systems, and, to a lesser extent, in the metatheory.

While the EL comprises the package, module, and core levels mentioned above, the IL comprises the filesystem, module, and core levels. Instead of the EL's rich package level the IL has only a light *filesystem* level that groups together expressions at the module level: syntactically, the IL's filesystem level organizes module expressions into *files*, accompanied by *file types* and bundled together in *directories*. This filesystem language intentionally resembles the representation of a multimodule Haskell program in GHC: a module expression is defined in a `.hs` file; a module interface is described in a `.hi` file; and all constituent modules are grouped together in a filesystem directory (or nesting thereof).

ELABORATION The elaboration effectively "compiles away" the package level of the EL. More concretely, the elaboration leaves the module and core levels of an EL expression mostly unchanged—crucially, except for module naming—and replaces the package level with the filesystem level for the resulting IL expression.

Although the type system specifies how to mechanically characterize valid Backpack expressions, the type system itself doesn't claim to represent Backpack's "semantics." Instead, it is the elaboration that bears the weight of that claim, for it is the elaboration that explains the essence of retrofitting a *weakly* modular system with *strong* modularity. To bear that weight, a statement *and proof* of *soundness* technically validates the elaboration's preservation of meaning and structure.

NO DYNAMIC SEMANTICS Following in the research tradition of the ML module system, Backpack's EL is defined via the static semantics of typing and elaboration; there is no dynamic semantics. Is that a limitation of the formalization? No, it's not. Because of the Haskell core language's purity, there is no core-level evaluation that happens as the result of processing a module in the IL. Then the evaluation of a program in the IL (*i.e.*, a directory expression with a "main" module file) would not be concerned with the order in which its module files were defined or processed.[2]

GOAL: COMPATIBILITY WITH HASKELL

One of the goals of the formalization is to maintain compatibility with Haskell. Given the stratification into a type system across three levels, that goal concretely means that the type system at the module level (and the core level subsequently) should adhere as closely as possible to the semantics of Haskell described in the Haskell 98 specification.[3]

While the Haskell 98 specification informs the design of the type system, it does not and cannot fully determine it, for Backpack adds a package level and extends the module level. For the parts of Backpack that already exist in Haskell, the type system follows Haskell 98; elsewhere, it reuses ingredients from the former, like import resolution and world determination.

2 In particular, recursive modules in the IL need not be concerned with the order in which they are loaded and processed. This simplification cannot be made with recursive modules in ML, where the evaluation of core-level bindings in a module might produce side effects, for example, by printing to the screen.

3 I use the Haskell 98 and Haskell 2010 specifications interchangeably, as they are identical with respect to the module system and identification of core entities. The one exception is the addition in 2010 of hierarchical module names, a purely cosmetic convention. For more information, see `https://www.haskell.org/onlinereport/haskell2010/haskellli2.html#x3-5000`

6 CORE LEVEL

We begin with the innermost level of Backpack: the core. At this level a program consists of a list of *Haskell entity bindings*, either concrete *definitions* (inside modules) or abstract *declarations* (inside signatures). The type system for the core level is primarily described by the following two typing judgments for definitions and declarations respectively:

- $\Phi; \nu_0; \mathit{eenv}; \omega \vdash \mathit{defs} : \mathit{dspcs}$ and
- $\Phi; \overline{\nu}; \mathit{eenv}; \omega \vdash \mathit{decls} : \mathit{dspcs}$.

INPUTS (FROM MODULE LEVEL) From the module level, the core level uses a *physical module context* (Φ) giving types to module identities; a module identity or list of identities (ν_0 or $\overline{\nu_0}$) determining the provenance of the corresponding entity bindings, respectively; an *entity environment* (*eenv*) mapping the syntactic references in the bindings to their unique *physical names* from the physical module context; and a *world* (ω) describing the type class facts known to these bindings.

OUTPUTS (TO MODULE LEVEL) The primary goal of the core level is to classify entity bindings with their corresponding *defined entity specifications* (*dspcs*), which act as the canonical static descriptions for those entities. Given (the unique physical name of) a value entity x, what is its type? The answer is determined by consulting the *dspc* for x—although how to obtain that *dspc* within a particular context is a problem for the module level.

INTERNALS As the innermost level of the Backpack formalization, there is no underlying level to employ in the internals of the core. Instead, the actual semantics of typing entity bindings is axiomatized and left undefined in the formalization. At exactly this point, as with countless other formalizations of ML module systems , Backpack abstracts over the (orthogonal) details of the underlying programming language, resulting in a smaller and more generic formalization.

ABSTRACTION BARRIER In the Backpack formalization the core level models that of Haskell. But we are not concerned with much of the details of Haskell's core level. Backpack's core level specifies only those portions of Haskell's core level necessary to build up the module and package levels. For that reason, it defines core entity names and bindings but not the full syntax of terms and types and especially not the semantics of typechecking on those terms. This delineation between what is defined and what is undefined is the *abstraction barrier* of Backpack's core level. We will see shortly how the abstraction barrier manifests in the syntax, semantics, and metatheory of the core level.

6.1 SYNTAX OF THE CORE LEVEL

The syntax of the core level is presented in Figure 6.1. Generally speaking, the core syntax is defined to be that of the underlying "programming language" of Haskell, *i.e.*, the types and terms of Haskell. However, for the purposes of simplicity and genericity, the formalization abstracts over the details of types (*utyp*) and terms (*uexp*). Instead, Backpack is concerned only with the named *bindings* of types and terms, collectively called *entities*, and with the syntactic references to these bound entities.

Backpack Core Syntax

Entity Names and References

Value Names	x	$\in$	$\mathit{ValueNames}$	
Datatype Names	T	$\in$	$\mathit{TypeNames}$	
Constructor Names	K	$\in$	$\mathit{CtorNames}$	
Class Names	C	$\in$	$\mathit{ClassNames}$	
Entity Names	χ	$::=$	$x \mid T \mid K \mid C$	
Entity References	eref	$::=$	χ	Unqualified Reference
		$\mid$	$\mathit{mref}\,\texttt{.}\chi$	Qualified Reference
Module References	mref	$::=$	$\texttt{Local}$	Local Module Name
		$\mid$	ℓ	Imported Module Name

Types and Terms

Type Variables	α	$\in$	$\mathit{TypeVars}$	
Types	utyp	$::=$	eref	Datatype Reference
		$\mid$	$\ldots$	
Value Exprs	uexp	$::=$	eref	Value Reference
		$\mid$	$\ldots$	
Kinds	knd	$::=$	$\texttt{*}$	Type Kind
		$\mid$	$\mathit{knd}\ \texttt{->}\ \mathit{knd}$	Arrow Kind
Kind Environments	kenv	$::=$	$\overline{\alpha\ \texttt{::}\ \mathit{knd}}$	
Class Constraints	ucls	$::=$	$\mathit{eref}\ \overline{\mathit{utyp}}$	

Entity Bindings

Entity Definitions	def	$::=$	$\texttt{data}\ T\ \mathit{kenv}\ \texttt{=}\ \overline{K\ \overline{\mathit{utyp}}}$	Datatype Definition
		$\mid$	$x\ \texttt{::}\ \mathit{utyp}\ \texttt{=}\ \mathit{uexp}$	Value Definition
		$\mid$	$\texttt{class}\ C\ \mathit{kenv}\ \overline{\texttt{<=}\,\mathit{ucls}}$ $\quad\texttt{where}\ \overline{x\ \texttt{::}\ \mathit{utyp}}$	Class Definition
		$\mid$	$\texttt{instance}\ \mathit{kenv}\ \overline{\mathit{ucls}\,\texttt{=>}}\ \mathit{ucls}$ $\quad\texttt{where}\ \overline{x\ \texttt{=}\ \mathit{uexp}}$	Instance
Entity Declarations	decl	$::=$	$\texttt{data}\ T\ \mathit{kenv}$	Abstract Datatype Declaration
		$\mid$	$\texttt{data}\ T\ \mathit{kenv}\ \texttt{=}\ \overline{K\ \overline{\mathit{utyp}}}$	Concrete Datatype Declaration
		$\mid$	$x\ \texttt{::}\ \mathit{utyp}$	Value Declaration
		$\mid$	$\texttt{class}\ C\ \mathit{kenv}\ \overline{\texttt{<=}\,\mathit{ucls}}$ $\quad\texttt{where}\ \overline{x\ \texttt{::}\ \mathit{utyp}}$	Class Declaration
		$\mid$	$\texttt{instance}\ \mathit{kenv}\ \overline{\mathit{ucls}\,\texttt{=>}}\ \mathit{ucls}$	Instance Declaration
List of Entity Definitions	defs	$::=$	$\overline{\mathit{def}}$	
List of Entity Declarations	decls	$::=$	$\overline{\mathit{decl}}$	

Figure 6.1: Syntax of Backpack's core level.

ENTITY NAMES AND REFERENCES Entity names are drawn from distinct sets and therefore distinct namespaces. An entity name χ is either a value name x (*e.g.*, `map`, a datatype name T (*e.g.*, `Maybe`), a datatype constructor name K (*e.g.*, `Just`), or a type class name C (*e.g.*, `Eq`). An entity χ can be referenced (*eref*) either qualified by some module reference (*e.g.*, the `map` in `L.map`) or unqualified (*e.g.*, the `Maybe` in `Maybe Int`).

In the case of a qualified entity reference, a module reference (*mref*) can be either the keyword `Local`, which refers to the present module or signature , or a logical module name ℓ (*e.g.*, `Data.Set`), which refers to some imported module that exports the entity. The core-level syntax otherwise contains no mention of module names; such names are introduced into the scope of a program by `import` statements at the module level.

TYPES AND TERMS The syntax of types (*utyp*) and terms (*uexp*), as already stated, is more or less inconsequential to Backpack. The sole interesting part of each is the inclusion of entity references to either a datatype entity T or a value entity x; the rest falls on the other, undefined side of the abstraction barrier. The formalization's omission of other kinds of types and terms in the core level—denoted by the ellipses (...)—follows many type systems for ML modules[1].

In addition to types and terms, the core-level syntax contains type class constraints (*ucls*) which likewise include references to type class entities C (applied to some number of type expressions).

The syntax also contains kinds (*knd*) to describe higher-kinded types. A kind environment (*kenv*) simply associates type variables with their kinds. Throughout the formalization, we presume that kind environments bind type variables over a constituent type or class constraint, where obvious, and thus obey standard α-equivalence on top of syntactic equivalence.

ENTITY BINDINGS At the core level, a program consists of a sequence of entity bindings that describe them. If the program is a signature, then the bindings are all *declarations*, in which no value expressions (*i.e.*, executable code) exists, and in which datatype entities may be declared abstractly without their constructors. Otherwise, if the program is a module, then the bindings are all *definitions*, in which value expressions exist in the cases of value bindings (x `::` *utyp* `=` *uexp*) and instance definitions (the method implementations $\overline{x = uexp}$).

Each entity binding describes a set of entity names. A value binding introduces only its name x; a datatype binding introduces its name T along with any constructor names $\overline{K}$ it describes; a type class binding introduces its name C along with any class method names $\overline{x}$ it describes; and an instance binding introduces no names. In the case of datatypes and type classes entities, the additional names are referred to as *subordinate names* of the entity.

6.1.1 *Comparison with Haskell 98*

- Module references in Backpack additionally include a reserved reference `Local`. Like a "self" reference in many OO languages, this module reference always identifies the current, ambient module (or signature). In Haskell, one can write a qualified reference to a locally defined entity χ in module ℓ as $\ell.\chi$. In Backpack, however, this could be written as `Local.`χ.
- Type synonyms and `newtype` declarations are excluded. While the latter would manifest in the formalization exactly like datatypes with one constructor, the former would be a little more involved.
- Value definitions must include type annotations. There's no particular reason for this other than to simplify presentation, as the Backpack formalization is not concerned with core-level type inference. See §6.4 for more.

1 Leroy (1994, 2000), Rossberg and Dreyer (2013), and Rossberg *et al.* (2014).

- As discussed in Chapter 4, the syntax of type classes and instances is more permissive than in Haskell 98. One restriction, however, is the absence of default implementations of class methods.

6.2 SEMANTICS OF THE CORE LEVEL

The semantics of the core level is defined as a type system of entity bindings: a program, represented as a list of entity bindings, is classified by *semantic objects* according to a *typing judgment*. For example, . The design of this type system introduces two main challenges.

SEMANTIC OBJECTS First, the semantic objects that act as "types" for entity bindings must refer to other bound core entities by name. For example, the "type" of the value definition for `member` must somehow mention the datatypes `Set` and `Bool`. Then we need a constituent semantic object to represent each bound core entity. Moreover, as we saw in §3.1, the core-level semantics must distinguish two Haskell types with the same entity name, such as `Foo.T` and `Bar.T` (for which we introduced the notion of *module identity*).

Because they permeate every level of the formalization, the real meat of the core semantics lies in the definition of semantic objects.

TYPING JUDGMENTS Second, the typing judgment for a program necessarily involves the actual static semantics of Haskell typing, kinding, and class constraint solving. Otherwise, how might the semantics determine which value definitions have well-typed expressions (matching their annotated types); which datatype definitions have well-kinded constructors; which instance definitions have their superclass constraints satisfied; and so on.

As already discussed, the answers to such questions lie on the other side of the abstraction barrier; the Backpack formalization does not provide them. Instead, the formalization merely *specifies* the typing judgments and, in the next section, imposes some axioms about their behavior.

As a concrete example of a core-level program, this section will focus primarily on a Backpack-ification of Haskell's canonical implementation of a set data structure.[2] Figure 6.2 provides the core-level definitions of this module, while Figure 6.3 provides the core-level declarations of a corresponding signature. This implementation of `Set` uses a straightforward binary tree to store the elements at the branch nodes, represented by the data constructor `Bin`. Crucially, the implementation of some operations, like `member`, requires an ordering on the element type, denoted as the `Ord a` class constraint.

As an additional example, of more interesting types and kinds, Figure 6.4 provides the core-level declarations of a signature for Haskell's canonical indexed-arrays implementation.[3]

6.3 SEMANTIC OBJECTS

The semantic objects of the core level are defined in Figure 6.5. As in all levels, these semantic objects represent bound names and classify syntactic expressions. Since the core is the innermost level, they pervade the entire formalization—physical entity names (*phnm*) in particular. Certain semantic objects that are critical to understanding the core level, like worlds (ω), will be defined in the next chapter, on the module level (Chapter 7).

In this section I describe these core-level semantic objects with examples of (core-level) programs.

2 The Data.Set module defined in the containers package: http://hackage.haskell.org/package/containers-0.5.6.3/docs/Data-Set.html

3 The Data.Array.IArray module defined in the arrays package: http://hackage.haskell.org/package/array-0.5.3.0/docs/Data-Array-IArray.html

```
data Set a = Tip | Bin a (Set a) (Set a)

empty :: Set a
empty = Tip

member :: Ord a => a -> Set a -> Bool
member = \ x s -> case s of
  Tip -> False
  Bin y l r -> case compare x y of
    LT -> member x l
    GT -> member x r
    EQ -> True

insert :: Ord a => a -> Set a -> Set a
insert = ...

toAscList :: Set a -> [a]
toAscList = \ s -> case s of
  Tip -> Nil
  Bin x l r -> append (toAscList l) (cons x (toAscList r))

instance Eq a => Eq (Set a) where
  (==) = \ s1 s2 -> toAscList s1 == toAscList s2

instance Ord a => Ord (Set a) where
  compare = \ s1 s2 -> compare (toAscList s1) (toAscList s2)
```

Figure 6.2: A slightly simplified excerpt of the Data.Set implementation. Within this excerpt, the only change to the standard implementation is the omission of a size field on the Set type.

```
data Set a

empty :: Set a
member :: Ord a => a -> Set a -> Bool
insert :: Ord a => a -> Set a -> Set a
toAscList :: Set a -> [a]

instance Eq a => Eq (Set a)
instance Ord a => Ord (Set a)
```

Figure 6.3: An excerpt of the Data.Set implementation converted to signature declarations. Note that Set has been made abstract.

```
class IArray (a :: * -> * -> *) e where
  bounds      :: Ix i => a i e -> (i, i)
  numElements :: Ix i => a i e -> Int

data Array i e

(!) :: (IArray a e, Ix i) => a i e -> i -> e
```

Figure 6.4: An excerpt of the Data.Array.IArray implementation converted to signature declarations.

Backpack Core Semantic Objects

Physical Entity Names	*phnm*	::=	$[\nu]\chi$	
Semantic Types	*typ*	::=	$[\nu]\mathsf{T}\ \overline{typ}$	Datatype (Application)
		|	`forall` $kenv.\ \overline{cls}$ `=>` typ	Polymorphic Type
		|	$\mathsf{a}\ \overline{typ}$	Type Variable (Application)
Semantic Class Constraints	*cls*	::=	$[\nu]\mathsf{C}\ \overline{typ}$	
Semantic Instance Heads	*head*	::=	$kenv\,.\,cls$	
Entity Definition Specs	*dspc*	::=	`data` $\mathsf{T}\ kenv$	Abstract Datatype
		|	`data` $\mathsf{T}\ kenv = \overline{\mathsf{K}\ \overline{typ}}$	Concrete Datatype
		|	$\mathtt{x} :: typ$	Value
		|	`class` $\mathsf{C}\ kenv\ \{\overline{cls}\}\ \overline{\mathtt{x} :: typ}$	Class
		|	`instance` $kenv\ \{\overline{cls}\}\ cls$	Instance
Entity Name Specs	*espc*	::=	$[\nu]\chi$	Simple Entity
		|	$[\nu]\chi(\overline{\chi'})$	Entity with Subnames
Entity Name Spec Sets	*espcs*	::=	$\overline{espc}$	
Entity Environments	*eenv*	::=	$\{\overline{eref \mapsto phnm}\};\ espcs$	
Module Types	τ	::=	$\llparenthesis\ \overline{dspc}\ ;\ espcs\ ;\ \overline{\nu}\ \rrparenthesis$	
Module Worlds	ω	::=	$\{\!\lvert\ \overline{head \mapsto \nu}\ \rvert\!\}$	
Physical Module Contexts	Φ	::=	$\{\!\lvert\ \overline{\nu{:}\tau^{m} @\, \omega}\ \rvert\!\}$	

Figure 6.5: Semantic objects relevant to Backpack's core level, with key objects in bold. Although presented here since they're mentioned in the semantics of the core level, module-level semantic objects like worlds (ω) and module types (τ) will be presented in Chapter 7.

physical names To get a sense for the ways in which nested semantic objects at the core level permeate the formalization at all levels, consider the definition of the value entity `member` in Figure 6.2. This function tests whether an element is in a given `Set`, in part by calling the `compare` function, which uses the `Ord a` constraint, to determine which side of the tree to search.

What would the type of `member` look like? In most type systems for Haskell,[4] the type would be some variation of

$$\forall\alpha.\,\mathtt{Ord}\ \alpha \Rightarrow \alpha \rightarrow \mathtt{Set}\ \alpha \rightarrow \mathtt{Bool}$$

where `Ord` and `Set`/`Bool` refer to class and type names (respectively) bound in some context. But that context should accept multiple *distinct* classes and types, defined in distinct modules, that happen to share the same syntactic names. For example, the type system of Faxén[5] would record, in its type of `member`, the *local name* `Set` since it denotes a type defined in the same, local module but also the *original name* `GHC.Types.Bool` since it denotes a type defined in an imported module. We therefore need a way to distinguish, in the semantic objects, similarly-named entities bound in distinct modules—and that was exactly the motivation for tracking module identity in 3.1.

Backpack uses *module identities* (ν) to represent the original, binding module for a core entity. Any entity named χ that was originally bound in the module with identity ν can then be uniquely identified with the *physical name* (*phnm*) $[\nu]\chi$. Physical names are thus used, for example, to distinguish between the bound occurrence of `Set` from module ν_S, $[\nu_S]$`Set`, and that of `Set` from some other module ν_S', $[\nu_S']$`Set`. (How the module identity ν_S got associated

4 Jones (1994), "Qualified Types: Theory and Practice".
5 Faxén (2002), "A static semantics for Haskell".

with `Set` will be described shortly.) Moreover, they generalize to uniquely identify all named entities, like $[\nu_S]$`member`.

semantic types and class constraints We now see how physical names play an important role in the representation of Haskell types, For example, consider the actual *semantic type* (*typ*) of `member`:

$$typ_1 \triangleq \texttt{forall}\ (\alpha :: *).\ [\nu_P]\texttt{Ord}\ \alpha \Rightarrow \alpha \to [\nu_S]\texttt{Set}\ \alpha \to [\nu_P]\texttt{Bool}$$

This type is more or less identical to the example type given above, modulo some cosmetic changes. In particular, the three *syntactic* class and type names are translated to physical names.

The most interesting form of *typ* is the datatype reference, $[\nu]\mathsf{T}\ \overline{typ}$, since this form explicitly mentions a *physical name* of a type: $[\nu]\mathsf{T}$. Such physical names of entities (*phnms*) enable the distinction of similarly-named types, a distinction that is necessary for defining type equivalence.

Polymorphic types are represented with a simple kind environment (*kenv*) that *binds* core-level type variables (α) over some constituent type (*typ*). We assume standard alpha-renaming of the type variables on these binders. Type variable references $\alpha\ \overline{typ}$ are in weak-head normal form simply to avoid a separate form of semantic type for applications (apart from the datatype references).

We keep types in normal form so that type equivalence is just syntactic equality (modulo alpha-conversion for polymorphic types) on *typ*. Because this restricted syntax of Haskell types[6] lacks type-level computation, there is no nontrivial equivalence necessary for core-level types in Backpack. (See §11.2 about the future work of supporting more advanced features like type families.)

Class constraints are also kept in normal form, thereby also enabling equivalence to be defined as syntactic equality—and here too, polymorphism stemming from binding kind environments on class constraints means that they can be alpha-converted.

entity environments and interpretation of syntax The translation of the syntactic type $utyp_1$, which is annotated on `member`'s definition, to the semantic one typ_1, is an example of a key operation in the formalization: the *interpretation* of syntactic terms into semantic objects.

But how did the syntactic references to class and type names `Ord`, `Set`, and `Bool` get translated to the physical names $[\nu_P]$`Ord`, $[\nu_S]$`Set`, and $[\nu_P]$`Bool`? By an accompanying *entity environment* (*eenv*) that maps the syntactic references to physical names. Entity environments are considered "inputs" to the core level, having been constructed in the module level. They're constructed by processing the import statements and local bindings of a module (or signature): each import statement and each local binding yields some portion of the entity environment, and those portions are all merged together. (The full details will be given in §7.2.)

The running example has been implicitly governed by the entity environment $eenv_1$, defined in Figure 6.6. A realistic entity environment for this program would contain additional mappings for *qualified names*, but we omit them for the sake of the discussion of the core level. The second component of $eenv_1$, the in-scope entity names, will be explained shortly.

The interpretation of syntactic terms into semantic objects is realized in the formalization with a straightforward partial operation applying an entity environment to a syntactic term. For example, $typ_1 = eenv_1(utyp_1)$. Interpretation of an entity reference (*eref*) simply behaves like the underlying mapping of the environment, whenever that mapping acts as a function for that particular reference. See Figure 6.7 for the full definition. Interpretation is also defined recursively on syntactic terms that contain entity references: class constraints ($ucls \mapsto cls$), and types ($utyp \mapsto typ$).

6 The Haskell 98spec, which lacks some of the richer, more modern type features like type families.

$$
\mathit{eenv}_1 \quad \triangleq \quad \left\{ \begin{array}{lcl}
\texttt{Bool} & \mapsto & [\nu_P]\texttt{Bool} \\
\texttt{Ord} & \mapsto & [\nu_P]\texttt{Ord} \\
\texttt{compare} & \mapsto & [\nu_P]\texttt{compare} \\
\texttt{Set} & \mapsto & [\nu_S]\texttt{Set} \\
\texttt{Bin} & \mapsto & [\nu_S]\texttt{Bin} \\
\texttt{Tip} & \mapsto & [\nu_S]\texttt{Tip} \\
\texttt{member} & \mapsto & [\nu_S]\texttt{member} \\
 & \vdots &
\end{array} \right\} ; \left(\begin{array}{l}
[\nu_P]\texttt{Bool()} \\
[\nu_P]\texttt{Ord(compare)} \\
[\nu_S]\texttt{Set(Bin,Tip)} \\
[\nu_S]\texttt{member}
\end{array} \right)
$$

Figure 6.6: Sample entity environment for the running example.

ENTITY NAME SPECIFICATIONS The second component of the entity environment, the *entity name specifications* (*espcs*), lists the entity names that are considered "in scope" within a program. Like the mapping of name references to physical names, it contains all the names gotten from import statements and from the local bindings of the program. (Hence the presence of imported names like $[\nu_P]$`Ord` and local names like $[\nu_S]$`member` in eenv_1.) Entity name specs look just like physical names but with one exception: for entities with subordinate names (*i.e.*, datatype names and class names), the spec contains a list of some number of those subordinate names. For example, the presence of $[\nu_S]$`Set(Bin,Tip)` in eenv_1 means that the subordinate names `Bin` and `Tip`, `Set`'s data constructors, are also in scope. It's possible that neither of these subordinate names would be part of the entity name spec for `Set`, in which case the constructors `Bin` and `Tip` would not be in scope in the program; such is the case for all modules that *import* the `Set` module, since it exports only $[\nu_S]$`Set()`. We shall see more examples of how entity name specs factor into import and export resolution at the module level.

ENTITY DEFINITION SPECIFICATIONS The formalization distinguishes the semantic type of `member` as a value, typ_1, with the "type" of the def_1 definition itself; *i.e.*, the distinction between the type of the value and the type of the whole definition/binding of that value. We call these "types" of core entity bindings *entity definition specifications* (*dspcs*). In the case of value bindings, this specification trivially pairs the value name with its semantic type:

$$\mathit{dspc}_1 \quad \triangleq \quad \texttt{member} :: \mathit{typ}_1$$

The specifications of datatype bindings look similar: the syntactic types mentioned in the datatype binding are interpreted into semantic types, yielding a specification that resembles the original syntax but with types replaced. And similarly for class and instance bindings. The exact syntax of these specifications don't matter much; they need only record all the interpreted, semantic types for any syntactic types occurring in the entity bindings.

The *dspcs* for declarations look very similar to the *dspcs* for definitions, as the specs contain only type information and not value-level terms. One distinction, however, is the additional *dspc* form for abstract datatype declarations.

USE OF PHYSICAL NAMES IN TYPECHECKING One of the motivations presented earlier for tracking module identity was to distinguish between distinct types with the same syntac-

Interpretation with Entity Environments

Interpretation of single entity references $\boxed{\mathit{eenv}(\mathit{eref}) \;:_{\mathsf{par}}\; \mathit{phnm}}$ $\boxed{\mathit{eenv}(\mathit{eref}) = \mathit{phnm} : \mathit{espc}}$

$$\mathit{eenv}(\mathit{eref}) \overset{\mathsf{def}}{=} \mathit{phnm} \quad \text{if} \quad \begin{cases} \mathit{eref} \mapsto \mathit{phnm} \in \mathit{eenv} \\ \forall\, \mathit{eref}' \mapsto \mathit{phnm}' \in \mathit{eenv} :\; \mathit{eref} = \mathit{eref}' \Rightarrow \mathit{phnm} = \mathit{phnm}' \end{cases}$$

$$\mathit{eenv}(\mathit{eref}) = \mathit{phnm} : \mathit{espc} \overset{\mathsf{def}}{\Leftrightarrow} \begin{cases} \mathit{espc} \in \mathsf{locals}(\mathit{eenv}) \\ \mathit{phnm} = \mathit{eenv}(\mathit{eref}) \\ \mathit{phnm} \in \mathsf{allphnms}(\mathit{espc}) \end{cases}$$

Syntactic intepretation $\boxed{\mathit{eenv}(\mathit{utyp}) \;:_{\mathsf{par}}\; \mathit{typ}}$ $\boxed{\mathit{eenv}(\mathit{ucls}) \;:_{\mathsf{par}}\; \mathit{cls}}$ $\boxed{\mathit{eenv}(\mathit{kenv}\,.\,\mathit{ucls}) \;:_{\mathsf{par}}\; \mathit{head}}$

$$\begin{aligned} \mathit{eenv}(\mathsf{T}\ \overline{\mathit{utyp}}) &= \mathit{eenv}(\mathsf{T})\ \overline{\mathit{eenv}(\mathit{utyp})} \\ \mathit{eenv}(\texttt{forall}\ \mathit{kenv}\,.\,\mathit{utyp}) &= \texttt{forall}\ \mathit{kenv}\,.\,\mathit{eenv}(\mathit{utyp}) \\ \mathit{eenv}(\alpha) &= \alpha \end{aligned}$$

$$\begin{aligned} \mathit{eenv}(\mathsf{C}\ \overline{\mathit{utyp}}) &= \mathit{eenv}(\mathsf{C})\ \overline{\mathit{eenv}(\mathit{utyp})} \\ \mathit{eenv}(\mathit{kenv}.\mathit{ucls}) &= \mathit{kenv}.\mathit{eenv}(\mathit{ucls}) \end{aligned}$$

Figure 6.7: Definition of syntactic interpretation via entity environments. Note the following: (1) The interpretation of a single entity reference requires that the reference be *unambiguous*: the entity environment (*eenv*) cannot map the reference to more than one physical name (*phnm*). (2) The $\mathsf{allphnms}(\mathit{espc})$ operation gathers up all the physical names concerning a particular export spec and is defined in Appendix §A.1.2.1. And (3) throughout this dissertation, the notation `OPERATION : OBJECT` is used to mean the meta-level `OPERATION` is a total function producing `OBJECT`; when ":" is replaced with "$:_{\mathsf{par}}$" then the meta-level `OPERATION` is a *partial* function producing `OBJECT`.

tic name. An implementation observes this distinction by comparing physical names of types. As an example, consider the following core program containing two definitions.

```
data T = MkT Int

n :: Int
n = f (MkT 5)
```

The function `f` is presumed to have been imported from another module; then the given *eenv* maps `f` to a physical name $[\nu]$`f`, and the given Φ maps $[\nu]$`f` to an entity spec `f ::` $[\nu']$`T ->` $[\nu_P]$`Int`. The `T` mentioned in the type of `f` is not the locally defined `T`, which has physical name $[\nu_0]$`T`, but rather some other type `T`, $[\nu']$`T`. Because the domain type of `f` does not match the type of the expression `(MkT 5)`, the program is ill-typed.

OVERLAPPING ENTITY NAMES Among the entity bindings of a core program, none may bind any common names. Clearly, it wouldn't make sense to bind two different values both name `x`, for example. But the subordinate names must also be distinct. The following core programs are ill-typed because of overlapping names.

```
data T
data U = K | MkU
data V = K | MkV
```

```
class C a b where
  g :: a -> b

f :: Int -> Bool
f = ...

g :: String -> String
g = ...
```

In the first program, a list of declarations, the constructor name `K` is a subordinate name of both `U` and `V`. In the second program, a list of definitions, the class method `g`, a subordinate name of class `C`, overlaps with the value definition `g`.[7]

As we shall see shortly, rejecting programs with such overlap is not merely a design decision to cohere with Haskell semantics; it enforces a key invariant in the formalization. Because Backpack ascribes identity to *modules* and not to *entities*, the formalization critically relies on the uniqueness of entity names paired with modules. If a single module with identity ν could bind two distinct entities for name χ, then the physical name $[\nu]\chi$ would no longer uniquely identify one or the other. Concretely, in terms of the second example above, if one were to request from Φ the specification for the physical name $[\nu]$`g`, should the specification for (the method `g` in) the class `C` or the specification for the value `g` be returned? By following Haskell's rejection of such overlap, we avoid this question altogether.

AMBIGUOUS ENTITY REFERENCES In Haskell it's possible that a module has two distinct entities with the same name in scope. For example, consider the following core program, a list of declarations, that declares a fresh datatype named `Bool`, which overlaps with the entity of the same name imported from the `Prelude` module.

```
data Bool = True | False | Dunno
```

7 If the Backpack formalization had included *records* in datatype bindings, then record field names would present another potential form of overlap.

Consider the typing of this program with respect to an *eenv* that contains the following entries:

$$\textit{eenv}\text{ contains }\left\{\begin{array}{lcl} \texttt{Bool} & \mapsto & [\nu_P]\texttt{Bool} \\ \texttt{Prelude.Bool} & \mapsto & [\nu_P]\texttt{Bool} \\ \texttt{Bool} & \mapsto & [\nu_1]\texttt{Bool} \\ \texttt{Local.Bool} & \mapsto & [\nu_1]\texttt{Bool} \\ & \vdots & \end{array}\right\}$$

In this example, the environment specifies that the locally declared `Bool` is identified with provenance ν_1, which is distinct from ν_P. Curiously, *eenv* has multiple entries for the syntactic reference `Bool`—but single entries for the syntactic references `Prelude.Bool` and `Local.Bool`—and therefore *cannot interpret* references to the unqualified name `Bool` into a physical name.

Despite this curiosity, the core program is still well-typed, in order to cohere with Haskell 98. That's because in this program there aren't any syntactic references to `Bool`; there is merely a declaration for `Bool` without any *reference* to `Bool`. It's quite unrealistic to define a datatype but no other entities that mention it, so consider a modified program and *eenv*:

```
data Bool = True | False | Dunno
not :: Bool -> Bool
not b = ... b ...
```

$$\textit{eenv}\text{ contains }\left\{\begin{array}{lcl} & \vdots & \\ \texttt{not} & \mapsto & [\nu_P]\texttt{not} \\ \texttt{Prelude.not} & \mapsto & [\nu_P]\texttt{not} \\ \texttt{not} & \mapsto & [\nu_2]\texttt{not} \\ \texttt{Local.not} & \mapsto & [\nu_2]\texttt{not} \end{array}\right\}$$

Because there are *references* to `Bool` in the type of `not`, and because *eenv* doesn't uniquely map `Bool` to a physical name, this program is ill-typed for having an ambiguous reference. We can recover well-typedness by eliminating the ambiguity with *qualified references*:

```
data Bool = True | False | Dunno
not :: Local.Bool -> Local.Bool
not b = ... b ...
```

In this final example, with the same *eenv* as the previous one, there are no longer any ambiguous references. This core program is well-typed. This would be a pain in practice; indeed, Haskell provides an alternative form of import statement that *hides* unwanted entities from the imported module. In Haskell, the above example might be written (with an explicit import of the Prelude module) as:

```
import Prelude hiding(Bool, not)
data Bool = True | False | Dunno
not :: Bool -> Bool
not b = ... b ...
```

Backpack omits this construct to simplify the presentation. It would pose no complication to the formalization.

Finally, it's important to note *in which* of the Backpack typing judgments such ambiguous references would be rejected. Because the presence of ambiguous references depends on the *eenv and* the actual core bindings, such programs are rejected here in the core level, not in the module level where the *eenv* is synthesized.

OVERLAPPING INSTANCES In §4.4 we saw that Backpack follows Haskell 98's rejection of overlapping type class instances. However, overlapping instances are rejected at the module level when the world ω is constructed, not in the core typing judgment. We return to this point in the next chapter.

6.4 TYPING JUDGMENTS

With the syntax and semantic objects in mind, we now return to the core typing judgments:

1. $\Phi; \nu_0; \mathit{eenv}; \omega \vdash \mathit{defs} : \mathit{dspcs}$ and
2. $\Phi; \overline{\nu}; \mathit{eenv}; \omega \vdash \mathit{decls} : \mathit{dspcs}$.

The judgments state, respectively, the following: in the module context Φ, with the core environment *eenv*, in the world ω,

1. the definitions *defs* for the program with identity ν_0 are well-formed with respective specifications *dspcs*; and
2. the declarations *decls* with respective provenances $\overline{\nu}$ are well-formed with respective specifications *dspcs*.

In other words, under the given assumptions about the module context and about the names and identities, the bindings are all well-typed and well-kinded.

As already discussed, we axiomatize over these definitions, rather than define them, in order to avoid the particulars of the Haskell programming language. (In the next subsection I formally define some axioms to describe their expected behavior.) We therefore have no inductive definitions for these judgments. A natural question to ask, then, is what necessitates each component of the judgments; *e.g.*, why does each judgment include Φ and *eenv*? The answer is that *any faithful definition* of these judgments—*i.e.*, any implementation of core typechecking—would require at least these components. For example, a core typechecker must know the specifications of imported entities (in Φ) and how to map syntactic names to their physical names (via *eenv*).

6.4.1 *Examples of Core Typing*

In this section we walk through some examples of well- (and ill-) typed core bindings and how the core typing judgments would accept (or reject) them. Figure 6.2 and Figure 6.4 provide the main examples for this section; the former a mock implementation of `Data.Set` (a list of definitions) and the latter a mock specification of `Data.Array.IArray` (a list of declarations).

We describe, in turn, the typing of each kind of core binding, distinguishing between *definitions* (in modules) and *declarations* (in signatures) where relevant. And though the Backpack formalization abstracts over these core-level judgments, we still relate them as tightly as possible to the rest of Backpack's judgments, revealing how the latter shine some light on the undefined former.

TYPECHECKING VALUE BINDINGS Value *definitions* contain actual Haskell expressions to typecheck, whereas value *declarations* contain only syntactic types to interpret. We look first at the former.

In the definition of the value `member` in Figure 6.2, we see both the annotated syntactic type of this value, $\mathit{utyp}_{\mathsf{m}}$, and the Haskell expression the entity is bound to, $\mathit{uexp}_{\mathsf{m}}$, and as discussed before the interpretation of that type, $\mathit{typ}_{\mathsf{m}}$, all reproduced below:

$\mathit{utyp}_{\mathsf{m}}$ = `forall (a :: *) . Ord a => a -> Set a -> Bool`

$\mathit{uexp}_{\mathsf{m}}$ = `\ x s -> case s of ...`

$\mathit{typ}_{\mathsf{m}}$ = $\mathtt{forall}\ (\alpha :: *) .\ [\nu_{\mathsf{P}}]\mathtt{Ord}\ \alpha \Rightarrow \alpha \rightarrow [\nu_{\mathsf{S}}]\mathtt{Set}\ \alpha \rightarrow [\nu_{\mathsf{P}}]\mathtt{Bool}$

As would be expected, the well-formedness of the definition would involve first synthesizing a type for $\mathit{uexp}_{\mathtt{m}}$; second, checking that it conforms to the type $\mathit{typ}_{\mathtt{m}}$; and third, ensuring that $\mathit{typ}_{\mathtt{m}}$ itself is well-formed and has kind $*$ (*i.e.*, that it is the type of terms). Backpack abstracts over the details of the first and second parts, providing enough "semantic ingredients" for someone else to implement, but it does offer the means to mechanically verify the third. Next we describe how Backpack's axiomatized core typing judgment facilitates these three steps.

First, to adequately abstract over the details of typechecking, the core typing judgment $\Phi; \nu_0; \mathit{eenv}; \omega \vdash \mathit{defs} : \mathit{dspcs}$ must provide enough information to *implement* that typechecking. Indeed it does. In order to typecheck $\mathit{uexp}_{\mathtt{m}}$, a typechecker implementation would necessarily consult

- *eenv* for the physical names of all syntactic entity references like `Ord`, `Set`, and `member`;
- Φ for the corresponding *dspc*s of those physical names, yielding their types, kinds, etc;[8] and
- ω for the visible type class instances that may be applied to class methods like `compare`,

all provided to it, in a sense, by the Backpack formalization.

Second, to check that the synthesized type conforms to the annotated type, an implementation would consult some kind of order on (semantic) types. But Backpack does not define any notion of subtyping, conversion, or even judgmental equality on semantic types. Instead, two semantic types are "equivalent" if and only if they are *syntactically equivalent* (modulo alpha conversion of bound type variables). An implementation, however, would support a "generalization" order on types, as defined in the Haskell spec,[9] allowing for the well-typedness of a definition like

```
intEq :: Int -> Int
intEq = \ x y -> x == y
```

Third, to determine whether a type is well-formed, an implementation would consult Backpack's *kinding judgment* on semantic types, written $\Phi; \mathit{kenv} \vdash \mathit{typ} : \mathit{knd}$.[10] Further explanation—and definition—of this rather conventional judgment will be presented in §6.5.1. For now, however, consider how the judgment applies to the `member` example. A derivation of $\Phi^{\mathsf{L}}_{\mathsf{S}}; \cdot \vdash \mathit{typ}_{\mathtt{m}} : *$ must be constructed in order to satisfy this third requirement of typechecking the `member` definition. But there's some sleight of hand in this application of the judgment. The module context $\Phi^{\mathsf{L}}_{\mathsf{S}}$ is *not* the same as Φ_{S}, the module context from the typing judgment for the whole set of definitions $\mathit{defs}_{\mathsf{S}}$. (That application of the core typing judgment is $\Phi_{\mathsf{S}}; \nu_{\mathsf{S}}; \mathit{eenv}_{\mathsf{S}}; \omega_{\mathsf{S}} \vdash \ldots, (\mathtt{member} :: \mathit{utyp}_{\mathtt{m}} = \mathit{uexp}_{\mathtt{m}}), \ldots : \mathit{dspcs}_{\mathsf{S}}$.) Why not? The ambient module context Φ_{S} lacks information about the local module ν_{S} and its defined entities, like $[\nu_{\mathsf{S}}]$`Set`, whereas the *localized* module context $\Phi^{\mathsf{L}}_{\mathsf{S}}$ contains the ambient context as well as these locally defined entities. We shall return to this idea of localization, and how to derive $\Phi^{\mathsf{L}}_{\mathsf{S}}$, later.

Typechecking a value definition comprises the three steps above, with Backpack only defining the means to perform the third step. Typechecking a value *declaration*, however, can be described entirely within Backpack, as it only requires that third step.

Consider the declaration of `insert` in Figure 6.3. The annotated type of this value, $\mathit{utyp}_{\mathtt{i}}$, is part of the declaration. And in the corresponding entity environment $\mathit{eenv}_{\mathsf{S}}$, that syntactic type is interpreted into the semantic type $\mathit{typ}_{\mathtt{i}}$. Together,

$$\begin{aligned} \mathit{utyp}_{\mathtt{i}} &= \mathtt{forall\ (a\ ::\ *)\ .\ Ord\ a\ =>\ a\ ->\ Set\ a\ ->\ Bool} \\ \mathit{typ}_{\mathtt{i}} &= \mathtt{forall}\ (\alpha :: *).\ [\nu_{\mathsf{P}}]\mathtt{Ord}\ \alpha \Rightarrow \alpha \rightarrow [\beta_{\mathsf{S}}]\mathtt{Set}\ \alpha \rightarrow [\nu_{\mathsf{P}}]\mathtt{Bool} \end{aligned}$$

8 Technically, Φ does not contain *dspc*s of anything defined locally within *defs*. We shall return to this point shortly.
9 Peyton Jones (2003), "Haskell 98 Language and Libraries: the Revised Report," §4.1.4.
10 This judgment exists in Backpack in order to define well-formedness of *module* types τ, but here it would be repurposed as part of core typechecking.

Then the well-formedness of the semantic type amounts to constructing the derivation $\Phi^{\mathsf{L}}_{\mathsf{S}}; \cdot \vdash typ_{\mathtt{i}} : *$, just as in the previous case of a value definition. (Once again, the module context for the kinding judgment is the localized context $\Phi^{\mathsf{L}}_{\mathsf{S}}$, not the ambient context Φ_{S}.)

TYPECHECKING DATATYPE BINDINGS Datatype bindings take two forms: concrete (with constructors) and abstract (without constructors), with the latter syntax only available for declarations, not definitions. Typechecking abstract datatype bindings, like `Set` in Figure 6.3, is a no-op.

Typechecking of concrete datatype bindings, on the other hand, resembles that of value declarations. Like the syntactic types in value declarations, each syntactic type appearing in concrete datatype bindings must be interpretable and well-formed with kind $*$ in the specified kind environment (*kenv*).

Consider the definition of `Set` in Figure 6.2, a binary tree. Each of the two constructors must have type fields that are indeed well-formed types. The `Tip` case is trivial, but the `Bin` case specifies three syntactic type fields: `a` (the value at this node), `Set a` (the left subtree), and `Set a` (the right subtree). First, these types are interpreted by the corresponding *eenv* into semantic types α (type variables are trivially interpreted in any entity environment), $[\nu_{\mathsf{S}}]\mathtt{Set}\ \alpha$, and $[\nu_{\mathsf{S}}]\mathtt{Set}\ \alpha$. Next, these semantic types have kind $*$ according to the kinding judgment *in the kind environment* $\alpha :: *$.[11] Indeed, we can prove

- $\Phi^{\mathsf{L}}_{\mathsf{S}}; (\alpha :: *) \vdash \alpha : *$ and
- $\Phi^{\mathsf{L}}_{\mathsf{S}}; (\alpha :: *) \vdash ([\nu_{\mathsf{S}}]\mathtt{Set}\ \alpha) : *$

in the corresponding *localized* module context $\Phi^{\mathsf{L}}_{\mathsf{S}}$.

TYPECHECKING CLASS BINDINGS Class bindings are syntactically identical between declaration and definition and, like datatype bindings and value declarations, contain no Haskell expressions.[12] As with the other bindings, all the syntactic types occurring in class bindings must be well-formed, in respective kind environments, and the types of class methods must have kind $*$.

Class bindings include superclass contexts, syntactically expressed as class constraints (*ucls*), which must also be well-formed in the kind environment annotating the class binding. That necessitates another judgment in the core level for *class constraint* well-formedness, written $\Phi; kenv \vdash cls$ wf, analogous to that for *type* well-formedness. The definition for this judgment, along with the other judgment, is described in §6.5.1.

As an example of checking class constraints, consider the `Prelude`'s definition of the `Ord` class, which has `Eq` as a superclass:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
```

This Haskell-like syntax translates into Backpack syntax (with kind inference for the type variable `a`) as

$$\mathtt{class\ Ord}\ (\alpha :: *) \Leftarrow \mathtt{Eq}\ \alpha\ \mathtt{where} \ldots,$$

making explicit the kind environment $(\alpha :: *)$ that binds the type variable α over the whole definition. Then the superclass constraint, interpreted as $[\nu_{\mathsf{P}}]\mathtt{Eq}\ \alpha$, must be well-formed in the localized module context $\Phi^{\mathsf{L}}_{\mathsf{P}}$ (containing all the entities defined locally in the `Prelude` module): $\Phi^{\mathsf{L}}_{\mathsf{P}}; (\alpha :: *) \vdash ([\nu_{\mathsf{P}}]\mathtt{Eq}\ \alpha)$ wf.

11 The reader might wonder where the kind of `a` was specified in the `Set` binding. Haskell 98 defines a “kind inference” procedure to synthesize kinds of type variables like `a`; for the sake of presentation we omit this synthesis but assume its existence.

12 As discussed earlier, unlike Haskell 98, Backpack does not support default method implementations in class bindings.

How might a class constraint be ill-formed? Suppose we modify the `IArray` class from Figure 6.4 to (erroneously) add a superclass constraint for `Eq a`:

```
class IArray (a :: * -> * -> *, i :: *, e :: *) <= Eq a where ...
```

The intention is for the type variable α to come with equality, but actually the type of arrays within `IArray` is `a i e`. The difference? The type variable `a` has kind `* -> * -> *`, not `Eq`'s expected kind `*`; therefore the statement

$$\Phi^{\mathsf{L}}_{\mathsf{A}};\ (\alpha :: * \texttt{->} * \texttt{->} *,\ i :: *,\ e :: *) \vdash ([\nu_{\mathsf{P}}]\mathtt{Eq}\ \alpha)\ \mathsf{wf}$$

cannot be proved, making this hypothetical class binding ill-formed.

On top of type well-formedness, class bindings (along with instance bindings) should satisfy additional validity checks—not for any technical reason for Backpack's metatheory, but for reasons specific to type class mechanics. These checks ensure crucial properties of type classes, what Sulzmann[13] concisely defines as "Haskell type class conditions." Since they are irrelevant for Backpack's study of *modularity* in Haskell, and since core-level typing is abstracted out of Backpack anyway, we can safely omit them entirely from the formalization.

TYPECHECKING INSTANCE BINDINGS The most involved sort of binding to check, instance definitions, necessitate three key checks: typechecking expressions (*i.e.*, Haskell code), checking the well-formedness of the instance head, and checking auxiliary invariants, as mentioned in the case of class bindings, purely for the soundness of properties specific to type classes. And since instance *declarations* contain no code, only the latter two checks are required.

Of course, instance uniqueness would also need to be checked, as we wouldn't want to accept an instance binding if an overlapping instance was imported from elsewhere. But this particular requirement of well-formed instances isn't part of core typing. That's because this check intersects with Backpack's concern with *modularity*; as Chapter 4 showed, the world consistency property is key. Instead, the world ω already contains all the instances known in this scope, including any locally declared instances, and this world is consistent by construction (*i.e.*, the implicit invariant $\mathsf{consistent}(\omega)$ that is maintained by all worlds). In short, when the module level appeals to this judgment on the core level, it has already done the work of proving world consistency, *i.e.*, global uniqueness of instances.

With that aside, now back to typechecking instance bindings. Consider the instance definition for equality on the `Set` type, from Figure 6.2 and reproduced below.

```
instance (a :: *) Eq a => Eq (Set a) where (==) = uexp==
```

The typechecking of the Haskell expression for `(==)` must be done in a context that knows not only the other instances among the `Set` definitions, but also the hypothesis `Eq a`. Again, Backpack abstracts over the details of this core-level typechecking and type class requirements.

Finally, the relevant part for Backpack: the well-formedness of the instance head must be checked. This amounts to proving the well-formedness of `Eq (Set a)` in the localized module context for `Set`, $\Phi^{\mathsf{L}}_{\mathsf{S}}$:

$$\Phi^{\mathsf{L}}_{\mathsf{S}};\ (\alpha :: *) \vdash [\nu_{\mathsf{P}}]\mathtt{Eq}\ ([\nu_{\mathsf{S}}]\mathtt{Set}\ \alpha)\ \mathsf{wf}$$

Since the type $[\nu_{\mathsf{S}}]\mathtt{Set}\ \alpha$ is well-formed with kind $*$, this statement would indeed be derivable.

13 Sulzmann (2006), "Extracting Programs from Type Class Proofs".

Auxiliary Core Judgments: Type and Class Constraints

Semantic type well-formedness ("kinding") $\boxed{\Phi; \mathit{kenv} \vdash \mathit{typ} : \mathit{knd}}$

$$\frac{(\alpha :: \mathit{knd}) \in \mathit{kenv}}{\Phi; \mathit{kenv} \vdash \alpha : \mathit{knd}}\ (\textsc{KndVar})$$

$$\frac{\mathsf{dom}(\mathit{kenv}) \mathbin{\#} \mathsf{dom}(\mathit{kenv}') \qquad \overline{\Phi; \mathit{kenv}, \mathit{kenv}' \vdash \mathit{cls}\ \mathsf{wf}} \qquad \Phi; \mathit{kenv}, \mathit{kenv}' \vdash \mathit{typ} : *}{\Phi; \mathit{kenv} \vdash (\texttt{forall}\ \mathit{kenv}'.\ \overline{\mathit{cls}} \Rightarrow \mathit{typ}) : *}\ (\textsc{KndAll})$$

$$\frac{\Phi(\nu)(\mathsf{T}) = \texttt{data}\ \mathsf{T}\ (\overline{\alpha :: \mathit{knd}}, \overline{\alpha' :: \mathit{knd}'})\ \ldots \qquad \overline{\Phi; \mathit{kenv} \vdash \mathit{typ} : \mathit{knd}}}{\Phi; \mathit{kenv} \vdash ([\nu]\mathsf{T}\ \overline{\mathit{typ}}) : \overline{\mathit{knd}'}\ \texttt{->}\ *}\ (\textsc{KndCtor})$$

Semantic class constraint well-formedness $\boxed{\Phi; \mathit{kenv} \vdash \mathit{cls}\ \mathsf{wf}}$

$$\frac{\Phi(\nu)(\mathsf{C}) = \texttt{class}\ \mathsf{C}\ (\overline{\alpha :: \mathit{knd}})\ \ldots \qquad \overline{\Phi; \mathit{kenv} \vdash \mathit{typ} : \mathit{knd}}}{\Phi; \mathit{kenv} \vdash ([\nu]\mathsf{C}\ \overline{\mathit{typ}})\ \mathsf{wf}}\ (\textsc{ClsWf})$$

Figure 6.8: Definition of auxiliary core well-formedness judgments for semantic types ("kinding") and for semantic class constraints. The notation $\Phi(\nu)(\chi) = \mathit{dspc}$ is used to mean that $\nu \in \Phi$, and *dspc* has name χ in ν's type; this mapping of names to specs is well-defined since a module type's *dspcs* have non-overlapping names.

6.5 METATHEORY OF THE CORE LEVEL

Most of the definition of the semantics of the core level of Backpack is abstracted away as undefined typing judgments. These judgments are nevertheless a key part of Backpack; the semantics of the outer levels depend in turn on their semantics. For that reason these judgments should satisfy particular metatheoretic properties required by the outer levels.

In this section I describe those metatheoretic properties of the core-level typing judgments as *axioms* that Backpack presumes to hold. The proofs of the necessary properties in the outer levels therefore rest on these axioms. But before detailing the axioms I present some well-formedness judgments that characterize contextually valid semantic objects at the core level.

6.5.1 *Well-formedness of semantic objects*

In the last section we described examples of typechecking core programs to illustrate how one might define core typing, *i.e.*, how core typing interacts with the mechanics of the Backpack formalization like module identities and entity specifications. In many examples above, the well-typedness of core programs depended upon the well-formedness (a.k.a. "kinding") of semantic types (*typ*) and class constraints (*cls*). To further illustrate how core typing interacts with the formalization, we provide, in Figure 6.8 definitions for these two well-formedness judgments.

The judgments are largely straightforward and conventional. They are contextualized by both a module context Φ and a kind environment *kenv*, the same environment occurs as part of syntactic types (*utyp*). We briefly describe the rules that define these judgments; of note are rules (KndCtor) and (ClsWf), since they illustrate how core typing depends upon the mechanics of the Backpack formalization.

- Rule (KndVar) defines the well-formedness of a type variable α. It is trivially well-formed, with kind *knd*, if the environment describes it as such.
- Rule (KndAll) defines the well-formedness of a polymorphic (qualified) type `forall` $kenv'$. $\overline{cls}$ => *typ*, which always has kind $*$. The first premise states that the variables bound in the ambient kind environment *kenv* does not overlap with those bound in $kenv'$. The second premise states that the class constraints $\overline{cls}$ are well-formed in the extended kind environment. And the third premise states that the constituent type *typ* has kind $*$.
- Rule (KndCtor) defines the well-formedness of a (possibly partial) application of a type constructor, $[\nu]\mathrm{T}\ \overline{typ}$. The kind of such a type is $*$ if all the "type arguments" are applied; otherwise, for a partial application, the type has some other, higher kind. For example, the type $[\nu_S]$`Set` (with no types applied), has kind $*$ -> $*$. The first premise of this rule states that the type constructor $[\nu]\mathrm{T}$ is defined in the module context Φ with a kind environment that binds the variables $\overline{\alpha}, \overline{\alpha'}$ with kinds $\overline{knd}, \overline{knd'}$; the former variables (and kinds) correspond to the type arguments $\overline{typ}$. Moreover, the ellipsis indicates that the specification for this type may either be an abstract or concrete one. The second premise states that the type arguments $\overline{typ}$ have kinds $\overline{knd}$, leaving the rest of the kinds $\overline{knd'}$ as part of the higher kind of the whole type, $\overline{knd'}$ -> $*$.
- Rule (ClsWf) defines the well-formedness of a class constraint $[\nu]\mathrm{C}\ \overline{typ}$. The first premise, once again, states that the class $[\nu]\mathrm{C}$ is defined in the module context Φ with a kind environment that binds the variables $\overline{\alpha}$ with kinds $\overline{knd}$. The second premise merely states that the types $\overline{typ}$ have those kinds.

6.5.2 *Axioms of core typing*

As already hammered home, most of the semantics of Backpack's core level—the formal model of Haskell's core language of types and terms (and classes and instances)—are abstracted out of this formalization. Now it's time to be a bit more precise about what that means.

The two core-level typing judgments,

$$\Phi;\ \nu_0;\ eenv;\ \omega\ \vdash\ defs\ :\ dspcs \text{ and}$$
$$\Phi;\ \overline{\nu};\ eenv;\ \omega\ \vdash\ decls\ :\ dspcs,$$

are treated as *parameters* to Backpack. What remains is to specify what's required of them. For that reason we assert some axioms about their expected behavior, axioms that are directly required in the proofs of desired properties in (the outer levels of) Backpack. Four of the five axioms fall into two categories—*regularity* and *soundness*—with one for each category and judgment.[14]

- **Regularity**: The regularity axioms simply assert some basic properties expected to hold of the typing judgments. These are the kinds of properties that would be immediate by inverting an inference rule whose conclusion is known. For example, regularity of both core typing judgments asserts that the number of bindings is equal to the number of *dspcs*, and that any `instance` bindings' syntactic heads can be interpreted by *eenv* into semantic heads that are matched by those bindings' corresponding specs in *dspcs*. Full axiom statements can be found in the appendix: Axiom A.13 for definition typing and Axiom A.14 for declaration typing.

14 These axioms differ considerably from their original definitions as presented in (Kilpatrick *et al.*, 2013). In the original work, only the *internal language* (IL) part of the formalization was parameterized by core typing on definitions, while the *external language* (EL) part actually defined core typing via elaboration into that parameterized IL judgment. Consequently, the corresponding soundness axiom was defined as a corollary to soundness on the IL judgment, as "Soundness of module definition checking in image of translation" (Corollary A.24). The definitions in this thesis have no such connection to elaboration and are much cleaner.

- **Soundness**: The soundness axioms are more interesting. They both assert, in some way, the well-formedness of the *dspcs* that classify core bindings. But instead of appealing to core-level well-formedness, of the *dspcs* object itself, it asserts *module*-level well-formedness, of module types and physical module contexts. We'll see those properties in more detail in the next chapter, on the module level. Full axiom statements can be found in the appendix: Axiom A.15 for definition typing and Axiom A.16 for declaration typing.

The axioms have been designed according to the proofs for which they're required in the rest of Backpack. Soundness axioms in particular have quite a bespoke set of premises and conclusion. On one hand, this over-fitting to the proofs is undesireable, but on the other hand, it's sufficient for our purposes. This thesis is simply not concerned with the semantics of Haskell's core level.

7 MODULE LEVEL

The previous level of Backpack, the core level, mostly introduced new notation to represent existing Haskell things. We look now at the next level out, the first one to include some significant new concepts introduced by this thesis: the module level.

Syntactically, Backpack introduces *signatures* to the module level of Haskell, and semantically, it introduces *shaping*. In comparison to Haskell 98, these two concepts quadruple the number of typing judgments needed to capture the semantics of a module-level program: from one in Haskell 98 (well-typing of a Haskell 98 module) to four in Backpack (well-shaping/-typing of a Backpack module/signature). To manage and mitigate this complexity, the judgments are defined with explicitly parallel structure and semantic objects: module variants mirror signature variants, and typing variants mirror shaping variants.

$$\begin{array}{r|l|l} & \textbf{\textit{modules}} & \textbf{\textit{signatures}} \\ \textbf{\textit{shaping}} & \widehat{\Gamma}; \nu_0 \Vdash M \Rightarrow \widehat{\tau} @ \widehat{\omega} & \widehat{\Gamma}; \overline{\nu} \Vdash S \Rightarrow \widehat{\sigma} @ \widehat{\omega} \mid \widehat{\Phi}_{\mathsf{sig}} \\ \textbf{\textit{typing}} & \Gamma; \nu_0 \vdash M : \tau @ \omega & \Gamma; \rho \vdash S : \sigma @ \omega \mid \Phi_{\mathsf{sig}} \end{array}$$

What's the goal of the module level? Notably, the goal is *not* to perform any unification or linking. (That's one of the goals of the package level.) In the *shaping* pass, the goal is to determine the entities exported by the module and their provenances, along with the world that the module inhabits. In the *typing* pass, the goal is to perform typechecking via the core level.

INPUTS (FROM PACKAGE LEVEL) From the package level, the module level uses, first, a *module context* (Γ) which has two constituent functions: it maps module names to module identities ($\mathcal{L}$, new in the module level) and it maps module identities to types (Φ, as used in the core level); and second either a single module identity to "name" the present module or, in the case of signatures, a *realizer* mapping entity names to module identities (ρ). All such inputs in the module level come in two flavors: the full versions as already stated, and the "shapey" versions, denoted with a $\widehat{\cdot}$, that contain names and module identities but lack any *core-level* types (*typ*).

OUTPUTS (TO PACKAGE LEVEL) The module level produces module types (τ) which describe entities exposed by a module, along with their specifications (*dspcs*)—as were determined by the core level—and the world they inhabit (ω). These will be used by the package level for linking and for setting up subsequent module contexts (Γ).

As an additional output for signatures, a *signature context* (Φ_{sig}) contains the specifications of the entities declared in the signature. It's a technical device to enable Backpack's definition of module linking in terms of merging package types, as we shall see in the next chapter.

INTERNALS The module level primarily uses its input objects to determine the core environment (*eenv*) that represents the scope of the module/signature and the world (ω) that it inhabits; this process is called *import resolution*. With these two objects, the core-level typing then determines the entity specifications (*dspcs*) of the entities defined/declared in the module/signature. That's only for the typing judgments, as the shaping judgments do not employ core-level typing. Indeed, the theme of shaping is that no core-level typechecking is involved. Finally, the set of names exposed by the module/signature (*espcs*) is determined by a process called *export resolution*.

Backpack Module Syntax

Logical Module Names	ℓ	$\in$	*ModNames*	
Module Expressions	M	::=	*impdecls*`;` *expdecl*`;` *defs*	
Signature Expressions	S	::=	*impdecls*`;` *decls*	
Imported Entity Specs	*import*	::=	χ	Entity Only
		|	χ`(`$\overline{\chi}$`)`	Entity & Some Subordinates
		|	χ`(..)`	Entity & All Subordinates
Imported Module Specs	*impspec*	::=	$\cdot$	All Entities
		|	`(`$\overline{import}$`)`	Some Entities
Import Declarations	*impdecl*	::=	`import` ℓ [`as` ℓ] *impspec*	Unqualified Import Declaration
		|	`import qualified` ℓ [`as` ℓ] *impspec*	Qualified Import Declaration
List of Import Declarations	*impdecls*	::=	$\overline{impdecl}$	List of Imports
Exported Entity Specs	*export*	::=	*eref*	Entity Only
		|	*eref*`(`$\overline{\chi}$`)`	Entity & Some Subordinates
		|	*eref*`(..)`	Entity & All Subordinates
		|	`scope` *mref*	Entities in Namespace
Export Statements	*expdecl*	::=	$\cdot$	Implicit Exports
		|	`export` `(`$\overline{export}$`)`	Explicit Exports

Figure 7.1: Syntax of Backpack's module level.

DETAILS, DETAILS Much of the formalization of Backpack's module level consists of boring technical details. These details have been designed to balance, on one hand, modeling Haskell 98's import and export resolution and, on the other hand, integrating them with the rest of the formalization, the metatheory in particular.

Backpack is the first formalization of Haskell 98's module system that treats it as a (more or less) conventional type system with attendant metatheory. As such, the semantics of import and export resolution must be defined in such a way that resulting semantic objects must be judged well-formed with respect to some context, yielding a key metatheoretic result, for example, that "import statements within sensible module contexts produce sensible results" Lemma A.17. Such lemmas are composed together to form proofs of the larger desired metatheory Theorem A.1 (§7.7.3). To support these lemmas, the formalization of the module level defines additional semantic objects and judgments that are entirely separate from the main four typing judgments.

7.1 SYNTAX OF THE MODULE LEVEL

The syntax of Backpack's module level is presented in Figure 7.1. The key syntactic forms are M, module expressions, and S, signature expressions.

Similarity between Backpack module expressions and Haskell modules (as defined in Haskell 98) is obvious and intentional. However, some deviations do exist:

- Modules in Backpack do not syntactically name themselves, and thus the syntax of module expressions M excludes a "self" name ℓ. As discussed in the core level, the module reference for an entity defined/declared locally in some module/signature expression is written as the new keyword `Local`.
- Module export lists[1] have a different syntax, as `export` statements.

1 Peyton Jones (2003), "Haskell 98 Language and Libraries: the Revised Report," §5.2.

- The keyword `scope` for an exported entity spec replaces Haskell's keyword `module`, given that the latter clashes too much with Backpack concepts. The semantics of this keyword is indeed subtly different in Backpack than in Haskell; see the description of (ExpModAll) in §7.5.
- Import "hiding" declarations are excluded from Backpack in order to simplify the presentation. They would be straightforward to include.

Signature expressions (S), on the other hand, are not based on any syntactic form in Haskell 98. (Instead, signatures are based on GHC Haskell's "boot files" mechanism for recursive modules.) They resemble module expressions with a few important differences:

- Signature expressions contain core *declarations*, not core *definitions*. Put simply, they don't contain any executable code to typecheck.
- Signature expressions have no `export` statements. The only entities exported by signatures are those that are *declared* in the signature—with the exception of type class instances, which flow into and out of signatures just as they do in modules.

7.2 SEMANTICS OF THE MODULE LEVEL

We have already seen the four main judgments for the module level, of which the two *typing* judgments are the principal concerns for the Backpack type system. But behind the typing judgments lie two prerequisite concerns:

CORE ENVIRONMENT CONSTRUCTION What are the entities within the scope of a module/signature?

EXPORT RESOLUTION What are the entities exported by a module/signature?

Both concerns are obviously relevant to the module system of Haskell, and here they have a particular relevance. In Backpack the shaping pass of the module level must produce the names and identities of core-level entities exported by the module. Those names and identities are used to perform unification when modules are linked into signatures. As we saw in the chapter on the core level (Chapter 6), core environments (*eenv*) perform the work of translating syntactic entities into semantic ones, *i.e.*, with module identities. Thus the module level must construct the core environments in the first place in order to facilitate linking. Moreover, core environment construction involves import resolution, which in turn requires that *export* resolution was performed on the imported modules. The processes that attend to both concerns therefore enable the critical mechanics of Backpack.

Core environment construction and export resolution constitute much of the boring details of Backpack's module level. Compounding that boringness is the fact that both shaping and typing judgments depend on these two mechanisms, with slight variations that require a duplication of their definitions, necessitating "typey" versions and "shapey" versions. Whereas the distinction between these two variants generally has revolved around the presence or absence of core types (*typ*), at the module level the distinction revolves around the presence of *worlds* (ω) as opposed to *world shapes* ($\widehat{\omega}$).

7.3 SEMANTIC OBJECTS

MODULE IDENTITIES Module identities are a key technical concept in Backpack. They've already been presented in §3.1 with as much detail as is needed to understand the module-level formalization. For more technical discussion of module identities, see Appendix §A.1.1.

MODULE TYPES Modules types (τ) model, as a key insight in the design of Backpack goes, the *binary interface files* (`.hi` files) of GHC. They classify modules by serving as specifications of a module's contents. That specification is threefold.

Backpack Module Semantic Objects

Entity Environments	$eenv$	::=	$\{\overline{eref \mapsto phnm}\}; espcs$	
Module Types	τ	::=	$\langle\!\langle \overline{dspc}\,; espcs\,; \overline{\nu} \rangle\!\rangle$	
Module Shapes	$\hat{\tau}$	::=	$\langle\, \overline{\hat{dspc}}\,; espcs\,; \overline{\nu} \,\rangle$	
Entity Definition Shape Specs	$\hat{dspc}$	::=	$\mathsf{T} \mid \mathsf{T}(\overline{\mathsf{K}}) \mid \mathsf{x} \mid \mathsf{C}(\overline{\mathsf{x}}) \mid \mathtt{instance}$	
Module Worlds	ω	::=	$\{\!\{\overline{head \mapsto \nu}\}\!\}^{\dagger}$	
Module Shape Worlds	$\hat{\omega}$	::=	$\{\overline{head \mapsto \nu}\}$	
Signature Realizers	ρ	::=	$\hat{\tau} @ \hat{\omega}$	
Physical Module Contexts	Φ	::=	$\{\!\{\overline{\nu{:}\tau^{m} @ \omega}\}\!\}$	
Physical Module Shape Contexts	$\hat{\Phi}$	::=	$\{\overline{\nu{:}\hat{\tau}^{m} @ \hat{\omega}}\}$	
Logical Module Contexts	$\mathcal{L}$	::=	$\overline{\ell \mapsto \nu}$	
Module Contexts	Γ	::=	$(\!\mid \Phi\,; \mathcal{L} \mid\!)$	
Module Shape Contexts	$\hat{\Gamma}$	::=	$(\,\hat{\Phi}\,; \mathcal{L}\,)$	
Identified Bindings	$ibnds$	::=	$(\overline{\nu} \mid bnds)$	
Realizable Bindings	$rbnds$	::=	$(\rho \mid decls)$	Signature Declarations to be Identified
		|	$ibnds$	Already Identified Bindings
Augmented Entity Envs	$aenv$	::=	$aenv^{+} \mid aenv^{-}$	
Augmented Mod Entity Envs	$aenv^{+}$	::=	$(\hat{\Phi}; \nu_0; defs)$	
Augmented Sig Entity Envs	$aenv^{-}$	::=	$(\hat{\Phi}; \rho; decls)$	

Figure 7.2: Semantic objects relevant to Backpack's module level, with key objects in bold. † refers to the fact that, although worlds have the presented syntax, they additionally carry the invariant consistent(−) (Figure 7.5).

- First, there are the core-level entity specifications (*dspcs*), first described in §6.3. These semantic objects, as discussed in the last chapter, capture all the static information about the values, types, and type classes *originally bound in* a module. Notably, they do not contain specifications of entities that are merely *exported by* the module. This part of the module type designates the definitive location of those specifications, for use in core-level type-checking.

  We implicitly consider a vector of specifications (*dspcs*) to declare non-overlapping entity names ($\overline{\chi}$).[2]

- The second component of a module type are the entity *name* specifications (*espcs*), also described in §6.3. This part of the module type serves two roles: designating the exposed entities available to importing modules, and signaling the module identity provenances to *unify* during linking. The first role is why *espcs* in module types—often written as $espcs \in \hat{\tau}$—appear all over the definitions of import resolution (Figure 7.7). The second role is part of why the shapes of signatures always lead to the n different β_i variables. At this point the reader should be aware of how a signature's n exposed entities can be implemented by definitions imported from up to n different defining modules. It's the *espcs* component of the signature's τ_1 and the implementing module's τ_2 that is used to actually unify the β_i for some entity exposed in τ_1 and the ν_i for *the corresponding entity* exposed in τ_2.

2 Except when specified explicitly with the straightforward nooverlap(−) property.

As with *dspcs*, we implicitly consider a vector of entity name specifications (*espcs*) to declare non-overlapping entity names ($\overline{\chi}$).

- The third and final component of a module type is a set of imported module identities. This part of the module type is used by the thinning feature of package inclusion. Effectively it allows "walking the dependency tree" of some module, collecting up information like its free module identity variables.

Module types classify both concrete module implementations and abstract module signatures. Often they're paired with a sign or polarity (m), with $+$ meaning the former and $-$ meaning the latter.

Module types form a partial commutative monoid (PCM) through a key merging operation ($\oplus$), which itself induces a partial order ($\leqslant$). The former means "combine the information in the two types" (think of signature *merging*) and the latter means "the l.h.s. type has more information than the r.h.s. type" (think of signature *matching*). Figure 7.3 contains the key definitions.

Module types appearing in module contexts are annotated with a polarity and a world, written $\tau^m@\,\omega$. The m specifies the sign of the module while the ω specifies the world the module inhabits.

MODULE SHAPES The recurring theme is that shapey variants are like typey variants minus core-level typing information like semantic types (*typ*) and semantic class constraints (*cls*). So module *shapes* ($\hat{\tau}$) have shapey versions of entity definition specs, $\widehat{dspcs}$, which contain only syntactic names of defined entities; for instance entities only the token `instance`. All the PCM definitions and annotations from module types carry over to module shapes.

PHYSICAL AND LOGICAL MODULE CONTEXTS Physical module contexts (Φ, as in "physical") are the primary typing context for modules and have already been demonstrated at length in Chapter 3. They act as finite mappings from module identities (ν) to typings of those modules, each such mapping written $\nu{:}\tau^m@\,\omega$, which denotes that module ν has type τ and inhabits world ω and is either a module implementation or signature depending on the *sign* m ($+$ for module, $-$ for signature). They contain module identity variables (α, β) which are implicitly bound; later, we shall see slightly more careful binding of them as part of package types.

Physical module contexts also form a PCM, as defined in Figure 7.4. Merging on these objects is a key operation in Backpack: it's precisely the operation that *structurally* links together modules, unifying holes and matching implementations against them. This structural merging is based on *physical*, *semantic* module identities (ν), not on the *logical*, *syntactic* module paths (ℓ). Other formalizations based on mixin linking, in particular those based on *linksets*,[3] contain such an operation, in some form.

What's missing from a physical module context is the mapping from *logical* path variables ℓ to the *physical* module identities they point to. That's precisely what is captured by logical module contexts ($\mathcal{L}$, as in "logical"), which are covered in the section on the package level. Backpack provides mixin linking *by (logical) name*, a feature that is enabled by the logical context's mapping from those names into the physical modules they denote.[4]

Logical contexts have another primary use besides mixin-linking at the package level: they determine which modules are designated by *import statements*, via *import resolution*. Every module imports logical paths $\overline{\ell}$, and it's up to the logical context to map those to particular module identities, as $\overline{\ell \mapsto \nu}$, whose entities are then determined by the typings (or really, the shapings) of $\overline{\nu}$ in the physical context Φ. Since physical and logical contexts are thus highly

3 Cardelli (1997), "Program fragments, linking, and modularization".
4 In earlier work, logical module contexts were written as $\mathcal{B}$ and mapped logical names (written p) to both module identities (ν) *and* module types (τ). The additional type was used to support examples like that of (Kilpatrick *et al.*, 2013, §3). Here, however, alias bindings confer no such typing information and instead purely affect the logical module context.

bound together, we pair them up as a compound semantic object simply called the *module context*, written as Γ to resemble traditional contextual typing judgments.

A fundamental property to note about the definition of physical module contexts is what I deemed the Package-Level Consistency (PLC) property in §4.4.5, specified as Property A.3. This property asserts, quite strongly, that among all the worlds inhabited by modules in a particular context Φ, none of them define conflicting instances with each other. (Recall the discussion of PLC from Chapter 4, §4.4.5.) I assume PLC to be a baked-in invariant about Φ objects, preserved by merging and always true of any explicit constructions of Φ as a vector of ν typings that appear in the formalization. In the section on the Elaboration Soundness Theorem, §9.6, PLC will play a key role.

Finally, the elaboration into the IL will act primarily on physical module contexts. Indeed, given a context, one can map out the entire physical structure of a program: some modules and signatures, their types, and (via those types) their imports of other modules and signatures, along with their worlds and entities.

WORLDS A *world* (ω) is a finite mapping from instance heads to module identities. Each such mapping, called a *fact*, describes a type class instance: its head and the identity of the module that defined it. See Figure 7.5 for relevant definitions on worlds, summarizing Chapter 4, and briefly described below.

An additional property of a world is that each of the instance heads in the domain of the mapping must be non-overlapping, *i.e.*, they must *avoid*, in the same sense as "overlapping instances." Concretely, a world ω by definition satisfies $\mathsf{consistent}(\omega)$. As a result, if any two mappings are unifiable in their type variables (a)—not in their module identity variables (α)—then those two mappings must be equal.[5]

This consistency property is a subtlety that is perhaps lost in the presentation of the syntax of worlds as mappings from instance heads to module identities. The world as a semantic object carries this invariant in such a way that every time a world object is denoted or results from an operation, the consistency property must be maintained. To the type system, *e.g.*, to the person typechecking a Backpack program, the consistency property is maintained in the definedness criteria of all the partial operations that denote worlds.

Moreover, like many other semantic objects, worlds form a partial commutative monoid (PCM): the identity is the empty world (*i.e.*, the empty mapping) and the partial merge operation ($\oplus$) merges two finite mappings, so long as they are non-overlapping. And this PCM induces an order as with other semantic objects.

Substituting module identities in a world ($\mathsf{apply}(\theta;\omega)$) is also a partial operation, since such a substitution might make two facts in a world overlap. §4.5.3 showed an example of why that partiality exists.

Finally, worlds have an extension relation ($\sqsupseteq$). A world ω_1 extends a world ω_2 exactly when $\omega_1 \leqslant \omega_2$ via the PCM order. (The name and notation are intended to refer to Kripke understandings of worlds.) A world of an implementation must extend the world of a hole it links into.

INSTANCE HEADS An *instance head* (*head*) is a pairing of a kind environment *kenv* with a semantic class constraint *cls*, over which the type variables in *kenv* are bound. See Figure 7.5 for relevant definitions, which are summarized below.

As the name suggests, an instance head denotes the "coverage" of a type class instance. For example, the instance head for the `Eq` instance for `Set` is $(a :: *)\,.\,[\nu_P]\mathtt{Eq}\ ([\nu_S]\mathtt{Set}\ a)$. Related to coverage, instance heads also have an avoidance relation (#) which determines whether the two heads are *non-overlapping* in the core types that they apply to. Avoidance is defined in

5 This invariant of the world semantic object isn't so dissimilar from conventional finite mappings, which require distinct "keys." The only difference is that instead of two mappings' keys needing to be *distinct*, in worlds the two mappings' keys (*i.e.*, instance heads) need to *avoid*, *i.e.*, not overlap.

terms of unification on core-level types, *i.e.*, over core type variables (a) rather than module identity variables (α). For example,

$$\begin{array}{lll} (a :: *).[\nu_P]\underline{\texttt{Eq}}\ ([\nu_S]\texttt{Set}\ a) & \# & (a :: *).[\nu_P]\underline{\texttt{Ord}}\ ([\nu_S]\texttt{Set}\ a) \\ (a :: *).[\nu_P]\texttt{Eq}\ ([\nu_S]\texttt{Set}\ a) & \not\# & ().[\nu_P]\texttt{Eq}\ ([\nu_S]\texttt{Set}\ ([\nu_P]\texttt{Int})) \quad \text{(overlap)} \\ (a :: *).[\nu_P]\texttt{Eq}\ ([\nu_S]\texttt{Set}\ a) & \# & ().[\nu_P]\texttt{Eq}\ ([\nu_P]\texttt{Int}) \end{array}$$

Instance heads are α-convertible in their type variables, so we extend ordinary syntactic equality to α-equality. In other words, we treat the following two instances heads as syntactically equal:

$$(a :: *).[\nu_P]\texttt{Eq}\ ([\nu_S]\texttt{Set}\ a) \quad \text{and} \quad (b :: *).[\nu_P]\texttt{Eq}\ ([\nu_S]\texttt{Set}\ b)$$

WORLD SHAPES A *world shape* ($\widehat{\omega}$) is the "shapey" variant of a world that exists during the shaping pass (and which is propagated as part of realizers (ρ)). Generally in Backpack the shapey variants of semantic objects lack any core-level typing information as compared to their typey variants Unlike the typey variants, module shapes have no non-overlapping property, so their merging operation ($\oplus$) is the trivial union of mappings, regardless of heads that overlap on core types.

Consider a module or signature that knows, during shaping, about a signature-declared instance $[\nu_C]\texttt{C}\ ([\nu_T]\texttt{T}) \mapsto \nu_1$ and a module-defined instance $[\nu_C]\texttt{C}\ ([\nu_T]\texttt{T}) \mapsto \nu_2$. These two facts clearly overlap but are distinct, as their defining module identities differ. However, consider the case that further in the package, linking will yield a substitution θ such that $\theta(\nu_1) = \theta(\nu_2)$, unifying the two facts—and the two instances—into a single one. For this reason, we relax the non-overlapping restriction for world shapes.

Because consistency is not baked into the definition of world shapes like it is with worlds, there's a separate definition $\mathsf{consistent}(\widehat{\omega})$ that has the exact same meaning.

REALIZERS For both modules and signatures in the typing pass, module identities must be passed in as "inputs" from the package level, after having been determined during the shaping pass, to be assigned to the core bindings contained within. How are these inputs transmitted from package to module level? Through a semantic object called a *realizer* (ρ).[6]

For signatures, the realizer (ρ) is a pairing of a module shape ($\widehat{\tau}_0$) with a world shape ($\widehat{\omega}_0$), which together determine the provenance of each entity declared in the signature: the former are determined by matching up the entity name from the declaration to the corresponding export spec (*espc*) in the shape ($\widehat{\tau}_0$), and the latter are determined by locating an instance declaration's head, interpreted via a core environment (*eenv*), in the world shape ($\widehat{\omega}_0$). See Figure 7.6 for an example of a signature and a corresponding realizer.

For modules, on the other hand, a single module identity (ν_0) subsumes the work of the realizer, as all definitions in the module trivially have that identity as their provenance. There is therefore no realizer or realizing for modules.[7]

Realizers are a roundabout way to transmit identities to the module level. Cleaner would be to "pass in" identities in direct correspondence with signature declarations. However, using the module shape and world shape as a realizer enables a very clean transfer of identities from shaping pass to typing pass, in the package-level signature binding rule (TYPKGSIG).

REALIZABLE AND IDENTIFIED BINDINGS There are generally four modes of analysis for a module-level expression, based on the combinations of shaping vs. typing and modules vs. signatures. In all such modes but the typing of signatures, the module identities designated as the provenance for the core-level bindings are straightforwardly "passed into" the analysis.

6 The term "realizers" comes from MixML (Rossberg and Dreyer, 2013).

7 In MixML realizers refer only to the abstract, unknown type components of a mixin module. In both that system and Backpack, they're unnecessary for concrete modules.

For shaping and typing of a module, there's only the one provenance for all core-level definitions—the module identity assigned to the module (usually written as ν_0), which is itself considered an input to the analysis. For shaping of a signature, there are freshly generated variables β_i designated to be the provenances of the core-level declarations. In all three of these cases, the provenances are passed from the package-level judgments (which is the domain of mixin linking and module identity unification) to the module-level judgments as a vector of module identities (or single identity) corresponding to the vector of core-level bindings. And in all three of these cases, that pairing of identities and bindings is called *identified bindings* (*ibnds*). It's a straightforward construct.

Constructing an entity environment for the local core-level bindings, with identified bindings, is similarly straightforward: pair up the locally bound entity names with their corresponding provenances, and you have an *eenv*. Similarly, constructing a local world simply involves a world with a $\mathit{fact}_i \mapsto \nu_i$ for each locally bound instance.

The fourth case, however, is more complicated. For typing of a signature, there is not simply a vector of module identities to be designated as the provenance of each declaration. Instead, as just discussed, there are *realizers* that act as the transmission of provenances from package level to module level. The pairing of a realizer and local core bindings is called *realizable bindings* (*rbnds*). More accurately, the pairing of provenances, in some form, with local bindings are considered realizable bindings, whether that comes in the form of the realizer (for signature typing) or the identified bindings (for other analyses).

The reader might wonder why we need these two seemingly very bespoke objects. The answer is that they help the definitions for core environment construction appear more regular and parallel—more generic—by shoving most of the distinction in the analysis into auxiliary definitions like $\mathsf{mklocworld}(-;\ -)$, all of which are defined in Figure 7.8.

AUGMENTED ENTITY ENVIRONMENTS An augmented entity environment *aenv* is a deeply technical semantic object that doesn't appear in the type system of Backpack. Instead, it's used in the metatheory about core environment construction.

Since we have a contextual judgment that derives a core environment $\mathit{eenv} @ \omega$ for a module expression, how do we state a property that the core environment is well-formed with respect to the context? *Regularity* properties like this tend to say things like "if e is judged to have type t in context Γ then t is well-formed with respect to Γ." Here, we can't simply say that $\mathit{eenv} @ \omega$ is well-formed with respect to the context, Γ because that context only contains the *imported* modules, and the core environment will also mention module identities, types, etc. from the *local* bindings.

An augmented environment is essentially a pairing of module context for imports and realizable bindings for locals. In particular it acts as the context for well-formedness judgments $\mathit{aenv} \Vdash \mathit{eenv}\ \mathsf{loc\text{-}wf}$ and $\mathit{aenv}; \mathit{eenv} \Vdash \omega\ \mathsf{loc\text{-}wf}$, which state that an entity environment or a world (shape) is "locally well-formed." As an example of where this judgment is employed, check the import resolution soundness metatheory (§7.7.3).

The utility of this kind of machinery might seem unnecessary. The reader should recall that, unlike other semantics for Haskell modules (of which core environment construction, *i.e.*, import resolution is a key part), Backpack's semantics comes with metatheory to justify the correctness of (most of!) the definitions. To develop a sense for how intrinsically complicated is the business of import resolution and core environments in Haskell, I refer the reader to the similar definitions of Faxén,[8] definitions which don't come with metatheory.

7.4 CORE ENVIRONMENT CONSTRUCTION

Figure 7.7 contains the definitions for core environment construction, *i.e.*, the entity environment (*eenv*) and world (ω) determined from the imported modules and the local core bindings. There are two variants with exactly parallel structure: a shapey one and a typey

8 Faxén (2002), "A static semantics for Haskell," Figures 11 and 15.

one. And both definitions comprise two parts: the core environment $eenv_i @ \omega_i$ for the imports and the core environment for the local bindings, which are then merged together to form the result. The first part is straightforward, so let's focus instead on the locals.

constructing the local entity environment The auxiliary definition $\mathsf{mkloceenv}(rbnds)$ converts the realizable bindings into a local entity environment. With the realizable bindings from Figure 7.6 as an example, $\mathsf{mkloceenv}((\rho_{\mathsf{Set}} \mid \overline{decl_{\mathsf{Set}}}))$ would yield the following entity environment:

$$\mathsf{mkeenv}\begin{pmatrix}[\beta_1]\mathtt{Set()} \\ [\beta_2]\mathtt{empty} \\ [\beta_3]\mathtt{member} \\ [\beta_4]\mathtt{insert} \\ [\beta_5]\mathtt{toAscList}\end{pmatrix} \oplus \quad \ldots$$

The realizer ρ_{Set} was used to designate the provenances for the syntactic declarations $\overline{decl_{\mathsf{Set}}}$, which were then straightforwardly mapped to entity export specs (*espcs*). The remaining $\mathsf{mkeenv}(-)$ function simply maps the five export specs into a syntactic mapping from entity name to its physical name. Finally the r.h.s. is simply the same as the l.h.s. but each syntactic reference is prefixed with the (reserved keyword) identifier `Local`.

One low-level detail in the definition of $\mathsf{mkeenv}(-)$ is the matching operator $\sqsubseteq$, which is defined when the entity spec on the left is mapped down into the export spec on the right. It's just extracting the name—and any potential subnames, like data constructors for a type or instance methods for a type class—from the entity spec.

That was an example of constructing a local entity environment, for signature typing, using a realizer. An example for shaping or for module typing would be even more straightforward: instead of having to "look up" the provenance of each core binding in a realizer, they're just given in order by the identified bindings.

constructing the local world The construction of local worlds is a little trickier. The goal is to create a local world ω_0 that contains all the locally bound type class instances. There's an added complication here because this definition requires the full entity environment for these bindings; and not just the environment synthesized for the local bindings in the last step, but also for the imported entities. That's because ω_0 mentions *semantic* class constraints *cls*, which we can only get from the local instances by mapping their *syntactic* instance heads via an entity environment—an entity environment which should also know the physical names of locally bound classes and types.

There are three definitions of the $\mathsf{mklocworld}(-;\ -)$ operation: a trickier one for signature typing which again uses a realizer, a more straightforward one for module typing, and a similar one for both kinds of shaping. We consider first the latter cases, defined for identified bindings *ibnds*. The idea goes as follows:

1. The set J contains the indices, into the vector of local bindings, of any `instance` bindings. This index set will be used also as an index into the designated identities, $\overline{\nu}$.
2. For each index $i \in J$, the mapping from syntactic instance head into semantic instance head must be defined. This rules out any ill-formed types or class constraints.
3. Every pair of `instance` bindings must avoid each other; *i.e.*, the space of types they apply to, if defined for the same class, must not overlap.
4. Then the result of the operation is the world gotten from the mapping of local semantic instance heads to the corresponding provenance ν_i.

The case for the shaping variant, $\mathsf{mklo\hat{c}world}(-;\ -)$, is exactly the same, with the trivial distinction of creating a world *shape* $\widehat{\omega}$ not a world ω.

Now the trickier case, for signature typing, $\mathsf{mklocworld}(\mathit{rbnds};\ \mathit{eenv})$. At first glance, since the realizer designates a world shape $\widehat{\omega}_0$, it might seem correct to just use that. However, the realizer might know about an eventual implementation of this signature, which perhaps contains a whole lot more instances than those declared in the signature. (Recall that for a module to implement a signature, the module's world must extend the signature's world.) So instead we must create the local world from the local instance declarations by matching those declarations against the (possibly much larger) world shape in the realizer. The r.h.s. of the definition contains exactly those facts from $\widehat{\omega}_0$ that are equal to the mapping of a local *syntactic* instance declaration into a *semantic* instance head. In effect, $\widehat{\omega}_0$ designates the provenance ν of the instance. And any other facts known to the realizer's world shape don't matter.

But what about the additional $\mathsf{consistent}(\widehat{\omega}_0)$ requirement? Essentially it serves to keep signature deterministic in the presence of ambiguity, as the $\widehat{\omega}_0$ might actually have two different mappings $\mathit{head} \mapsto \nu_1$ and $\mathit{head} \mapsto \nu_2$. (Recall that world *shapes* are simply sets and do not bake in any kind of consistency.) This requirement also serves to fail fast in signature typing, instead of allowing packages to be well-typed despite requiring very particular conditions to use.

As an example, consider the following package, along with the world shapes synthesized for its two bindings:

$$\begin{array}{l}
\textbf{package}\ \mathsf{unreasonable}\ \textbf{where} \\
\quad \mathsf{A} = \left[\begin{array}{l} \quad\texttt{data T = T Int} \\ \texttt{instance Eq T where ...} \end{array}\right] \\
\quad \mathsf{B} :: \left[\begin{array}{l} \quad\texttt{import A} \\ \texttt{instance Eq T} \end{array}\right]
\end{array}
\qquad
\begin{array}{lcl}
\widehat{\omega}_A & = & \{[\nu_P]\mathsf{Eq}\ [\nu_A]\mathsf{T} \mapsto \nu_A\} \\
\widehat{\omega}_B & = & \left\{\begin{array}{l} [\nu_P]\mathsf{Eq}\ [\nu_A]\mathsf{T} \mapsto \nu_A \\ [\nu_P]\mathsf{Eq}\ [\nu_A]\mathsf{T} \mapsto \beta_B \end{array}\right\}
\end{array}$$

(Recall that a fresh variable β is synthesized to stand in for the provenance of each declaration in a signature.) When it comes time to *type* the signature, its realizer will pass in $\widehat{\omega}_B$ for constructing its local world. At this point, because $\neg\mathsf{consistent}(\widehat{\omega}_B)$, the local world is undefined and thus the rule (CoreEnv) cannot be applied, making the signature ill-typed.

What would happen without that consistency requirement? There'd be nondeterminism in constructing the local world, as there are two possible mappings for this instance in the realizer. Or perhaps one could apply unification *within* the world shape (during (ShPkgSeq), §8.2), so as to unify the two facts inside it and thus the two world shapes. That, however, would be quite a departure from the rest of linking unification in Backpack, which is always directed by *logical module names*, and here the two names are distinct.

IMPORT RESOLUTION There are two points worth mentioning about the semantics of import resolution.

First, subordinate names of entities cannot be imported in isolation from their owning names. In particular, Haskell allows the import of a type class method (*e.g.*, `hash`) without importing the class itself (*e.g.*, `Hashable`). In Backpack, because imported names are designated by an export spec *espc*, and because subordinate names do not have their own *espc*, there's no way to import those subordinate names separately. For example, the *espc* $[\nu_H]$`Hashable(hash)` denotes the export of the class and the class method, and $[\nu_H]$`Hashable()` denotes the class without any methods, but there's no *espc* for `hash` itself.

Deviation from Haskell: importing subordinate names.

Second, the judgments for import resolution are *shapey*: they are defined on module *shape* contexts, not full typing contexts. That's so that the shaping and typing judgments can reuse the same definitions; note the $\mathsf{shape}(\Gamma)$ appearing in (CoreEnv). There would be even more "sharing" of definitions if not for the presence of world *shapes* in the shapey variants, which subtly change the meaning.

7.5 EXPORT RESOLUTION

Figure 7.9 contains the definitions of export resolution for modules. For the most part these are straightforward specifications of Haskell semantics. Some notes on the definitions:

- Haskell permits export of subordinate names without their owning names in the case of class methods. Backpack doesn't allow this because every exported entity must have its own *espc*, but a class method does not. This exactly parallels the situation for imports just discussed.

Deviation from Haskell: exporting subordinate names.

- (ExpLocal): In the case that no exports are explicitly declared (*i.e.*, *expdecl* $= \cdot$), the resolved exported entities are exactly those contained in the entity environment *eenv* that are qualified with the distinguished module reference `Local`. The definition of core environment construction, of `mkloceenv` in particular, was set up so that such entities are exactly the locally defined ones.

- (ExpModAll): As per the Haskell spec,[9] exactly those entities which are accessible both unqualified (as χ) and qualified (as *mref* `.`χ) are exported with "`scope` *mref*."[10] But in Backpack I impose an additional restriction, the bottom premise of the rule: every entity so exported must be unambiguously identifiable with both the qualified and unqualified entity reference. This restriction was required in the proof of soundness of the elaboration (Lemma A.136), in order to prove that it's semantics-preserving to elaborate a module to one with all exports explicit.Therefore this rule shows how metatheory can iron out the exact definitions of subtle language features like Haskell's `module` exports.

Deviation from Haskell: export lists.

7.6 SHAPING AND TYPING JUDGMENTS

Figure 7.10 provides the shaping and typing judgments for the module level. One can immediately see some parallelism among the four judgments:

- All four judgments involve (1) an ambient module context ($\Gamma/\widehat{\Gamma}$), (2) the construction of a core environment (*eenv* @ $\omega/\widehat{\omega}$), (3) the checking of core bindings (*decls* or *defs*) according to specs ($dspcs/\widehat{dspcs}$), and (4) the determination of what specs to expose (*espcs*).

- The two module judgments differ only in the shapeyness of certain objects and in the way in which the core entity specs are determined: either derived directly from the definitions (for shaping) or by *core typing* of them (for typing).

- The two signature judgments differ in similar ways and in one key additional way. Both judgments either derive shapey entity definition specs or full entity specs via core typing. But they also differ in how they determine the provenances of the core entities they declare, $\overline{\nu}$.

The judgments have been designed to elucidate this parallelism. As noted in the introduction, shaping is based on the "static pass" in type systems for recursive modules, a la Dreyer's work. Instead of defining two sets of highly parallel rules, Dreyer employed clever *shading* of select parts of typing rules, such that the variant for the static pass was defined to be the same rules but with shaded premises removed.[11]

9 Marlow (2010), "Haskell 2010 Language Report," §5.2.

10 Why did Haskell define this export form in this way? In private correspondence, early Haskell contributor Simon Peyton Jones could not recall any particular reason for the official definition. However, Iavor Diatchki suggested that the Haskell definition enables a convenient way for a module to expose an imported module's interface with modification:

```
module ModifiedA (x, module A) where
  import qualified A
  import A hiding (x)
  x = modify_in_some_way A.x
```

11 Dreyer (2007a), "A Type System for Recursive Modules," Figure 6.

MODULE SHAPING The goal with module shaping is to determine the *names and structure* of the module, rather than any typing information. Concretely, this judgment synthesizes the shape, $\widehat{\tau}$, that determines the (names of) entities defined locally, $\widehat{dspcs}$, the name specifications of exposed entities, *espcs*, and the module identities directly imported, N. Additionally, the world shape $\widehat{\omega}$ is synthesized as part of the annotated shape of the module.

The provenance of the module is passed in as the single identity of this module, ν_0, as determined by the package level. Consequently that identity is "replicated" into a vector of size equal to the local definitions *defs* in order to form the *identified bindings* (§7.3) used to construct the core environment.

MODULE TYPING In module typing the goal is not to determine simply the names and structure of the module but the full typing information about its contents. The key difference from module shaping is the way the entity definition specs are determined, *dspcs*. The shapey version of this object, $\widehat{dspcs}$, simply contains names and can be directly derived from *defs*. But here in typing we must appeal to the typing judgment of the core level. As already covered in Chapter 6, that judgment takes as "inputs" the following objects: the physical module context component of the ambient module context, denoted $\Gamma.\Phi$; the identity of this module, ν_0; and the two objects forming the core environment, *eenv* and ω, which describe the imported and locally defined entities. All of those together are used to judge the well-typedness of the definitions according to specifications *dspcs*.

SIGNATURE JUDGMENTS Signature judgments necessitate an additional component not present in the module judgments: the *signature declaration context*, written Φ_{sig}.

Already we've seen plenty of times, most recently in the discussion of *realizable bindings* (§7.3), that the n declarations in a signature are actually granted n different module identities to represent their provenances. For signatures that designate holes in a package (*i.e.*, those signatures left unimplemented), those "β" variables allow the n entities to have been defined in any module, not simply the module that (eventually) implements the hole, *i.e.*, the module for which α is substituted. In the example of Figure 7.6, for the Set signature, they were $\beta_1, \ldots, \beta_7$; the variable for the signature itself would be some additional module identity variable, α.

The consequence of that setup is that each β_i denotes itself another module expression—an abstract one, a signature—that declares and exposes the corresponding declaration $decl_i$ from the original signature. Those n additional signatures each have a module type of the form

$$\tau_i = \langle\!|\, dspc_i \,;\, espc_i \,;\, \cdot \,|\!\rangle$$

That type locally declares the $decl_i$, which has spec $dspc_i$, and exposes that entity with name spec $espc_i$. The merging of these n additional signatures is called the *signature declaration context*, written Φ_{sig} and defined in Figure 7.10.

A signature S is judged well-formed with respect to a module type τ, but that τ has no defined entity specs (*dspcs*) in it. Instead, the defined entity specs come from the signature declaration context for S. However, τ does expose all of the entities from that context, since clients of the signature should be able to import them from that signature. The module type of S—before any merging into the module context, at the package level—therefore has the form

$$\tau = \langle\!|\, \cdot \,;\, espcs \,;\, \cdot \,|\!\rangle$$

The typing information about the core-level declarations is relegated to the accompanying signature declaration context as an additional "output" of signature judgments.

Figure 7.11 presents an example of a signature declaration context, from the Set signature and realizer of the running example. Given the definition of $\mathsf{sigenv}(-;\ -;\ -)$ in Figure 7.10, we can see how this example context is constructed by merging the singleton contexts corresponding to each of the n entities. Two points about the definition are worth noting. First, the *espc* is derived from the shapey entity definition spec $\widehat{dspc}$ and the associated provenance ν_i (in our example, β_i). Second, each such singleton context has the same world ω for the whole signature.

SIGNATURE SHAPING With signature declaration contexts in mind, the signature shaping judgment is fairly straightforward. That's because, as we saw in the discussion of realizers, the provenances of the *decls* are passed into the judgment as a simply vector of module identities. A core environment is constructed; the shapey entity definition specs are derived from the declarations; name specs are derived from those specs and the provenances; and the signature declaration context is constructed. Notably, that context is actually shapey, but the definition of the shapey version of construction, $\widehat{\mathsf{sigenv}}(-;\ -;\ -)$, differs only in the shapeyness of the *dspcs* and ω. The actual module shape of the signature is empty except for the n entity name specs, *espcs*, for the n declarations bound in the $\widehat{\Phi}_{\mathsf{sig}}$.

SIGNATURE TYPING At this point most of the pieces of this judgment look straightforward. Like with module typing, a core environment is constructed and passed to the core-level typing judgment in order to ascertain the entity definition specs (*dspcs*). And like with signature shaping, those entity definition specs determine the name specs (*espcs*) and, along with the provenances and world, the signature declaration context Φ_{sig}. The difference from those judgments lies in the determination of the provenances, $\overline{\nu}$, which is captured in the top right premise.

This determination of provenances in signature typing is called *signature declaration realization*, a judgment also defined in Figure 7.10. As the name implies, it's the mechanism that puts the realizers to work. See Figure 7.6c for the instance of this judgment on the Set example.

Rule (1) specifies the realization for a named, non-`instance` declaration. The first premise identifies an *espc′* in (the module shape component of) the realizer such that it matches, possibly by being more specific (*i.e.*, having more subnames) than, the concerning *decl*. The rule is deterministic since the *espcs* inside module shapes (and types) have non-overlapping names.

Rule (2) specifies the realization for an `instance` declaration. The syntactic head of the instance, $\mathsf{head}(decl)$, is interpreted through the entity environment in the judgment, resulting in some semantic instance head *head*. If there's a fact $head \mapsto \nu$ in (the world shape component of) the realizer, then the provenance of this instance declaration is that ν.

Finally, rule (3), technically a judgment on a *vector* of declarations, simply maps the main judgment over that vector.

7.7 METATHEORY OF THE MODULE LEVEL

Backpack's module level is defined not just as a type system, specified with typing judgments, but also as metatheory that validates those judgments. The latter distinguishes (the Haskell portion of) Backpack from previous formalizations of Haskell's module system. The development of this metatheory was driven by the statement and proof of the main Soundness theorem for Backpack. The aspects of the metatheory specific to the elaboration into the IL will be presented later §9.3. For now I'll concentrate on those aspects of the metatheory that, while used by the Soundness proof, constitute some reasonably conventional statements one would like to know about a type system.

Backpack's language of types is extremely rich. Take module types for example; they contain multiple objects—module identities, entity names, and entity specifications—that only make sense in particular contexts. Because of that richness, it's important to know that the module types judged to be the types of module expressions are themselves *well-formed*, *i.e.*, that all the constituent objects inside them make sense. This notion of well-formedness is generally defined via inductive inference rules in the form of judgments, one judgment per semantic object to be considered well-formed.

The definition of well-formedness aside, a key property about the module-level typing judgments is that, when the context for a typing judgment is well-formed, then so is the "result" of the judgment, like the module type. As in the rest of the formalization, we refer to these properties as *Regularity* statements.

In this section I present the definitions of well-formedness judgments on some key semantic objects for the module level. Then I present the regularity statements about the typing judgments at the module level, along with a sketch of their proofs. Of particular note are the corresponding regularity properties on core environment construction and on export resolution, both of which are involved in those proofs.

7.7.1 *Well-formedness*

The various well-formedness judgments for the module level aren't too surprising for the most part. On module types and shapes, they reduce trivially to the well-formedness of core objects, already covered in §6.5.1.

One notable distinction is between well-formedness of world facts ($\mathit{head} \mapsto \nu$) with respect to a physical module context and that of world facts with respect to a physical module context *shape*. the latter does not structurally resemble the former. Rule (WfFact) judges a world fact to be well-formedness in Φ when Φ maps the module ν to a type τ that locally binds an instance whose head is equal to that of the fact. Essentially, it verifies that the context properly contains the instance that originated the fact, the evidence of it. Rule (WfFactShp), on the other hand, cannot search the context $\widehat{\Phi}$ for the originating instance because module context shapes don't have any information about the instances or their heads; recall that the instance form of $\hat{\mathit{dspc}}$ in module shapes is simply a token `instance`. Instead, the rule requres that the fact be contained in the world shape bound to the module ν.[12]

Well-formedness on module contexts is less straightforward, for two key reasons. First, a module typing in a context, $\nu{:}\tau^{\mathsf{m}}\ \omega \in \Phi$, depends not only on "previous" modules bound in the context, but also on *itself*; entities defined locally in ν will manifest, as physical names with provenance ν, in the type τ. Then judging well-formedness of this typing requires a self-reference to ν.

Second, modules in Backpack and in Haskell are implicitly defined in recursive knots, not sequentially. (Backpack's module bindings are processed sequentially, but the semantic object representing those bindings—physical module contexts—does not preserve that sequence.) As a result, multiple module typings within a context might refer to each other. For example, two mutually recursive module implementations in Φ, $\mu\alpha_1.\nu_1$ and $\mu\alpha_2.\nu_2$, will mention each other's identities in the "imports" component of their module types.

To address both of these requirements, well-formedness of a physical module context must *assume itself*, thereby enabling the consideration of such recursive references. Well-formedness of (combined) module contexts also does this in the application of the judgment to its constituent logical module context (WfModCtx). The main rule for physical module context well-formedness (WfPhModCtx) exhibits this strange definition: the context to be judged is merged into the ambient context in the premises. In those premises are *auxiliary* judgments, $\Phi \vdash \Phi$ X-wf, where X is one of four analyses of four different components of physical module context bindings: specs, for the entity definition specs in module types; exps, for the entity name/export specs in module types; imps, for the imports in module types; and wlds, for the worlds annotating module types. And shapey versions as well, of course.[13] All such definitions are provided in Figure 7.13. Although the context being judged as well-formed doesn't get smaller from the consequent to the premises, the judgment on that context switches to these "smaller" judgments.

12 Further review of the formalization would perhaps reveal this premise to be unnecessary for the desired proofs.

13 Notably absent from (WfPhModCtxShp) is the shapey version of the imps-wf judgment. This property was simply unnecessary in the proofs of the formalization as presented in this thesis.

The four constituent judgments are mostly straightforward: though defined on whole physical module contexts, they're decomposed into a universal property on all annotated module types (or module context typings) in the context. Some notes on them follow:

- (WfSpecs)/(WfShSpecs): The $\mathsf{validspc}(dspc; m)$ property simply determines whether the given spec is valid for the polarity of the module. Essentially, this verifies that abstract data type declarations only appear in signatures, *i.e.*, for $m = -$.

- (WfShSpecs): There's no well-formedness judgment on shapey entity definition specs ($\hat{dspcs}$) since they're too trivial to say anything about.

- (WfSigImports): This rule ensures that the imports inside *signature* types are always empty; *cf.* §7.6.

- (WfModImports): This rule ensures that any module identities occuring in the type of a (concrete) module, as physical name provenances, ($\mathsf{provs}(\tau)$) or in its world, as world fact provenances, ($\mathsf{idents}(\omega)$) are either (1) the module itself (ν) or (2) gotten from walking the dependency graph of the module's imports and collecting up provenances in imported modules' types, transitively ($\mathsf{depends}^{+}_{\Phi}(\mathsf{imps}(\tau))$). (This auxiliary function is defined in the appendix.) A module context typing would *not* have well-formed imports if, for example, there exists a physical name $[\nu_?]\chi$ such that $\nu_?$ is not (a provenance in the type of) a transitively imported module.

- (WfWorlds): The second premise requires that the world of the module context typing (ω) extends the world extracted from locally-bound instances in the type (τ) of the module (ν). In other words, the context's world for the module must have at least every local instance defined in the module, as determined from the entity definition specs of the module type. Similarly here, a module context typing would *not* have a well-formed world if there exists some `instance` spec that doesn't appear, with this module as its provenance, in that world.

7.7.2 *"Local" well-formedness*

In the metatheory at the module level there is an additional notion of "local" well-formedness, encapsulated in judgments like $aenv \Vdash eenv\ \mathsf{loc\text{-}wf}$. This notion does not appear in the main well-formedness definitions described in the previous section. Instead it's used internally as part of the statements and proofs of metatheory at the module level.

Local well-formedness is necessitated by the fact that entities appearing in the core bindings within a module expression originate from two places: the physical module context, when they are imported from other modules, and the local core bindings, when they are bound locally inside this module expression.

One way around this bifurcation would be to add the "recursive self" module binding to the context as part of the well-typing of a module expression. At first glance this looks just like how the well-formedness of physical module contexts "assumes itself." With a *typing* judgment, however, it's a different story: the typing judgments of Backpack have been designed to be *deterministic* so as to also describe a *typechecking algorithm* for expressions. For the typing of a module to assume itself, we'd have to nondeterministically guess its module type (and, in the case of a signature, its signature declaration context) in order to move it into the context, a nondeterministic guess that the other judgment doesn't require.

The key context in the local well-formedness judgments is the *augmented entity environment*, as introduced in §7.3. This environment packs in those two sources: the module context (shape) and the local bindings. Local well-formedness with respect to an augmented entity environment looks at *shapey* information about names and provenances only. In particular, in the full details in the appendix, the ctxmatch and locmatch definitions on these environments describe how to perform the bifurcated lookup of the source of any entity. Their details are uninteresting and left entirely to the appendix.

7.7.3 *Regularity of module-level typing*

As in the general theme of the formalization, the typing judgments of the module level come equipped with regularity properties that enable the composing of well-formedness judgments. Aside from the elaboration soundness statements (), the two key metatheoretic statements about the module level are the regularity of module and signature typing. Both statements are presented as part of Theorem A.1 but are restated below (assume $\Gamma = (\!|\, \Phi\,;\, \mathcal{L}\, |\!)$):

- **Regularity of module typing**: If $\Phi \vdash \Gamma$ wf and $\Gamma; \nu_0 \vdash M : \tau @ \omega$ and $\Phi \oplus_? \nu_0{:}\tau^+ @ \omega$, then $\Phi \vdash \nu_0{:}\tau^+ @ \omega$ wf. A well-typed module, in a well-formed context that's mergeable with the module itself, has a singleton physical module context that is well-formed in the context.

- **Regularity of signature typing**: If $\Phi \vdash \Gamma$ wf and $\Gamma; \rho \vdash S : \sigma @ \omega \mid \Phi_{\mathsf{sig}}$ and $\Phi \oplus_? \Phi_{\mathsf{sig}}$ and $\Phi \oplus_? \nu_0{:}\sigma^- @ \omega$, then $\Phi \vdash \Phi_{\mathsf{sig}}$ wf and $\Phi \oplus \Phi_{\mathsf{sig}} \vdash \nu_0{:}\sigma^- @ \omega$ wf. A well-typed signature, in a well-formed context that's mergeable with the signature itself and with the signature environment context, has a singleton physical module context that is well-formed in the context and a signature environment context that is well-formed in the context.

The proofs of these regularity statements will be discussed shortly. At a high level, though, they're decomposed into properties about core environment construction, export resolution, and the axioms about core-level typing.

CORE ENVIRONMENT CONSTRUCTION SOUNDNESS The semantics of core environment construction are validated with a corresponding soundness property: roughly, if the module context is well-formed and if the core environment construction judgment is derivable, then the entity environment and world are themselves well-formed. Well-formed in, well-formed out. This soundness property is needed in the proof of regularity of module-level typing. Its own proof requires a number of highly technical lemmas for the various judgments defining import resolution, and for the auxiliary definitions like $\mathsf{mklocworld}(-;\ -)$. See Lemma A.17 for the full statement, along with the subsequent technical lemmas needed for its proof.

EXPORT RESOLUTION SOUNDNESS Serving a similar purpose is the soundness property for export resolution. This property, also needed in the proof of regularity of module-level typing, ensures that in the ambient module context extended with "this module," this module's type has well-formed exports, *i.e.*, entity name specs. The statement on modules is defined as Lemma A.27, while the statement on signatures (rather, signature environment contexts) is defined as Lemma A.28.

PROOFS OF REGULARITY OF TYPING The proofs of regularity of module and signature typing require the properties above, along with various technical lemmas provided in the appendix. Critically, the proofs also require the axioms about core typing presented in the previous chapter (§6.5). Now, in the context of these proofs at the module level, the exact premises of those axioms should make more sense.

The proof of regularity of module typing composes core environment construction soundness (Lemma A.17) and the axiomatized core definition typing regularity (Axiom A.15) to get specs-wf. Then the proof of exps-wf comes from Lemma A.27, and imps-wf from Lemma A.49. And finally Lemma A.37 yields wlds-wf.

The proof of regularity of signature typing first addresses the well-formedness of Φ_{sig}. First, Φ_{sig} specs-wf is a direct result of the axiom about core-level typing of signature declarations (Axiom A.16). Second, exps-wf comes from Lemma A.28. Third, wlds-wf is a straightforward corollary of the trickier lemma Lemma A.36. And fourth, imps-wf is trivial since module types in signature contexts have no imported identities. As a result, $\Phi \vdash \Phi_{\mathsf{sig}}$ wf. Then the well-formedness of the singleton context comes from Lemma A.38.

Algebraic Definitions for Module Types

Module types merging

$\boxed{\tau \oplus \tau :_{\mathsf{par}} \tau}$ $\boxed{\tau \oplus_{?} \tau}$

$$\langle\!| \overline{dspc_1} ; \overline{espc_1} ; \overline{v_1} |\!\rangle \oplus \langle\!| \overline{dspc_2} ; \overline{espc_2} ; \overline{v_2} |\!\rangle \overset{\mathsf{def}}{=} \langle\!| \overline{dspc} ; \overline{espc} ; \overline{v} |\!\rangle$$

$$\text{where} \begin{cases} \overline{dspc} = \overline{dspc_1} \oplus \overline{dspc_2} \\ \overline{espc} = \overline{espc_1} \oplus \overline{espc_2} \\ \mathsf{nooverlap}(\overline{dspc}) \wedge \mathsf{nooverlap}(\overline{espc}) \\ \{\overline{v}\} = \{\overline{v_1}\} \cup \{\overline{v_2}\} = \{\overline{v_1}\} \text{ or } \{\overline{v_2}\} \end{cases}$$

$$\langle\!| \overline{dspc_1} ; \overline{espc_1} ; \overline{v_1} |\!\rangle \oplus_{?} \langle\!| \overline{dspc_2} ; \overline{espc_2} ; \overline{v_2} |\!\rangle \overset{\mathsf{def}}{\Leftrightarrow} \begin{cases} \overline{dspc_1} \oplus_{?} \overline{dspc_2} \\ \overline{espc_1} \oplus_{?} \overline{espc_2} \\ \mathsf{nooverlap}(\overline{dspc_1} \oplus \overline{dspc_2}) \\ \mathsf{nooverlap}(\overline{espc_1} \oplus \overline{espc_2}) \\ \{\overline{v_1}\} \cup \{\overline{v_2}\} = \{\overline{v_1}\} \text{ or } \{\overline{v_2}\} \end{cases}$$

Module types implementation order

$\boxed{\tau \leqslant \tau}$

$$\langle\!| \overline{dspc_1}, \overline{dspc'_1} ; \overline{espc_1}, \overline{espc'_1} ; \overline{v_1} |\!\rangle \leqslant \langle\!| \overline{dspc_2} ; \overline{espc_2} ; \overline{v_2} |\!\rangle$$
$$\overset{\mathsf{def}}{\Leftrightarrow}$$
$$\overline{dspc_1 \leqslant dspc_2} \wedge \overline{espc_1 \leqslant espc_2} \wedge \{\overline{v_1}\} \supseteq \{\overline{v_2}\}$$

Annotated module types merging

$\boxed{\tau^{m}@\,\omega \oplus \tau^{m}@\,\omega :_{\mathsf{par}} \tau^{m}@\,\omega}$ $\boxed{\tau^{m}@\,\omega \oplus_{?} \tau^{m}@\,\omega}$

$$\tau_1{}^{-}@\,\omega_1 \oplus \tau_2{}^{-}@\,\omega_2 \overset{\mathsf{def}}{=} (\tau_1 \oplus \tau_2)^{-}@\,(\omega_1 \oplus \omega_2)$$

$$\tau_1{}^{-}@\,\omega_1 \oplus \tau_2{}^{+}@\,\omega_2 \overset{\mathsf{def}}{=} \tau_2{}^{+}@\,\omega_2 \text{ if } \begin{cases} \tau_1 \geqslant \tau_2 \\ \omega_1 \sqsubseteq \omega_2 \end{cases}$$

$$\tau_1{}^{+}@\,\omega_1 \oplus \tau_2{}^{-}@\,\omega_2 \overset{\mathsf{def}}{=} \tau_1{}^{+}@\,\omega_1 \text{ if } \begin{cases} \tau_1 \leqslant \tau_2 \\ \omega_1 \sqsupseteq \omega_2 \end{cases}$$

$$\tau_1{}^{+}@\,\omega_1 \oplus \tau_2{}^{+}@\,\omega_2 \overset{\mathsf{def}}{=} \tau_1{}^{+}@\,\omega_1 \text{ if } \begin{cases} \tau_1 = \tau_2 \\ \omega_1 = \omega_2 \end{cases}$$

$$\tau_1{}^{m_1}@\,\omega_1 \oplus_{?} \tau_2{}^{m_2}@\,\omega_2 \overset{\mathsf{def}}{\Leftrightarrow} \left(\begin{array}{lcl} m_1, m_2 = -,- & \Rightarrow & \tau_1 \oplus_{?} \tau_2 \wedge \omega_1 \oplus_{?} \omega_2 \\ m_1, m_2 = -,+ & \Rightarrow & \tau_1 \geqslant \tau_2 \wedge \omega_1 \sqsubseteq \omega_2 \\ m_1, m_2 = +,- & \Rightarrow & \tau_1 \leqslant \tau_2 \wedge \omega_1 \sqsupseteq \omega_2 \\ m_1, m_2 = +,+ & \Rightarrow & \tau_1 = \tau_2 \wedge \omega_1 = \omega_2 \end{array} \right)$$

Module identity substitution

$\boxed{\theta\tau : \tau}$

$$\theta\langle\!| dspcs ; espcs ; \overline{v} |\!\rangle \overset{\mathsf{def}}{=} \langle\!| (\theta dspcs) ; (\theta espcs) ; \overline{\theta v} |\!\rangle$$

Figure 7.3: Partial commumative monoids (PCMs) and induced partial orders on module types and annotated module types. As usual, the $\oplus_{?}$ binary relation is just the definedness property of $\oplus$. Definitions of $\oplus_{?}$ are nonetheless stated above for clarity.

Algebraic Definitions for Physical Module Contexts

Physical module contexts merging

$\boxed{\Phi \oplus \Phi :_{\mathsf{par}} \Phi}$ $\boxed{\Phi \oplus_? \Phi}$

$$\Phi_1 \oplus \Phi_2 \overset{\mathsf{def}}{=} \overline{\nu{:}\tau^{m}@\,\omega}, \Phi_1', \Phi_2' \qquad \text{where} \begin{cases} \Phi_1 & = \overline{\nu{:}{\tau_1}^{m_1}@\,\omega_1}, \Phi_1' \\ \Phi_2 & = \overline{\nu{:}{\tau_2}^{m_2}@\,\omega_2}, \Phi_2' \\ \overline{\tau^{m}@\,\omega} & = \overline{{\tau_1}^{m_1}@\,\omega_1 \oplus {\tau_2}^{m_2}@\,\omega_2} \\ \mathsf{dom}(\Phi_1') \mathbin{\#} \mathsf{dom}(\Phi_2') \\ \forall \omega_1 \in \Phi_1, \omega_2 \in \Phi_2 : \ \omega_1 \oplus_? \omega_2 \end{cases}$$

$$\Phi_1 \oplus_? \Phi_2 \overset{\mathsf{def}}{\Leftrightarrow} \forall \begin{pmatrix} \nu{:}{\tau_1}^{m_1}@\,\omega_1 \in \Phi_1 \\ \nu{:}{\tau_2}^{m_2}@\,\omega_2 \in \Phi_2 \end{pmatrix} : \ {\tau_1}^{m_1}@\,\omega_1 \oplus_? {\tau_2}^{m_2}@\,\omega_2 \quad \wedge \quad \omega_1 \oplus_? \omega_2$$

Module identity substitution

$\boxed{\mathsf{apply}(\theta;\Phi) :_{\mathsf{par}} \Phi}$

$$\mathsf{apply}(\theta;\Phi) \overset{\mathsf{def}}{=} \bigoplus_{\nu:\tau^{m}@\omega \in \Phi} (\theta\nu){:}(\theta\tau)^{m}@\,(\theta\omega), \quad \text{if} \quad \forall \nu{:}\tau^{m}@\,\omega \in \Phi : \mathsf{extworld}(\theta\nu;\theta\tau) \text{ defined}$$

Figure 7.4: Partial commumative monoid (PCM) on physical module contexts, *i.e.*, merging of physical module contexts. As usual, the $\oplus_?$ binary relation is just the definedness property of $\oplus$. Definitions of $\oplus_?$ are nonetheless stated above for clarity. The side condition on substitution prohibits substitution from unifying locally defined instances to a module in the context; $\mathsf{extworld}(\nu;\tau)$ will be defined later.

Auxiliary Definitions on Worlds and World Shapes

Instance Heads

$\boxed{\mathsf{head}(\mathit{fact}) \;:\; \mathit{head}}$

$$\mathsf{head}(\mathit{head} \mapsto \nu) \overset{\mathsf{def}}{=} \mathit{head}$$

Instance Head Avoidance

$\boxed{\mathit{head} \,\#\, \mathit{head}}$

$$\big(\mathit{kenv}_1.([\nu_1]C_1\ \overline{\mathit{typ}_1})\big) \# \big(\mathit{kenv}_2.([\nu_2]C_2\ \overline{\mathit{typ}_2})\big) \overset{\mathsf{def}}{\Leftrightarrow} \begin{cases} \text{true} & \nu_1 \neq \nu_2 \vee C_1 \neq C_2 \\ \forall i : \neg\mathsf{unify}(\mathit{typ}_{1,i}; \mathit{typ}_{2,i}) & \text{otherwise} \end{cases}$$

World (Shape) Consistency

$\boxed{\mathsf{consistent}(\omega)}$ $\boxed{\mathsf{consistent}(\widehat{\omega})}$

$$\begin{aligned} \mathsf{consistent}(\omega) &\overset{\mathsf{def}}{\Leftrightarrow} \forall \mathit{fact}_1, \mathit{fact}_2 \in \omega : \ \mathsf{head}(\mathit{fact}_1) \# \mathsf{head}(\mathit{fact}_2) \vee \mathit{fact}_1 = \mathit{fact}_2 \\ \mathsf{consistent}(\widehat{\omega}) &\overset{\mathsf{def}}{\Leftrightarrow} \text{likewise} \end{aligned}$$

World (Shape) Merging

$\boxed{\omega \oplus \omega \;:_{\mathsf{par}}\; \omega}$ $\boxed{\widehat{\omega} \oplus \widehat{\omega} \;:\; \widehat{\omega}}$

$$\{\!|\overline{\mathit{fact}_1}|\!\} \oplus \{\!|\overline{\mathit{fact}_2}|\!\} \overset{\mathsf{def}}{=} (\overline{\mathit{fact}_1}) \cup (\overline{\mathit{fact}_2})$$
$$\text{if} \quad \forall \mathit{fact}_1 \in \overline{\mathit{fact}_1}, \mathit{fact}_2 \in \overline{\mathit{fact}_2} : \ \mathsf{head}(\mathit{fact}_1) \# \mathsf{head}(\mathit{fact}_2) \vee \mathit{fact}_1 = \mathit{fact}_2$$

$$\{\!|\overline{\mathit{fact}_1}|\!\} \oplus \{\!|\overline{\mathit{fact}_2}|\!\} \overset{\mathsf{def}}{=} (\overline{\mathit{fact}_1}) \cup (\overline{\mathit{fact}_2})$$

World Extension

$\boxed{\omega \sqsupseteq \omega}$

$$\omega' \sqsupseteq \omega \overset{\mathsf{def}}{\Leftrightarrow} \exists \omega_F : \ \omega' = \omega \oplus \omega_F$$

Identity Substitution on Worlds

$\boxed{\mathsf{apply}(\theta; \omega) \;:_{\mathsf{par}}\; \omega}$

$$\mathsf{apply}(\theta; \{\!|\overline{\mathit{fact}}|\!\}) \overset{\mathsf{def}}{=} \{\!|\overline{\theta \mathit{fact}}|\!\} \qquad \text{if} \quad \forall \mathit{fact}_1, \mathit{fact}_2 \in \overline{\mathit{fact}} \ : \ \theta\mathsf{head}(\mathit{fact}_1) \# \theta\mathsf{head}(\mathit{fact}_2) \vee \theta\mathit{fact}_1 = \theta\mathit{fact}_2$$

Figure 7.5: Auxiliary definitions for worlds in Backpack. A couple notes: unify is core-level (*i.e.*, Haskell-level) unification of types; and ⊎ is the conventional (partial) union on finite mappings.

$$
\begin{array}{rcl}
S_{\mathsf{Set}} & = & (\texttt{import}\ \mathsf{Prelude};\ \overline{\mathit{decl}_{\mathsf{Set}}}) \\
\text{where}\quad \overline{\mathit{decl}_{\mathsf{Set}}} & = & \left(\begin{array}{l}
\texttt{data Set a} \\
\\
\texttt{empty :: Set a} \\
\texttt{member :: Ord a => a -> Set a -> Bool} \\
\texttt{insert :: Ord a => a -> Set a -> Set a} \\
\texttt{toAscList :: Set a -> [a]} \\
\\
\texttt{instance Eq a => Eq (Set a)} \\
\texttt{instance Ord a => Ord (Set a)}
\end{array}\right)
\end{array}
$$

(a) The definition of a full signature for Data.Set: an import to Prelude followed by the same declarations that were presented in Figure 6.3.

$$
\begin{array}{rcl}
\rho_{\mathsf{Set}} & = & \widehat{\tau}_0 \mathbin{@} \widehat{\omega}_0 \\
\text{where} & & \left\{\begin{array}{l}
\widehat{\tau}_0 = \langle\ \cdot\ ;\ \left(\begin{array}{l}
[\beta_1]\texttt{Set} \\
[\beta_2]\texttt{empty} \\
[\beta_4]\texttt{member} \\
[\beta_3]\texttt{insert} \\
[\beta_5]\texttt{toAscList}
\end{array}\right)\ ;\ \cdot\ \rangle \\
\widehat{\omega}_0 = \left\{\begin{array}{lcl}
 & \cdots & \\
(a :: *).[\nu_{\mathsf{P}}]\texttt{Eq}\ ([\beta_1]\texttt{Set}\ a) & \mapsto & \beta_6 \\
(a :: *).[\nu_{\mathsf{P}}]\texttt{Ord}\ ([\beta_1]\texttt{Set}\ a) & \mapsto & \beta_7 \\
 & \cdots &
\end{array}\right\}
\end{array}\right.
\end{array}
$$

(b) An example of a particular realizer that determines the identities for all the declarations in the signature.

$$
\rho_{\mathsf{Set}};\ \mathit{eenv} \vdash \overline{\mathit{decl}_{\mathsf{Set}}} \rightsquigarrow \overline{\nu_{\mathsf{Set}}}
$$

$$
\text{where}\quad \overline{\nu_{\mathsf{Set}}} = \left(\begin{array}{l}
\beta_1 \\ \beta_2 \\ \beta_4 \\ \beta_3 \\ \beta_5 \\ \beta_6 \\ \beta_7
\end{array}\right)
\begin{array}{l}
\texttt{data Set a} \\
\texttt{empty :: Set a} \\
\texttt{member :: Ord a => a -> Set a -> Bool} \\
\texttt{insert :: Ord a => a -> Set a -> Set a} \\
\texttt{toAscList :: Set a -> [a]} \\
\texttt{instance Eq a => Eq (Set a)} \\
\texttt{instance Ord a => Ord (Set a)}
\end{array}
$$

(c) The result of realizing the signature declarations with ρ_{Set} and some related entity environment *eenv*: the seven provenances of the declared entities.

Figure 7.6: A reproduction of the declarations of a typical Data.Set signature first presented in Figure 6.3, followed by an example of a realizer that identifies the provenance of every entity declared in the signature.

Core Environment Construction

Environment construction

$$\boxed{\widehat{\Gamma} \Vdash \mathit{impdecls}; \mathit{ibnds} \rightsquigarrow \mathit{eenv} \mathbin{@} \widehat{\omega}} \quad \boxed{\Gamma \vdash \mathit{impdecls}; \mathit{rbnds} \rightsquigarrow \mathit{eenv} \mathbin{@} \omega}$$

$$\frac{\begin{array}{c}\forall i \in [1..n] : \widehat{\Gamma} \Vdash \mathit{impdecl}_i \rightsquigarrow \mathit{eenv}_i \\ \begin{array}{rcll}\mathit{eenv} & = & \bigoplus_{i\in[1..n]} \mathit{eenv}_i & \oplus \;\; \mathsf{mkloceenv}(\mathit{ibnds}) \\ \widehat{\omega} & = & \bigoplus_{i\in[1..n]} \mathsf{world}_{\widehat{\Gamma}}(\mathsf{imp}(\mathit{impdecl}_i)) & \oplus \;\; \mathsf{mkloc\widehat{w}orld}(\mathit{ibnds};\ \mathit{eenv})\end{array}\end{array}}{\widehat{\Gamma} \Vdash \mathit{impdecl}_1, \ldots, \mathit{impdecl}_n; \mathit{ibnds} \rightsquigarrow \mathit{eenv} \mathbin{@} \widehat{\omega}}\ (\textsc{CoreEnvSh})$$

$$\frac{\begin{array}{c}\forall i \in [1..n] : \mathsf{shape}(\Gamma) \Vdash \mathit{impdecl}_i \rightsquigarrow \mathit{eenv}_i \\ \begin{array}{rcll}\mathit{eenv} & = & \bigoplus_{i\in[1..n]} \mathit{eenv}_i & \oplus \;\; \mathsf{mkloceenv}(\mathit{rbnds}) \\ \omega & = & \bigoplus_{i\in[1..n]} \mathsf{world}_{\Gamma}(\mathsf{imp}(\mathit{impdecl}_i)) & \oplus \;\; \mathsf{mklocworld}(\mathit{rbnds};\ \mathit{eenv})\end{array}\end{array}}{\Gamma \vdash \mathit{impdecl}_1, \ldots, \mathit{impdecl}_n; \mathit{rbnds} \rightsquigarrow \mathit{eenv} \mathbin{@} \omega}\ (\textsc{CoreEnv})$$

Import resolution

$$\boxed{\widehat{\Gamma} \Vdash \mathit{impdecl} \rightsquigarrow \mathit{eenv}} \quad \boxed{\widehat{\Gamma}; \ell \Vdash \mathit{impspec} \rightsquigarrow \mathit{espcs}} \quad \boxed{\widehat{\Gamma}; \ell \Vdash \mathit{import} \rightsquigarrow \mathit{espc}}$$

$$\frac{\widehat{\Gamma}; \ell \Vdash \mathit{impspec} \rightsquigarrow \mathit{espcs} \qquad \mathit{eenv}_{\mathsf{base}} = \mathsf{mkeenv}(\mathit{espcs}) \qquad \mathit{eenv}_{\mathsf{qual}} = \mathsf{qualify}(\ell[']; \mathit{eenv}_{\mathsf{base}})}{\widehat{\Gamma} \Vdash (\mathtt{import}\ \ell\ [\mathtt{as}\ \ell']\ \mathit{impspec}) \rightsquigarrow (\mathit{eenv}_{\mathsf{base}} \oplus \mathit{eenv}_{\mathsf{qual}})}\ (\textsc{ImpDeclUnqual})$$

$$\frac{\widehat{\Gamma}; \ell \Vdash \mathit{impspec} \rightsquigarrow \mathit{espcs} \qquad \mathit{eenv}_{\mathsf{base}} = \mathsf{mkeenv}(\mathit{espcs}) \qquad \mathit{eenv}_{\mathsf{qual}} = \mathsf{qualify}(\ell[']; \mathit{eenv}_{\mathsf{base}})}{\widehat{\Gamma} \Vdash (\mathtt{import\ qualified}\ \ell\ [\mathtt{as}\ \ell']\ \mathit{impspec}) \rightsquigarrow \mathit{eenv}_{\mathsf{qual}}}\ (\textsc{ImpDeclQual})$$

$$\frac{\mathit{espcs} \in \widehat{\Gamma}(\ell)}{\widehat{\Gamma}; \ell \Vdash \cdot \rightsquigarrow \mathit{espcs}}\ (\textsc{ImpSpecAll})$$

$$\frac{\forall i \in [1..n] : \widehat{\Gamma}; \ell \Vdash \mathit{import}_i \rightsquigarrow \mathit{espc}_i \qquad \mathit{espcs} = \bigoplus_{i\in[1..n]}\{\mathit{espc}_i\}}{\widehat{\Gamma}; \ell \Vdash \mathtt{(}\mathit{import}_1, \ldots, \mathit{import}_n\mathtt{)} \rightsquigarrow \mathit{espcs}}\ (\textsc{ImpSpecSome})$$

$$\frac{\mathit{espc} \in \widehat{\Gamma}(\ell) \qquad \mathit{espc} = [\nu]\chi}{\widehat{\Gamma}; \ell \Vdash \chi \rightsquigarrow \mathit{espc}}\ (\textsc{ImpSing}) \qquad \frac{\mathit{espc} \in \widehat{\Gamma}(\ell) \qquad \mathit{espc} \leqslant \mathit{espc}' = [\nu]\chi()}{\widehat{\Gamma}; \ell \Vdash \chi \rightsquigarrow \mathit{espc}'}\ (\textsc{ImpSingParent})$$

$$\frac{\mathit{espc} \in \widehat{\Gamma}(\ell) \qquad \mathit{espc} = [\nu]\chi(\overline{\chi}')}{\widehat{\Gamma}; \ell \Vdash \chi\mathtt{(..)} \rightsquigarrow \mathit{espc}}\ (\textsc{ImpMultAll}) \qquad \frac{\mathit{espc} \in \widehat{\Gamma}(\ell) \qquad \mathit{espc} \leqslant \mathit{espc}' = [\nu]\chi(\overline{\chi}')}{\widehat{\Gamma}; \ell \Vdash \chi\mathtt{(}\overline{\chi}'\mathtt{)} \rightsquigarrow \mathit{espc}'}\ (\textsc{ImpMultSome})$$

Entity Environment Qualification

$$\boxed{\mathsf{qualify}(\mathit{mref};\ \mathit{eenv}) :_{\mathsf{par}} \mathit{eenv}}$$

$$\begin{array}{rcl}\mathsf{qualify}(\mathit{mref};\ \mathit{eenv}) & \overset{\mathsf{def}}{=} & \{\mathit{mref}\mathtt{.}\chi \mapsto \mathit{phnm} \mid \chi \mapsto \mathit{phnm} \in \mathit{eenv}\}; \mathsf{locals}(\mathit{eenv}) \\ & & \text{where } \forall \mathit{eref} \in \mathsf{dom}(\mathit{eenv}) : \exists \chi : \mathit{eref} = \chi\end{array}$$

Figure 7.7: Definition of core environment construction and import resolution. Note that the rules for resolving an *impdecls* employ bracketing ([]) to denote rule *schemas*. As a result each of these rules is interpreted as two distinct rules: with and without the parts in brackets.

Core Environment Construction (Continued)

Local Entity Environment Construction

$\boxed{\mathsf{mkloceenv}(\mathit{rbnds}) \;:_{\mathsf{par}}\; \mathit{eenv}}$ $\boxed{\mathsf{mkeenv}(\mathit{espc}) \;:_{\mathsf{par}}\; \mathit{eenv}}$ $\boxed{\mathsf{mkeenv}(\mathit{espcs}) \;:_{\mathsf{par}}\; \mathit{eenv}}$ $\boxed{\mathsf{locespcs}(\mathit{rbnds}) \;:_{\mathsf{par}}\; \mathit{espcs}}$

$$\begin{aligned}
\mathsf{mkloceenv}(\mathit{rbnds}) \;\overset{\text{def}}{=}\;& \mathsf{mkeenv}(\mathsf{locespcs}(\mathit{rbnds})) \\
& \oplus\, \mathsf{qualify}(\texttt{Local};\ \mathsf{mkeenv}(\mathsf{locespcs}(\mathit{rbnds})))
\end{aligned}$$

$$\begin{aligned}
\mathsf{mkeenv}([\nu]\chi) \;&\overset{\text{def}}{=}\; \{\chi \mapsto [\nu]\chi\};\ [\nu]\chi \\
\mathsf{mkeenv}([\nu]\chi(\overline{\chi'})) \;&\overset{\text{def}}{=}\; \{\chi \mapsto [\nu]\chi\} \cup \{\chi' \mapsto [\nu]\chi' \mid \chi' \in \overline{\chi'}\};\ [\nu]\chi(\overline{\chi'}) \\
\mathsf{mkeenv}(\mathit{espcs}) \;&\overset{\text{def}}{=}\; \textstyle\bigoplus_{\mathit{espc}\in\mathit{espcs}} \mathsf{mkeenv}(\mathit{espc})
\end{aligned}$$

$$\begin{aligned}
\mathsf{locespcs}((\nu_1,\ldots,\nu_n \mid \mathit{bnd}_1,\ldots,\mathit{bnd}_n)) \;&\overset{\text{def}}{=}\; (\mathit{espc} \mid \exists i \in [1..n] : \mathit{bnd}_i \sqsubseteq_{\nu_i} \mathit{espc}) \\
\mathsf{locespcs}((\widehat{\tau}_0 \mathbin{@} \_ \mid \mathit{decls})) \;&\overset{\text{def}}{=}\; (\mathit{espc}' \mid \mathit{espc} \in \widehat{\tau}_0,\ \exists \mathit{decl} \in \mathit{decls} : \mathit{decl} \sqsubseteq \mathit{espc}' \leqslant \mathit{espc})
\end{aligned}$$

Local World Construction

$\boxed{\mathsf{mklocworld}(\mathit{rbnds};\ \mathit{eenv}) \;:_{\mathsf{par}}\; \omega}$ $\boxed{\mathsf{mkloc\hat{w}orld}(\mathit{ibnds};\ \mathit{eenv}) \;:_{\mathsf{par}}\; \widehat{\omega}}$

$$\mathsf{mklocworld}((\_ \mathbin{@} \widehat{\omega}_0 \mid \mathit{decl}_1,\ldots,\mathit{decl}_n);\ \mathit{eenv}) \;\overset{\text{def}}{=}\; \{\!|\, \mathit{eenv}(\mathsf{head}(\mathit{decl}_i)) \mapsto \nu \;\in\; \widehat{\omega}_0 \mid i \in \mathcal{I} \,|\!\},$$
$$\text{where} \begin{cases} \mathsf{consistent}(\widehat{\omega}_0) \\ \mathcal{I} = \{i \in [1..n] \mid \mathit{decl}_i = \texttt{instance}\ldots\} \\ \forall i \in \mathcal{I} : \mathit{eenv}(\mathsf{head}(\mathit{decl}_i)) \text{ defined} \\ \forall i \neq j \in \mathcal{I} : \mathit{eenv}(\mathsf{head}(\mathit{decl}_i)) \mathbin{\#} \mathit{eenv}(\mathsf{head}(\mathit{decl}_j)) \end{cases}$$

$$\mathsf{mklocworld}((\nu_1,\ldots,\nu_n \mid \mathit{bnd}_1,\ldots,\mathit{bnd}_n);\ \mathit{eenv}) \;\overset{\text{def}}{=}\; \{\!|\, \mathit{eenv}(\mathsf{head}(\mathit{bnd}_i)) \mapsto \nu_i \mid i \in \mathcal{I} \,|\!\},$$
$$\text{where} \begin{cases} \mathcal{I} = \{i \in [1..n] \mid \mathit{bnd}_i = \texttt{instance}\ldots\} \\ \forall i \in \mathcal{I} : \mathit{eenv}(\mathsf{head}(\mathit{bnd}_i)) \text{ defined} \\ \forall i \neq j \in \mathcal{I} : \mathit{eenv}(\mathsf{head}(\mathit{bnd}_i)) \mathbin{\#} \mathit{eenv}(\mathsf{head}(\mathit{bnd}_j)) \end{cases}$$

$$\mathsf{mkloc\hat{w}orld}((\nu_1,\ldots,\nu_n \mid \mathit{bnd}_1,\ldots,\mathit{bnd}_n);\ \mathit{eenv}) \;\overset{\text{def}}{=}\; \text{(similar)}$$

Figure 7.8: Definition of core environment construction, continued.

Module Export Resolution

Module Export Declarations Resolution

$$\boxed{\mathit{eenv} \Vdash \mathit{expdecl} \rightsquigarrow \mathit{espcs}}$$

$$\frac{\overline{\mathit{phnm}} = ([\nu]\chi \mid \texttt{Local}.\chi \mapsto [\nu]\chi \in \mathit{eenv})}{\mathit{eenv} \Vdash \cdot \rightsquigarrow \mathsf{filterespcs}(\mathsf{locals}(\mathit{eenv}); \overline{\mathit{phnm}})}\ (\textsc{ExpLocal})$$

$$\frac{\forall i \in [1..n] : \mathit{eenv} \Vdash \mathit{export}_i \rightsquigarrow \mathit{espc}_i \qquad \mathit{espcs} = \bigoplus_{i \in [1..n]} \{\mathit{espc}_i\} \quad \mathsf{nooverlap}(\mathit{espcs})}{\mathit{eenv} \Vdash \texttt{export (}\mathit{export}_1, \ldots, \mathit{export}_n\texttt{)} \rightsquigarrow \mathit{espcs}}\ (\textsc{ExpList})$$

Entity Export Resolution

$$\boxed{\mathit{eenv} \Vdash \mathit{export} \rightsquigarrow \mathit{espc}}$$

$$\frac{\mathit{eenv}(\mathit{eref}) = [\nu]\chi : [\nu]\chi}{\mathit{eenv} \Vdash \mathit{eref} \rightsquigarrow [\nu]\chi}\ (\textsc{ExpSimple}) \qquad \frac{\mathit{eenv}(\mathit{eref}) = [\nu]\chi : [\nu]\chi(\overline{\chi'})}{\mathit{eenv} \Vdash \mathit{eref} \rightsquigarrow [\nu]\chi()}\ (\textsc{ExpSimpleEmpty})$$

$$\frac{\mathit{eenv}(\mathit{eref}) = [\nu]\chi : [\nu]\chi(\overline{\chi'}, \overline{\chi''})}{\mathit{eenv} \Vdash \mathit{eref}\texttt{(}\overline{\chi'}\texttt{)} \rightsquigarrow [\nu]\chi(\overline{\chi'})}\ (\textsc{ExpSubList}) \qquad \frac{\mathit{eenv}(\mathit{eref}) = [\nu]\chi : [\nu]\chi(\overline{\chi'})}{\mathit{eenv} \Vdash \mathit{eref}\texttt{(..)} \rightsquigarrow [\nu]\chi(\overline{\chi'})}\ (\textsc{ExpSubAll})$$

$$\frac{\overline{\mathit{phnm}} = ([\nu]\chi \mid \chi \mapsto [\nu]\chi \in \mathit{eenv},\ \mathit{mref}\texttt{.}\chi \mapsto [\nu]\chi \in \mathit{eenv}) \qquad \forall [\nu]\chi \in \overline{\mathit{phnm}} : \mathit{eenv}(\chi) = [\nu]\chi \wedge \mathit{eenv}(\mathit{mref}\texttt{.}\chi) = [\nu]\chi}{\mathit{eenv} \Vdash \texttt{scope}\ \mathit{mref} \rightsquigarrow \mathsf{filterespcs}(\mathsf{locals}(\mathit{eenv}); \overline{\mathit{phnm}})}\ (\textsc{ExpModAll})$$

$$\begin{aligned}
\mathsf{filterespc}([\nu]\chi; \overline{\mathit{phnm}}) &\overset{\text{def}}{=} [\nu]\chi \\
\mathsf{filterespc}([\nu]\chi(\overline{\chi'}); \overline{\mathit{phnm}}) &\overset{\text{def}}{=} [\nu]\chi(\overline{\chi''}) \quad \text{where } \{\overline{[\nu]\chi''}\} = \{\overline{[\nu]\chi'}\} \cap \{\overline{\mathit{phnm}}\} \\
\mathsf{filterespcs}(\overline{\mathit{espc}}; \overline{\mathit{phnm}}) &\overset{\text{def}}{=} \left\{ \mathsf{filterespc}(\mathit{espc}; \overline{\mathit{phnm}}) \;\middle|\; \begin{array}{l} \mathit{espc} \in \overline{\mathit{espc}}, \\ \mathit{phnm} \in \overline{\mathit{phnm}}, \\ \mathit{espc} \sqsubseteq \mathit{phnm} \end{array} \right\}
\end{aligned}$$

Figure 7.9: Definition of module export resolution.

Module-Level Shaping and Typing

Module shaping

$$\boxed{\widehat{\Gamma};\, \nu_0 \Vdash M \Rightarrow \widehat{\tau} @ \widehat{\omega}}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \Vdash \mathit{impdecls}; (\nu_0, \ldots, \nu_0 \mid \mathit{defs}) \rightsquigarrow \mathit{eenv} @ \widehat{\omega} \qquad \mathit{eenv} \Vdash \mathit{expdecl} \rightsquigarrow \mathit{espcs} \\ \mathit{defs} \sqsubseteq \widehat{\mathit{dspcs}} \qquad N = \{\widehat{\Gamma}(\mathsf{imp}(\mathit{impdecl})) \mid \mathit{impdecl} \in \mathit{impdecls}\}\end{array}}{\widehat{\Gamma};\, \nu_0 \Vdash (\mathit{impdecls}; \mathit{expdecl}; \mathit{defs}) \Rightarrow \langle\, \widehat{\mathit{dspcs}} \,;\, \mathit{espcs} \,;\, N \,\rangle @ \widehat{\omega}} \text{(ShMod)}$$

Module typing

$$\boxed{\Gamma;\, \nu_0 \vdash M : \tau @ \omega}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{impdecls}; (\nu_0, \ldots, \nu_0 \mid \mathit{defs}) \rightsquigarrow \mathit{eenv} @ \omega \qquad \mathit{eenv} \Vdash \mathit{expdecl} \rightsquigarrow \mathit{espcs} \\ \Gamma.\Phi;\, \nu_0;\, \mathit{eenv};\, \omega \vdash \mathit{defs} : \mathit{dspcs} \qquad N = \{\Gamma(\mathsf{imp}(\mathit{impdecl})) \mid \mathit{impdecl} \in \mathit{impdecls}\}\end{array}}{\Gamma;\, \nu_0 \vdash (\mathit{impdecls}; \mathit{expdecl}; \mathit{defs}) : \langle\!\langle\, \mathit{dspcs} \,;\, \mathit{espcs} \,;\, N \,\rangle\!\rangle @ \omega} \text{(TyMod)}$$

Signature shaping

$$\boxed{\widehat{\Gamma};\, \overline{\nu} \Vdash S \Rightarrow \widehat{\sigma} @ \widehat{\omega} \mid \widehat{\Phi}_{\mathsf{sig}}}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \Vdash \mathit{impdecls}; (\overline{\nu} \mid \mathit{decls}) \rightsquigarrow \mathit{eenv} @ \widehat{\omega} \\ \mathit{decls} \sqsubseteq \widehat{\mathit{dspcs}} \qquad \widehat{\mathit{dspcs}} \sqsubseteq_{\overline{\nu}} \mathit{espcs} \qquad \widehat{\Phi}_{\mathsf{sig}} = \widehat{\mathsf{sigenv}}(\overline{\nu};\, \widehat{\mathit{dspcs}};\, \widehat{\omega})\end{array}}{\widehat{\Gamma};\, \overline{\nu} \Vdash (\mathit{impdecls}; \mathit{decls}) \Rightarrow \langle\, \cdot \,;\, \mathit{espcs} \,;\, \cdot \,\rangle @ \widehat{\omega} \mid \widehat{\Phi}_{\mathsf{sig}}} \text{(ShSig)}$$

Signature typing

$$\boxed{\Gamma;\, \rho \vdash S : \sigma @ \omega \mid \Phi_{\mathsf{sig}}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{impdecls}; (\rho \mid \mathit{decls}) \rightsquigarrow \mathit{eenv} @ \omega \qquad \rho;\, \mathit{eenv} \vdash \mathit{decls} \rightsquigarrow \overline{\nu} \\ \Gamma.\Phi;\, \overline{\nu};\, \mathit{eenv};\, \omega \vdash \mathit{decls} : \mathit{dspcs} \qquad \mathit{dspcs} \sqsubseteq_{\overline{\nu}} \mathit{espcs} \qquad \Phi_{\mathsf{sig}} = \mathsf{sigenv}(\overline{\nu};\, \mathit{dspcs};\, \omega)\end{array}}{\Gamma;\, \rho \vdash (\mathit{impdecls}; \mathit{decls}) : \langle\!\langle\, \cdot \,;\, \mathit{espcs} \,;\, \cdot \,\rangle\!\rangle @ \omega \mid \Phi_{\mathsf{sig}}} \text{(TySig)}$$

Signature declaration context construction

$$\boxed{\widehat{\mathsf{sigenv}}(\overline{\nu};\, \widehat{\mathit{dspcs}};\, \widehat{\omega}) :_{\mathsf{par}} \widehat{\Phi}} \qquad \boxed{\mathsf{sigenv}(\overline{\nu};\, \mathit{dspcs};\, \omega) :_{\mathsf{par}} \Phi}$$

$$\begin{array}{lcl}\mathsf{sigenv}(\nu_1, \ldots, \nu_n;\, \mathit{dspc}_1, \ldots, \mathit{dspc}_n;\, \omega) & \overset{\mathsf{def}}{=} & \bigoplus\, (\nu_i{:}\langle\!\langle\, \mathit{dspc}_i \,;\, \mathit{espc} \,;\, \cdot \,\rangle\!\rangle^{-} @ \omega \mid i \in [1..n],\, \mathit{dspc}_i \sqsubseteq_{\nu_i} \mathit{espc}) \\ \widehat{\mathsf{sigenv}}(\nu_1, \ldots, \nu_n;\, \widehat{\mathit{dspc}}_1, \ldots, \widehat{\mathit{dspc}}_n;\, \widehat{\omega}) & \overset{\mathsf{def}}{=} & \text{(similar))}\end{array}$$

Signature declaration realization

$$\boxed{\rho;\, \mathit{eenv} \vdash \mathit{decl} \rightsquigarrow \nu} \qquad \boxed{\rho;\, \mathit{eenv} \vdash \mathit{decls} \rightsquigarrow \overline{\nu}}$$

$$\frac{\mathit{decl} \sqsubseteq_{\nu} \mathit{espc} \geqslant \mathit{espc}' \qquad \mathit{espc}' \in \widehat{\tau}_0}{\widehat{\tau}_0 @ \widehat{\omega}_0;\, \mathit{eenv} \vdash \mathit{decl} \rightsquigarrow \nu} (1) \qquad \frac{\mathit{eenv}(\mathsf{head}(\mathit{decl})) \mapsto \nu \in \widehat{\omega}_0}{\widehat{\tau}_0 @ \widehat{\omega}_0;\, \mathit{eenv} \vdash \mathit{decl} \rightsquigarrow \nu} (2) \qquad \frac{\overline{\rho;\, \mathit{eenv} \vdash \mathit{decl} \rightsquigarrow \nu}}{\rho;\, \mathit{eenv} \vdash \overline{\mathit{decl}} \rightsquigarrow \overline{\nu}} (3)$$

Figure 7.10: Definition of shaping and typing on module level.

$$\Phi_{\mathsf{sig}} \;=$$

$$\left\{\!\!\left|\begin{array}{lllllllll}
\beta_1 & : & \llparenthesis & \texttt{data}\;\texttt{Set}\;a\;\texttt{::}\;* & ; & [\beta_1]\texttt{Set}() & ; & \cdot & \rrparenthesis^{-} @\, \omega \\
\beta_2 & : & \llparenthesis & \texttt{empty}\;\texttt{::}\;\texttt{forall}\;(a\;\texttt{::}\;*).\;\mathit{typ} & ; & [\beta_2]\texttt{empty} & ; & \cdot & \rrparenthesis^{-} @\, \omega \\
\beta_3 & : & \llparenthesis & \texttt{member}\;\texttt{::}\;\texttt{forall}\;(a\;\texttt{::}\;*).\;[\nu_{\mathsf{P}}]\texttt{Ord}\;a\;\texttt{=>}\;a\;\texttt{->}\;\mathit{typ}\;\texttt{->}\;[\nu_{\mathsf{P}}]\texttt{Bool} & ; & [\beta_3]\texttt{member} & ; & \cdot & \rrparenthesis^{-} @\, \omega \\
\beta_4 & : & \llparenthesis & \texttt{insert}\;\texttt{::}\;\texttt{forall}\;(a\;\texttt{::}\;*).\;[\nu_{\mathsf{P}}]\texttt{Ord}\;a\;\texttt{=>}\;a\;\texttt{->}\;\mathit{typ}\;\texttt{->}\;\mathit{typ} & ; & [\beta_4]\texttt{insert} & ; & \cdot & \rrparenthesis^{-} @\, \omega \\
\beta_5 & : & \llparenthesis & \texttt{toAscList}\;\texttt{::}\;\texttt{forall}\;(a\;\texttt{::}\;*).\;\mathit{typ}\;\texttt{->}\;[\nu_{\mathsf{P}}]\texttt{List}\;a & ; & [\beta_5]\texttt{toAscList} & ; & \cdot & \rrparenthesis^{-} @\, \omega \\
\beta_6 & : & \llparenthesis & \texttt{instance}\;(a\;\texttt{::}\;*)\;\{[\nu_{\mathsf{P}}]\texttt{Eq}\;a\}\;[\nu_{\mathsf{P}}]\texttt{Eq}\;\mathit{typ} & ; & \cdot & ; & \cdot & \rrparenthesis^{-} @\, \omega \\
\beta_7 & : & \llparenthesis & \texttt{instance}\;(a\;\texttt{::}\;*)\;\{[\nu_{\mathsf{P}}]\texttt{Ord}\;a\}\;[\nu_{\mathsf{P}}]\texttt{Ord}\;\mathit{typ} & ; & \cdot & ; & \cdot & \rrparenthesis^{-} @\, \omega
\end{array}\right|\!\!\right\}$$

$$\text{where}\;\left\{\begin{array}{l}
\mathit{typ} = [\beta_1]\texttt{Set}\;a \\
\omega = \omega_{\mathsf{imp}} \;\oplus\; \left\{\!\!\left|\begin{array}{ll}
(a\;\texttt{::}\;*)\,.\,[\nu_{\mathsf{P}}]\texttt{Eq}\;\mathit{typ} & \mapsto \beta_6 \\
(a\;\texttt{::}\;*)\,.\,[\nu_{\mathsf{P}}]\texttt{Ord}\;\mathit{typ} & \mapsto \beta_7
\end{array}\right|\!\!\right\}
\end{array}\right.$$

Figure 7.11: The signature declaration context for the Set signature and realizer from Figure 7.6.

Well-Formedness at the Module Level

Module types and shapes

$$\boxed{\Phi \vdash \tau\ \mathsf{wf}} \quad \boxed{\widehat{\Phi} \Vdash \widehat{\tau}\ \mathsf{wf}}$$

$$\frac{\begin{array}{c}\forall dspc \in dspcs : \Phi \vdash dspc\ \mathsf{wf}\\ \forall espc \in espcs : \Phi \vdash espc\ \mathsf{wf}\\ \{\overline{\nu}\} \subseteq \mathsf{dom}(\Phi)\end{array}}{\Phi \vdash \langle\!|\, dspcs\, ;\, espcs\, ;\, \overline{\nu}\, |\!\rangle\ \mathsf{wf}}\ (\textsc{WfModTyp}) \qquad \frac{\begin{array}{c}\forall espc \in espcs : \widehat{\Phi} \Vdash espc\ \mathsf{wf}\\ \{\overline{\nu}\} \subseteq \mathsf{dom}(\widehat{\Phi})\end{array}}{\widehat{\Phi} \Vdash \langle\, \widehat{dspcs}\, ;\, espcs\, ;\, \overline{\nu}\, \rangle\ \mathsf{wf}}\ (\textsc{WfModShp})$$

Worlds and world shapes

$$\boxed{\Phi \vdash \omega\ \mathsf{wf}} \quad \boxed{\widehat{\Phi} \Vdash \widehat{\omega}\ \mathsf{wf}}$$

$$\frac{\overline{\Phi \vdash head \mapsto \nu\ \mathsf{wf}}}{\Phi \vdash \{\!|\, \overline{head \mapsto \nu}\, |\!\}\ \mathsf{wf}}\ (\textsc{WfWorld}) \qquad \frac{\overline{\widehat{\Phi} \Vdash head \mapsto \nu\ \mathsf{wf}}}{\widehat{\Phi} \Vdash \{\, \overline{head \mapsto \nu}\, \}\ \mathsf{wf}}\ (\textsc{WfWorldShp})$$

World facts

$$\boxed{\Phi \vdash head \mapsto \nu\ \mathsf{wf}} \quad \boxed{\widehat{\Phi} \Vdash head \mapsto \nu\ \mathsf{wf}}$$

$$\frac{\nu{:}\tau^{m}@\,\omega \in \Phi \qquad dspc \in \tau \qquad head = \mathsf{head}(dspc)}{\Phi \vdash head \mapsto \nu\ \mathsf{wf}}\ (\textsc{WfFact}) \qquad \frac{\nu{:}\widehat{\tau}^{m}@\,\widehat{\omega} \in \widehat{\Phi} \qquad head \mapsto \nu \in \widehat{\omega}}{\widehat{\Phi} \Vdash head \mapsto \nu\ \mathsf{wf}}\ (\textsc{WfFactShp})$$

Module contexts

$$\boxed{\Phi \vdash \Gamma\ \mathsf{wf}} \quad \boxed{\widehat{\Phi} \Vdash \widehat{\Gamma}\ \mathsf{wf}}$$

$$\frac{\begin{array}{c}\Phi \ \oplus_{?}\ \Phi'\\ \Phi \vdash \Phi'\ \mathsf{wf} \qquad \Phi \oplus \Phi' \vdash \mathcal{L}'\ \mathsf{wf}\end{array}}{\Phi \vdash (\!|\, \Phi' ; \mathcal{L}'\, |\!)\ \mathsf{wf}}\ (\textsc{WfModCtx}) \qquad \frac{\begin{array}{c}\widehat{\Phi} \ \oplus_{?}\ \widehat{\Phi}'\\ \widehat{\Phi} \Vdash \widehat{\Phi}'\ \mathsf{wf} \qquad \widehat{\Phi} \oplus \widehat{\Phi}' \Vdash \mathcal{L}'\ \mathsf{wf}\end{array}}{\widehat{\Phi} \Vdash (\, \widehat{\Phi}' ; \mathcal{L}'\,)\ \mathsf{wf}}\ (\textsc{WfModShpCtx})$$

Logical module contexts

$$\boxed{\Phi \vdash \mathcal{L}\ \mathsf{wf}} \quad \boxed{\widehat{\Phi} \Vdash \mathcal{L}\ \mathsf{wf}}$$

$$\frac{\{\overline{\nu}\} \subseteq \mathsf{dom}(\Phi)}{\Phi \vdash \overline{\ell \mapsto \nu}\ \mathsf{wf}}\ (\textsc{WfLogModCtx}) \qquad \frac{\{\overline{\nu}\} \subseteq \mathsf{dom}(\widehat{\Phi})}{\widehat{\Phi} \Vdash \overline{\ell \mapsto \nu}\ \mathsf{wf}}\ (\textsc{WfLogModShpCtx})$$

Figure 7.12: Definition of well-formedness of some semantic objects relevant to module level. Well-formedness of physical module contexts (Φ) is defined in Figure 7.13.

Well-Formedness of Module Contexts

Module contexts

$\boxed{\Phi \vdash \Phi\ \mathsf{wf}}$ $\boxed{\widehat{\Phi} \Vdash \widehat{\Phi}\ \mathsf{wf}}$

$$\frac{\begin{array}{c}\Phi \oplus_{?} \Phi' \\ \Phi \oplus \Phi' \vdash \Phi'\ \mathsf{spcs\text{-}wf} \\ \Phi \oplus \Phi' \vdash \Phi'\ \mathsf{exps\text{-}wf} \\ \Phi \oplus \Phi' \vdash \Phi'\ \mathsf{imps\text{-}wf} \\ \Phi \oplus \Phi' \vdash \Phi'\ \mathsf{wlds\text{-}wf}\end{array}}{\Phi \vdash \Phi'\ \mathsf{wf}}\ (\textsc{WfPhModCtx}) \qquad \frac{\begin{array}{c}\widehat{\Phi} \oplus_{?} \widehat{\Phi}' \\ \widehat{\Phi} \oplus \widehat{\Phi}' \Vdash \widehat{\Phi}'\ \mathsf{spcs\text{-}wf} \\ \widehat{\Phi} \oplus \widehat{\Phi}' \Vdash \widehat{\Phi}'\ \mathsf{exps\text{-}wf} \\ \widehat{\Phi} \oplus \widehat{\Phi}' \Vdash \widehat{\Phi}'\ \mathsf{wlds\text{-}wf}\end{array}}{\widehat{\Phi} \Vdash \widehat{\Phi}'\ \mathsf{wf}}\ (\textsc{WfPhModCtxShp})$$

Entities well-formed in contexts

$\boxed{\Phi \vdash \Phi\ \mathsf{spcs\text{-}wf}}$ $\boxed{\widehat{\Phi} \Vdash \widehat{\Phi}\ \mathsf{spcs\text{-}wf}}$

$\boxed{\Phi \vdash \tau^{m}\ \mathsf{spcs\text{-}wf}}$ $\boxed{\widehat{\Phi} \Vdash \widehat{\tau}^{m}\ \mathsf{spcs\text{-}wf}}$

$$\frac{\overline{\Phi \vdash \tau^{m}\ \mathsf{spcs\text{-}wf}}}{\Phi \vdash \{\!|\,\overline{\nu{:}\tau^{m}@\,\omega}\,|\!\}\ \mathsf{spcs\text{-}wf}} \qquad \frac{\overline{\widehat{\Phi} \Vdash \widehat{\tau}^{m}\ \mathsf{spcs\text{-}wf}}}{\widehat{\Phi} \Vdash \{\,\overline{\nu{:}\widehat{\tau}^{m}@\,\widehat{\omega}}\,\}\ \mathsf{spcs\text{-}wf}}$$

$$\frac{\mathsf{nooverlap}(\mathit{dspcs}) \qquad \forall \mathit{dspc} \in \mathit{dspcs} :\ \ \Phi \vdash \mathit{dspc}\ \mathsf{wf} \wedge \mathsf{validspc}(\mathit{dspc}; m)}{\Phi \vdash \langle\!|\ \mathit{dspcs}\ ;\ \mathit{espcs}\ ;\ \overline{\nu}\ |\!\rangle^{m}\ \mathsf{spcs\text{-}wf}}\ (\textsc{WfSpecs})$$

$$\frac{\mathsf{nooverlap}(\mathit{ds\hat{p}cs}) \qquad \forall \mathit{ds\hat{p}c} \in \mathit{ds\hat{p}cs} :\ \ \mathsf{validspc}(\mathit{ds\hat{p}c}; m)}{\widehat{\Phi} \Vdash \langle\ \mathit{ds\hat{p}cs}\ ;\ \mathit{espcs}\ ;\ \overline{\nu}\ \rangle^{m}\ \mathsf{spcs\text{-}wf}}\ (\textsc{WfShSpecs})$$

Exports well-formed in contexts

$\boxed{\Phi \vdash \Phi\ \mathsf{exps\text{-}wf}}$ $\boxed{\widehat{\Phi} \Vdash \widehat{\Phi}\ \mathsf{exps\text{-}wf}}$

$\boxed{\Phi \vdash \tau\ \mathsf{exps\text{-}wf}}$ $\boxed{\widehat{\Phi} \Vdash \widehat{\tau}\ \mathsf{exps\text{-}wf}}$

$$\frac{\overline{\Phi \vdash \tau\ \mathsf{exps\text{-}wf}}}{\Phi \vdash \{\!|\,\overline{\nu{:}\tau^{m}@\,\omega}\,|\!\}\ \mathsf{exps\text{-}wf}} \qquad \frac{\mathsf{nooverlap}(\mathit{espcs}) \qquad \forall \mathit{espc} \in \mathit{espcs} :\ \Phi \vdash \mathit{espc}\ \mathsf{wf}}{\Phi \vdash \langle\!|\ \mathit{dspcs}\ ;\ \mathit{espcs}\ ;\ \overline{\nu}\ |\!\rangle\ \mathsf{exps\text{-}wf}}\ (\textsc{WfExports})$$

$$\frac{\overline{\widehat{\Phi} \Vdash \widehat{\tau}\ \mathsf{exps\text{-}wf}}}{\widehat{\Phi} \Vdash \{\,\overline{\nu{:}\widehat{\tau}^{m}@\,\widehat{\omega}}\,\}\ \mathsf{exps\text{-}wf}} \qquad \frac{\mathsf{nooverlap}(\mathit{espcs}) \qquad \forall \mathit{espc} \in \mathit{espcs} :\ \hat{\Phi} \Vdash \mathit{espc}\ \mathsf{wf}}{\widehat{\Phi} \Vdash \langle\ \overline{\mathit{ds\hat{p}cs}}\ ;\ \mathit{espcs}\ ;\ \overline{\nu}\ \rangle\ \mathsf{exps\text{-}wf}}\ (\textsc{WfShExports})$$

Imports well-formed in contexts

$\boxed{\Phi \vdash \Phi\ \mathsf{imps\text{-}wf}}$ $\boxed{\Phi \vdash \nu{:}\tau^{m}@\,\omega\ \mathsf{imps\text{-}wf}}$

$$\frac{\overline{\Phi \vdash \nu{:}\tau^{m}@\,\omega\ \mathsf{imps\text{-}wf}}}{\Phi \vdash \{\!|\,\overline{\nu{:}\tau^{m}@\,\omega}\,|\!\}\ \mathsf{imps\text{-}wf}} \qquad \frac{\mathsf{imps}(\tau) = \cdot}{\Phi \vdash \nu{:}\tau^{-}@\,\omega\ \mathsf{imps\text{-}wf}}\ (\textsc{WfSigImports})$$

$$\frac{\mathsf{imps}(\tau) \subseteq \mathsf{dom}(\Phi) \qquad \mathsf{provs}(\tau) \cup \mathsf{idents}(\omega) \subseteq \{\nu\} \cup \mathsf{depends}^{+}_{\Phi}(\mathsf{imps}(\tau))}{\Phi \vdash \nu{:}\tau^{+}@\,\omega\ \mathsf{imps\text{-}wf}}\ (\textsc{WfModImports})$$

Worlds well-formed in contexts

$\boxed{\Phi \vdash \Phi\ \mathsf{wlds\text{-}wf}}$ $\boxed{\widehat{\Phi} \Vdash \widehat{\Phi}\ \mathsf{wlds\text{-}wf}}$

$\boxed{\Phi \vdash \tau^{m}@\,\omega\ \mathsf{wlds\text{-}wf}}$ $\boxed{\widehat{\Phi} \Vdash \widehat{\tau}^{m}@\,\widehat{\omega}\ \mathsf{wlds\text{-}wf}}$

$$\frac{\overline{\Phi \vdash \tau^{m}@\,\omega\ \mathsf{wlds\text{-}wf}}}{\Phi \vdash \{\!|\,\overline{\nu{:}\tau^{m}@\,\omega}\,|\!\}\ \mathsf{wlds\text{-}wf}} \qquad \frac{\overline{\widehat{\Phi} \Vdash \widehat{\tau}^{m}@\,\widehat{\omega}\ \mathsf{wlds\text{-}wf}}}{\widehat{\Phi} \Vdash \{\,\overline{\nu{:}\widehat{\tau}^{m}@\,\widehat{\omega}}\,\}\ \mathsf{wlds\text{-}wf}}$$

$$\frac{\Phi \vdash \omega\ \mathsf{wf} \qquad \omega \sqsupseteq \mathsf{extworld}(\nu;\tau)}{\Phi \vdash \tau^{m}@\,\omega\ \mathsf{wlds\text{-}wf}}\ (\textsc{WfWorlds}) \qquad \frac{\widehat{\Phi} \Vdash \widehat{\omega}\ \mathsf{wf}}{\widehat{\Phi} \Vdash \widehat{\tau}^{m}@\,\widehat{\omega}\ \mathsf{wlds\text{-}wf}}\ (\textsc{WfShWorlds})$$

Figure 7.13: Definition of well-formedness of module contexts and associated objects.

8 PACKAGE LEVEL

Finally, the package level of Backpack is the outermost one. This is an entirely new contribution to Haskell, rather than a formalization of existing Haskell entities or semantics, like the core level and much of the module level are. The novelty of this level happily leads to a formalization that is cleaner and more free of all the technical specifics that pervade the module level. This isn't surprising, however, since this level was entirely designed in tandem with its formalization.

The main expression at the package level is, obviously, the *package*. Packages are declared in an ambient package context, instantiated, and linked together. They're treated, on one hand, like mixin modules built out of logically named inner components, and on the other hand, like dependency graphs of their constituent modules. Mixin linking of the logical module names appeals to *unification* of semantic objects in order to determine the physical structure of packages.

Much of the package level for Backpack was already introduced with motivating examples in Chapter 2. This chapter will therefore focus on a more detailed discussion of the technical formalization of packages in Backpack.

INTERACTION WITH MODULE LEVEL As the package level is the outermost level, there are no Backpack objects "passed into" the package level from other levels. Instead, the package level is concerned with top-level package declarations (D) and contexts thereof (Δ). It does, however, appeal to the module level for its own shaping and typing passes (as the module level appeals to the core level).

The package level synthesizes a *package shape* ($\hat{\Xi}_{\mathsf{pkg}}$) that determines the physical structure of the package. From that, for each module ν in the package, it extracts the module-level information about the physical names (*i.e.*, module identities) bound in the module expression. For modules, that information is just the module identity to designate "this" module, ν_0, and for signatures, that information is a *realizer* (ρ, §7.3) that designates the provenances of core declarations, whether they're named core declarations or `instance` declarations.

With that structural information from the package shape, the module level performs the typing of the module expression, yielding a module type (τ), a world (ω), and, for signatures, a signature declaration context (Φ_{sig}). All of that type information gets assembled into a *package type* (Ξ) describing both the logical and physical aspects of the package.

For shaping, the interaction with the module level is similar, with shapey variants of all the objects from above. The passing of structural information from a package shape ($\widehat{\Xi}$) to module level though is simpler during package shaping: it's simply some fresh module identities that are generated.

INTERNALS Unlike the module level, the package level has additional expression forms whose typing does not appeal to lower levels of the formalization: a top-level package declaration D and a package **include** binding. Both these expression forms involve *thinning* and *renaming*, two key concepts in the package level without concern to the module level.

8.1 SYNTAX OF THE PACKAGE LEVEL

The syntax of Backpack's module level is presented in Figure 8.1. In a couple key ways, this syntax differs from that presented in the earlier Backpack work.[1] First, there's no "expression"

1 Kilpatrick *et al.* (2014), "Backpack: Retrofitting Haskell with Interfaces".

Backpack Package Syntax

Package Names	P	$\in$	*PkgNames*	
Logical Module Names	ℓ	$\in$	*ModNames*	
Package Bindings	B	$::=$	$\ell = M$	Module Binding
		$\mid$	$\ell :: S$	Signature Binding
		$\mid$	$\ell = \ell$	Alias Binding
		$\mid$	**include** $P\ t\ r$	Package Inclusion
Thinning Specs	t	$::=$	$(\overline{\ell})$	
Renaming Specs	r	$::=$	$\langle\overline{\ell \mapsto \ell}\rangle$	
Package Definitions	D	$::=$	**package** $P\ t$ **where** $\overline{B}$	
Package Repositories	R	$::=$	$\overline{D}$	

Figure 8.1: Syntax of Backpack's package level.

form—previously denoted with metavariable E—for modules, signatures, or module name references, and with a single binding form $\ell = E$. Instead, this formalization distributes the three expression forms into three distinct binding forms. Second, module names no longer have any hierarchical, "dotted path" structure, a restriction that eliminates some complications around *thinning*.

Package names (P) model the entire package name. Package names are drawn from a sort *PkgNames* and therefore have no structure to them. In particular, two different "versions" of a package would have names containers-1.0 and containers-1.5, names that have no semantic connection to each other. (Recall that versioning is not part of Backpack.)

Logical module names (ℓ)[2] are similarly drawn from a sort (*ModNames*) rather than having any structure. As mentioned above, this is a departure from earlier presentations of Backpack.

Package bindings (B) are the key expression forms at the package level. The first form, module bindings, assigns a module name ℓ to a module expression M. As already discussed in the module level, this is how module names are bound to (nameless) module expressions to be subsequently imported, as opposed to the self-named module expressions in Haskell. Signature bindings are similar.

Package inclusion brings the bindings from the designated package P into the current namespace, after applying the given thinning and renaming specs, t and r respectively. A thinning spec designates some subset of module names from a package, and a renaming spec designates a renaming of module names when including the target bindings into the namespace; the l.h.s. of the arrow is renamed to the r.h.s.

Package declarations (D) introduce new packages as defined by a sequence of bindings ($\overline{B}$) followed by a thinning spec (t). A couple deviations from existing Haskell are worth noting.

- In Haskell today—or rather, in the Cabal specification language—a package is defined as an unordered collection of modules. A build tool deterministically derives the dependency order of the collection so as to process the modules in order. (The same is true of the internal language, or IL, introduced in the next chapter.) In Backpack, however, a package is defined with a *sequence* of module bindings that must be processed in order; this restriction regularized the definition of shaping and typing of bindings.

- Moreover, in a Cabal package specification, the constituent modules that comprise a package's definition are partitioned, by the package author, into *exposed* modules vs. *hidden* modules. The former modules are provided to client packages and made available for importing by those clients' modules, while the latter modules are internal implementation details that are *not* available for import by clients. This partition acts as

2 The metavariable ℓ comes from that used in MixML (Dreyer and Rossberg, 2008). Though that system imposes a hierarchical structure to these "labels", as "label paths" (ℓs).

a form of *implementor-side abstraction* (§1.1.1): the implementor, *i.e.*, the package author, decides which aspects of the package to reveal to clients. In Backpack, this same form of abstraction is expressed as a thinning specification on a package declaration. That thinning spec designates the subset of module names that are bound within the package that should be made available to clients.

Finally, a package repository is simply a sequence of package declarations. Every package declaration is considered within some repository. The bindings that comprise a package declaration can only **include** other packages which have been declared earlier in the sequence. The vast complexity of package management systems is thereby swept under this austere rug.

8.1.1 *Inferring thinning and renaming specs*

The careful reader might have noticed that the motivating examples from Chapter 2 omitted thinning and renaming specs on **include** bindings and thinning specs on **package** declarations, in contrast to the defined syntax which requires these specs on those forms. Though they've been omitted from the examples for the sake of presentation, both thinning and renaming can be implicitly derived in a straightforward manner as follows:

- The implicit thinning spec on an **include** P binding is the full list of logical names bound in P. For example, within the structures package from Figure 2.2 (p. 38), the **include** arrays-sig has an implicit thinning spec $(\mathsf{Prelude}, \mathsf{Array})$ since those are the two logical module names bound in arrays-sig. This can be derived from the package type of P (or even its shape).
- The implicit renaming spec on an **include** P binding is simply the empty renaming $\langle\cdot\rangle$.
- The implicit thinning spec on a **package** P **where** $\overline{B}$ declaration is, again, the full list of logical module names bound in the $\overline{B}$. Returning to the structures package, its own declaration's implicit thinning spec is $(\mathsf{Prelude}, \mathsf{Array}, \mathsf{Set}, \mathsf{Graph}, \mathsf{Tree})$. This can be derived from the package type (or shape) of the bindings.

8.2 SEMANTICS OF THE PACKAGE LEVEL

As with the module level, typing in the package level comprises a *shaping* pass and a *typing* pass. The former concerns itself with all the mixin linking, unification, and dependency graph structure, while the latter uses the result of that in order to perform typechecking all the way down to the core level.

The package declaration form (D) is distinguished in Backpack in that it's the origin of the shaping-vs.-typing split. In the package type system, the typing of a package declaration involves first synthesizing a package shape for the package's bindings, and then using that shape to synthesize a package type. The package shape is the origin of all "shapiness" across the levels of Backpack.

The package level is given meaning via a typing judgment that classifies well-formed package bindings and package declarations with semantic objects. Those semantic objects are package types (Ξ) and parameterized package types ($\forall\overline{\alpha}.\Xi$) respectively. The typing judgment directly yields the deterministic process we call *the typing pass*, and similarly a shaping judgment yields *the shaping pass*. Those judgments are presented in §8.6

Also novel to the semantics of the package level are the concepts of *thinning* and *renaming*. The former involves the traversal and analysis of dependency graphs formed by the module identities occurring in package types, while the latter involves a straightforward transformation of the logical module names contained within package types (cf. the depends auxiliary function).

Backpack Module Semantic Objects

Signature Realizers	ρ	::=	$\widehat{\tau} @ \widehat{\omega}$
Physical Module Contexts	Φ	::=	$\{\!\mid \overline{\nu{:}\tau^{m} @ \omega} \mid\!\}$
Physical Module Shape Contexts	$\widehat{\Phi}$	::=	$\{ \overline{\nu{:}\widehat{\tau}^{m} @ \widehat{\omega}} \}$
Logical Module Contexts	$\mathcal{L}$	::=	$\overline{\ell \mapsto \nu}$
Module Contexts & Package Types	Γ, Ξ	::=	$(\!\mid \Phi \,;\, \mathcal{L} \mid\!)$
Module Shape Contexts & Package Shapes	$\widehat{\Gamma}, \widehat{\Xi}$	::=	$(\widehat{\Phi} \,;\, \mathcal{L})$
Module Identity Substitutions	θ	::=	$\{\overline{\alpha := \nu}\}$
Package Environments	Δ	::=	$\cdot \mid \Delta, P : \forall\overline{\alpha}.\Xi$

Figure 8.2: Semantic objects relevant to Backpack's package level, with key objects in bold.

8.3 SEMANTIC OBJECTS

Figure 8.2 presents the semantic objects particularly relevant to the package level. Almost all of these objects were introduced in the module level. The only object that is entirely new at this level is the package environment (Δ). Additionally, module identity substitutions (θ) aren't really new to this level, but they haven't served as an explicit part of the semantics of shaping and typing in the lower levels like they do in the package level. And finally, package types (Ξ) are the same object as module contexts (Γ) but deserve additional consideration now.

PACKAGE TYPES AND PARAMETERIZED PACKAGE TYPES A *package type* (Ξ) is the exact same object as the module contexts (Γ) from the module level: it comprises a physical module context (Φ) that contain typings for module identities and a logical module context mapping *logical* module names to those *physical* module identities. Here at the package level, the same object serves as the type of *package bindings*.

A *parameterized package type* ($\forall\overline{\alpha}.\Xi$) serves as the type of whole packages. The $\forall$-quantifier binds some module identity variables ($\overline{\alpha}$) over the constituent package type (Ξ). Those variables represent precisely those identities representing module holes (the "αs") among the package bindings designated by (Ξ), along with the provenances of those holes' entities (the "βs"). That the variables are *bound* simply denotes the type of a *package*, *i.e.*, a collection of module bindings that will not be extended or merged or unified (but, naturally, will be **include**d into other packages).

Parameterized package types admit α-conversion and -equivalence. Unlike conventional $\forall$ binders, however, parameterized package types are eliminated into package types not via *application* of module identities but via *substitution* of those identities. We'll see more on this in §8.5.

PACKAGE ENVIRONMENTS A *package environment* (Δ) is a context that maps package names (P) to parameterized package types ($\forall\overline{\alpha}.\Xi$). References to other packages by their names, via the **include** binding, are statically processed by looking up their types in this environment. The environment is initialized to be empty and is extended during the processing of successive package declarations (D) within a package repository (R).

8.4 THINNING JUDGMENT

Figure 8.3 presents the definition of the thinning judgment on package types. At the core is a straightforward idea: gather up all the module identities which the chosen logical module names (t) denote or on which they transitively depend, thereby accumulating a subset of module identities N, and restrict the package type Ξ to only those physical modules and only the chosen logical ones.

The second premise appeals to the $\mathsf{depends}^{+}_{\Phi}(\nu)$ definition for the set of all module identities that ν transitively depends on within Φ, including itself, resulting in N. The third premise restricts both physical and logical components of the source package type Ξ: the former is restricted to the set N, while the latter is restricted to only those logical names designated in the thinning spec t. And finally the fourth premise asserts that any remaining holes must be fillable, via the $\mathsf{located}_{\Xi}(\nu)$ definition. The second and fourth premises deserve additional consideration.

dependency closure The $\mathsf{depends}^{+}_{\Phi}(\nu)$ function is just the addition of ν to $\mathsf{depends}_{\Phi}(\nu)$. The latter function in turn appeals to the inner function $\mathsf{depends}_{\Phi;N}(\nu)$ which carries out the meat of the transitive closure. Its definition is perhaps most easily understood as the reachability set within the dependency graph induced by the context Φ, while carrying the set N of visited nodes. Each "step" of dependency is defined, via the set N', as all the modules occurring within the module type (τ) and world (ω) of the source module ν's typing in Φ.

abstract identities located in package Every abstract module in the resulting package type must either be a module hole itself, with some logical name ℓ used to fill it, or the identity of some declared entity in a signature that itself can be filled. Without this premise, we might get a thinned package that has holes that have no ℓ by which to implement them!

One might wonder why this fact must be explicitly ensured via the $\mathsf{located}_{\Xi}(\nu)$ premise of the thinning judgment; why does it not merely fall out of the Backpack type system? The source of potential trouble lies in the thinning spec t. Consider the structures package from Figure 2.2 (p. 38), which has two holes, Prelude and Array. Now consider the thinning spec $t = (\mathsf{Array}, \mathsf{Graph})$ applied to the type of this package. If we apply the thinning judgment to this type with this thinning spec, our set N contains the module identity variable denoting the Prelude module—call it α_P—but there's no logical name ℓ pointing to this hole in the resulting package type. The variable α_P is therefore not "located" in the resulting package type. Because such a variable can only be unified by linking in a concrete module binding with the same logical name, that variable would be impossible to unify. The final premise of the thinning judgment rules out such an ill-formed thinning.

8.5 unification

Unification in Backpack is the technical manifestation of linking: module identities of holes are unified with the module identities of implementations. Figure 8.4 provides all the relevant definitions for unification of semantic objects in Backpack. The various unification judgments are explained in order:

At the outermost level is the *unifying merge* judgment on package shapes, $\Vdash \widehat{\Xi} + \widehat{\Xi} \Rightarrow \widehat{\Xi}$ which we'll see is invoked in the shaping rule for sequencing package bindings. Rule (UnifMerge) describes analyzing the two package shapes to determine a unifying substitution θ, which is then applied to each side and merged together to form the result.

The third premise of (UnifMerge) gathers up all the pairs of identities for logical module names in both sides of the merge. Each such (ν_1, ν_2) describes a pair that should be linked together, *i.e.*, unified. The last premise invokes the unification of these pairs within their respective physical module shape contexts, $\widehat{\Phi}_1$ and $\widehat{\Phi}_2$, each restricted to only the chosen identities.

For example, if $\widehat{\Xi}_1$ contains the implementation of a Data.Set module, ν_S, and $\widehat{\Xi}_1$ contains a signature for Data.Set, α_S, then the common logical name will yield a pair to unify, (ν_S, α_S). As a final result, not only will $\alpha_S := \nu_S$ be part of the resulting subsitution, but so will all the $\beta_i := \nu_i$ determined from unifying the provenances of entities declared in the Data.Set signature and implemented in the Data.Set module (*espcs*), as well as further such βs from unifying the identities of their respective world shapes ($\widehat{\omega}$).

The next unification judgment, $\Vdash \left(\overline{\nu};\widehat{\Phi}\right) \leftrightarrows \left(\overline{\nu};\widehat{\Phi}\right) \rightsquigarrow \theta$, defines the main process of unification in Backpack: unify, in order, (1) all pairs of module identities, (2) all pairs of (export specs of) module shapes, and (3) all pairs of world shapes. Each unification produces a substitution which is then applied to the subsequent objects to unify. In the end, the resulting substitution is the (function) composition of the substitutions in order.

This unification judgment intentionally does not proceed "module-wise" through the pairs of modules. Rather, it's important that this last step (3) occurs *after* the second step (2) since the former relies on all the class constraints in worlds having been already unified and substituted. Specifically, rule (UnifWorldShp) checks that two world facts to unify have *syntactically identical* instance heads; since those heads can mention classes declared in signatures being presently unified with implementations, we'd like that substitution to be applied to their respective world (shapes) before trying to unify those.

Consider this staging in the Data.Set example. The module and signature would have the following export specs and world (shape) facts:

$$\begin{array}{ll}
\text{Data.Set module:} & \begin{array}{l} \mathit{espc}_1 = [\nu_S]\mathtt{Set}(\ldots) \\ \mathit{fact}_1 = (a :: *).[\nu_P]\mathtt{Eq}\ ([\nu_S]\mathtt{Set}\ a) \ \mapsto\ \nu_S \end{array} \\
\text{Data.Set signature:} & \begin{array}{l} \mathit{espc}_2 = [\beta_S]\mathtt{Set}() \\ \mathit{fact}_2 = (a :: *).[\nu_P]\mathtt{Eq}\ ([\beta_S]\mathtt{Set}\ a) \ \mapsto\ \beta'_S \end{array}
\end{array}$$

In order to unify fact_1 and fact_2, we must first apply the substitution $\beta_S := \nu_S$ to each side, resulting in two instance heads that are syntactically the same. Only at that point can the two facts be unified, yielding $\beta'_S := \nu_S$, which allows the two modules' world shapes to be merged in (UnifMerge).

Unification on module identities appeals to an underlying unification of first-order recursive terms, denoted $\mathsf{unify}(-)$. This definition is not defined here in this formalization but it's standard folklore—from (Huet, 1976) to (Gauthier and Pottier, 2004), for example—and can be performed efficiently. See A.1.1 for more details about unification on module identities.

Unification on sets of entity name specs, *i.e.*, export specs, proceeds by matching syntactic entity names. In the earlier example, the entity name `Set` appearing in both sides' export specs led to the unification of their respective provenances, ν_S and β_S. Rule (UnifEspcs) applies when two such specs have matching names, while rule (UnifEspcsTriv) applies when none of them match.

Unification on world shapes proceeds by matching syntactic instance heads. As already discussed above, in order to correctly unify world shapes in the presence of locally declared classes and types—which may appear in the instance heads of those world shapes—those classes and types should have already been unified in (UnifPhysCtx). The structure of these rules follows that of the previous rules.

Finally, the judgments for sequentially unifying vectors of objects are entirely straightforward.

8.6 SHAPING AND TYPING JUDGMENTS

The shaping judgments for the package level are defined in Figure 8.5. These judgments are straightforward and closely match the abridged versions from Chapter 3. A few observations about the rules defining the judgments are worth noting:

- Compared to the earlier presentation, the module (ShPkgMod) and signature (ShPkgSig) rules feature world shapes and the **include** rule (ShPkgInc) features thinning.
- The thinning rule (ShPkgInc) operates on *types*, appealing to the *thinning judgment* of §8.4, before casting down the resulting package type to a shape.

- The notation $\overline{\alpha}$ fresh is a conventional way to denote α-converted variables such that the $\overline{\alpha}$ do not appear in the contexts of the rule conclusion.
- Rule (ShPkgSeq) invokes the *unifying merge* judgment of §8.5. Here is where the Backpack type system determines linking.

The typing judgments for the package level are likewise defined in Figure 8.6. Again, these judgments are straightforward extensions of the earlier abridged versions, with the following observations:

- Compared to the earlier presentation, the module (TyPkgMod) and signature (TyPkgSig) rules feature worlds and the **include** (TyPkgInc) and package definition (TyPkg) rules feature thinning.
- The presence of worlds in (TyPkgSig) is worth noting in particular. The contextual package shape, $\hat{\Xi}_{\mathsf{pkg}}$, designates not just the module shape $\hat{\tau}_0$, which determines the provenances of the named entities declared in the signature, but now also the world shape $\hat{\omega}_0$, which determines the provenances of the type class instances declared in the signature. Together, these two objects constitute the *realizer*, $\hat{\tau}_0 \,@\, \hat{\omega}_0$, that's passed to the module-level judgment; recall §7.3.
- The typing of package repositories (R) has been defined. There's nothing technically interesting about it, although we'll soon see an interesting modification of it for the *elaboration semantics*.

8.7 METATHEORY OF THE PACKAGE LEVEL

The package level introduces one new semantic object that comes equipped with a well-formedness judgment: package environments (Δ). This judgment is straightforwardly defined as the well-formedness of every package type bound in the environment:

$$\vdash \Delta \ \mathsf{wf} \quad \overset{\mathsf{def}}{\Leftrightarrow} \quad \forall (P : \forall \overline{\alpha}.\Xi) \in \Delta : \ \cdot \ \vdash \ \Xi \ \mathsf{wf}$$

8.7.1 *Structural metatheory about well-formedness*

The well-formedness judgments on semantic objects across all of Backpack satisfy various *structural* properties. Here I use the term *structural* to indicate both their structural similarity across all such judgments and their utility to the rest of the metatheory. I use conventional names for some of these properties as found elsewhere in the type theory literature, *e.g.*, *Weakening* and *Cut*.

The structural metatheoretic properties about the well-formedness of a semantic object A are the following:

- **Weakening** (Lemma A.4): If A is well-formed in some physical module context Φ and $\Phi \oplus_? \Phi_W$, then A is also well-formed in the *weakened*, sub-context[3] $\Phi \oplus \Phi_W$. We can keep adding more (compatible) module information to the context and preserve the well-formedness of A.
- **Merging** (Lemma A.5): If A_1 is well-formed in Φ_1 and A_2 is well-formed in Φ_2, and if $A_1 \oplus_? A_2$ and $\Phi_1 \oplus_? \Phi_2$, then the merged object $A_1 \oplus A_2$ is well-formed in the merged context $\Phi_1 \oplus \Phi_2$. The individual well-formedness of two mergeable objects combines into their merged well-formedness.
- **Invariance under substitution**[4] (Lemma A.6): If Φ is well-formed and if module identity substitution θ is applicable to Φ (*i.e.*, $\mathsf{apply}(\theta; \Phi)$ is defined), and if A is well-formed

3 According to the $\leqslant$ order on Φ.

4 This is not called "substitutability" since that generally refers to the preservation of expression typing under substituting a free variable with an expression of the same type. In Backpack, substitution is not on *expressions* but on *types*, in a sense. That fits in with the general theme of Backpack being about *linking* rather than *substitution*.

in Φ and θ is applicable to A, then $\mathsf{apply}(\theta; A)$ is well-formed in $\mathsf{apply}(\theta; \Phi)$. We can link implementations for holes and still preserve the well-formedness of A, so long as the linking substitution was compatible with the context (which is a big "if").

- **Strengthening** (Lemma A.7): If A is well-formed in Φ and N is a subset of the module identities contained in Φ, and if A in some sense depends only on the identities in N, then A is also well-formed in the *strengthened*, super-context $\Phi|_N$. We can drop some module information from the context, so long as A doesn't rely on any of it,[5] and preserve the well-formedness of A.
- **Cut** (Lemma A.8): This property is primarily defined for physical module contexts. If Φ_1 is well-formed in Φ and Φ_2 is well-formed in the merged context $\Phi \oplus \Phi_1$, and if $\Phi_1 \oplus_? \Phi_2$, then $\Phi_1 \oplus \Phi_2$ is well-formed in the original context Φ. Chaining together the well-formedness of Φ_1 with the contingent well-formedness of Φ_2 assuming Φ_1 means we can prove the well-formedness of their merged result as if they were originally considered together.

Many instances of these properties are defined quite literally by substituting various semantic objects for A. Other instances are "thematically equivalent" to that substitution but not literally so. In any case, the full definitions are given in the appendix (B.2).

8.7.2 *Regularity of typing judgments*

The typing judgments of the package level satisfy the same kind of *regularity* property as those of the module level. As we'll see in the next chapter, the more interesting property of these judgments is *elaboration soundness*, but for now the regularity property suffices—and it's used in the proofs of elaboration soundness.

- **Regularity of typing of a package binding:** A well-typed package binding, in a well-formed package environment and a well-formed context, has a package type that's well-formed if that type is mergeable within the module context.
- **Regularity of typing of a sequence of package bindings:** A well-typed sequence of package bindings, in a well-formed package environment and a well-formed context, has a package type that's well-formed.
- **Regularity of typing of a package definition:** A well-typed package definition, in a well-formed package environment, has a parameterized package type that's well-formed, *i.e.*, the constituent package type is well-formed including the free variables.

The mergeability hypothesis in the regularity for package bindings, $\Gamma \oplus_? \Xi$ (which is notation for $\Gamma.\Phi \oplus_? \Xi.\Phi$), might seem surprising. The justification for this hypothesis is located in the recursive appeal to regularity of *module* typing, which is required in the proof of this property (for the case that the B is a $\ell = [M]$), and which has its own such mergeability requirement, that $\Gamma.\Phi \oplus_? \nu_0{:}\tau^+ @ \omega$. That requirement stems from the fact that, as discussed in §7.7.1, well-formedness of a physical module context must "assume itself" by merging itself into its own context, thus $\Gamma.\Phi \oplus \nu_0{:}\tau^+ @ \omega \vdash \ldots$. To summarize, we have that mergeability is a requirement of regularity of module typing which is a requirement of regularity of package binding typing.

Another point to note is that the latter two regularity properties don't concern a module context (Γ). That's because both judgments are not contextualized by a module context; instead they're both contextualized only by a *package* context.

The proofs of these regularity statements rely on two key ingredients: first, the structural metatheory about the various *well-formedness* judgments on semantic objects, across all layers, and second, metatheory about various auxiliary definitions that are part of the type system.

5 Backpack's syntax of "types," *i.e.*, its semantic objects, is so rich that a type expresses all the dependency that actually exists at the "term level," *i.e.*, its module expressions. That richness of semantic objects is why the Strengthening property holds.

Theorem (Regularity of typing of a package binding)**.**

If $\vdash\ \Delta$ *wf and* $\cdot\ \vdash\ \Gamma$ *wf and* $\Delta;\Gamma;\hat{\Xi}_{pkg}\ \vdash\ B\ :\ \Xi$ *and* $\Gamma\ \oplus_?\ \Xi$*, then* $\Gamma.\Phi\ \vdash\ \Xi$ *wf.*

Proof. By case analysis on the rule of the judgment derivation. Let $\Gamma = (\!|\ \Phi\ ;\ \mathcal{L}\ |\!)$.

- Case (TyPkgAlias): immediate.
- Case (TyPkgMod): As already discussed above, the mergeability premise yields mergeability of $\Phi\ \oplus_?\ \nu_0{:}\tau^{+}@\,\omega$. Then by regularity of module typing we have $\Phi\ \vdash\ \Xi$ wf.
- Case (TyPkgSig): The mergeability premise gives $\Phi\ \oplus_?\ (\nu_0{:}\sigma^{-}@\,\omega\ \oplus\ \Phi_{\mathsf{sig}})$, which entails that $\Phi\ \oplus_?\ \nu_0{:}\sigma^{-}@\,\omega$ and $\Phi\ \oplus_?\ \Phi_{\mathsf{sig}}$ individually. Then by regularity of signature typing we have $\Phi\ \vdash\ \Phi_{\mathsf{sig}}$ wf and $\Phi\ \oplus\ \Phi_{\mathsf{sig}}\ \vdash\ \nu_0{:}\sigma^{-}@\,\omega$ wf. Then by *Cut* we have the desired result.
- Case (TyPkgInc): We need to show that $\mathsf{apply}(\theta;\Xi'')$ is well-formed.
  - Wf-ness of Δ implies wf-ness of Ξ, the type of P, by definition.
  - Wf-ness of Ξ implies wf-ness of Ξ' due to the preservation of wf-ness by thinning (Lemma A.9).
  - Wf-ness of Ξ' implies wf-ness of Ξ'' due to the preservation of wf-ness by logical module renaming (Lemma A.12).
  - The existence of this derivation means that the resulting package type, $\mathsf{apply}(\theta;\Xi'')$ we know that θ is applicable to Ξ''. Then by *Invariance under Substitution*, wf-ness of Ξ'' implies wf-ness of $\mathsf{apply}(\theta;\Xi'')$.

□

Theorem (Regularity of typing of a sequence of package bindings)**.**

If $\vdash\ \Delta$ *wf and* $\Delta;\hat{\Xi}_{pkg}\ \vdash\ \overline{B}\ :\ \Xi$*, then* $\Phi\ \vdash\ \Xi$ *wf.*

Proof. By induction on the judgment derivation.

- Case (TyPkgNil): immediate.
- Case (TyPkgSeq):
  - By inductive hypothesis we have wf-ness of Ξ_1.
  - By inversion of the rule we have that $\Xi = \Xi_1\ \oplus\ \Xi_2$, then $\Xi_1.\Phi\ \oplus_?\ \Xi_2.\Phi$.
  - By regularity on single package binding we have $\Xi_1.\Phi\ \vdash\ \Xi_2$ wf.
  - By *Cut* (2) we have wf-ness of Ξ.

□

Theorem (Regularity of typing of a package definition)**.**

If $\vdash\ \Delta$ *wf and* $\Delta\ \vdash\ D\ :\ \forall\overline{\alpha}.\Xi$*, then* $\Phi\ \vdash\ \Xi$ *wf.*

Proof. By inversion of the (TyPkg) rule. We want to show $\cdot\ \vdash\ \Xi'$ wf.

- By regularity on sequences of package bindings we have wf-ness of Ξ.
- By preservation of wf-ness by thinning (Lemma A.9) we have wf-ness of Ξ'.

□

Thinning of Package Types

Thinning

$$\boxed{\vdash \Xi \xrightarrow{t} \Xi}$$

$$\frac{\begin{array}{cc} \Xi = (\!|\, \Phi \,;\, \mathcal{L} \,|\!) & N = \bigcup_{\ell \in \overline{\ell}} \mathsf{depends}^{+}_{\Phi}(\mathcal{L}(\ell)) \\ \Xi' = (\!|\, \Phi|_{N} \,;\, \mathcal{L}|_{\overline{\ell}} \,|\!) & \forall \nu{:}\tau^{-}@\,\omega \in \Xi'.\Phi : \ \mathsf{located}_{\Xi'}(\nu) \end{array}}{\vdash \Xi \xrightarrow{(\overline{\ell})} \Xi'}$$

$$\begin{array}{lcl} \Phi|_{N} & \overset{\mathsf{def}}{=} & \{\!|\, \nu{:}\tau^{m}@\,\omega \mid \nu{:}\tau^{m}@\,\omega \in \Phi \ \wedge \ \nu \in N \,|\!\} \\ \mathcal{L}|_{\overline{\ell}} & \overset{\mathsf{def}}{=} & (\ell \mapsto \nu \mid \ell \mapsto \nu \in \mathcal{L} \ \wedge \ \ell \in \overline{\ell}) \end{array}$$

Module identity dependency

$$\boxed{\mathsf{depends}_{\Phi}(\nu) \ :_{\mathsf{par}} \ N} \quad \boxed{\mathsf{depends}_{\Phi;N}(\nu) \ :_{\mathsf{par}} \ N} \quad \boxed{\mathsf{depends}^{+}_{\Phi}(\nu) \ :_{\mathsf{par}} \ N}$$

$$\mathsf{depends}_{\Phi}(\nu) \ \overset{\mathsf{def}}{=} \ \mathsf{depends}_{\Phi;\cdot}(\nu) \qquad \text{where } \nu \in \mathsf{dom}(\Phi)$$

$$\begin{array}{lcl} \mathsf{depends}_{\Phi;N}(\nu) & \overset{\mathsf{def}}{=} & \emptyset \qquad \text{where } \nu \in N, \ \nu \in \mathsf{dom}(\Phi) \\ \mathsf{depends}_{\Phi;N}(\nu) & \overset{\mathsf{def}}{=} & N' \cup \left(\bigcup_{\nu' \in N'} \mathsf{depends}_{\Phi;(N,\nu)}(\nu')\right) \\ & & \text{where } \begin{cases} \nu \notin N \\ N' = \mathsf{provs}(\tau) \cup \mathsf{provs}(\omega) \quad \text{if } \nu{:}\tau^{m}@\,\omega \in \Phi \end{cases} \end{array}$$

$$\mathsf{depends}^{+}_{\Phi}(\nu) \ \overset{\mathsf{def}}{=} \ \begin{cases} \{\nu\} & \nu \notin \mathsf{dom}(\Phi) \\ \{\nu\} \cup \mathsf{depends}_{\Phi}(\nu) & \nu \in \mathsf{dom}(\Phi) \end{cases}$$

Provenances & imports

$$\boxed{\mathsf{imps}(\tau) \ : \ N} \quad \boxed{\mathsf{provs}(\{\tau, \mathit{dspc}, \mathit{typ}, \mathit{cls}, \omega, \mathit{fact}\}) \ : \ N}$$

$$\begin{array}{lcl} \mathsf{imps}(\langle\!|\, \overline{\mathit{dspc}} \,;\, \overline{\mathit{espc}} \,;\, \overline{\nu} \,|\!\rangle) & \overset{\mathsf{def}}{=} & \overline{\nu} \\ \mathsf{provs}(\langle\!|\, \overline{\mathit{dspc}} \,;\, \overline{\mathit{espc}} \,;\, \overline{\nu} \,|\!\rangle) & \overset{\mathsf{def}}{=} & \left(\bigcup_{\mathit{dspc} \in \overline{\mathit{dspc}}} \mathsf{provs}(\mathit{dspc})\right) \cup \left\{\overline{\mathsf{ident}(\mathit{espc})}\right\} \cup \{\overline{\nu}\} \end{array}$$

$$\begin{array}{lcl} \mathsf{provs}(\mathtt{data}\ T\ \mathit{kenv}) & \overset{\mathsf{def}}{=} & \emptyset \\ \mathsf{provs}(\mathtt{data}\ T\ \mathit{kenv} = \overline{K\ \overline{\mathit{typ}}}) & \overset{\mathsf{def}}{=} & \bigcup_{\mathit{typ} \in \overline{\overline{\mathit{typ}}}} \mathsf{provs}(\mathit{typ}) \\ \mathsf{provs}(x\ \mathtt{::}\ \mathit{typ}) & \overset{\mathsf{def}}{=} & \mathsf{provs}(\mathit{typ}) \\ \mathsf{provs}(\mathtt{class}\ C\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \overline{x\ \mathtt{::}\ \mathit{typ}}) & \overset{\mathsf{def}}{=} & \left(\bigcup_{\mathit{cls} \in \overline{\mathit{cls}}} \mathsf{provs}(\mathit{cls})\right) \cup \left(\bigcup_{\mathit{typ} \in \overline{\mathit{typ}}} \mathsf{provs}(\mathit{typ})\right) \\ \mathsf{provs}(\mathtt{instance}\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \mathit{cls}) & \overset{\mathsf{def}}{=} & \left(\bigcup_{\mathit{cls} \in \overline{\mathit{cls}}} \mathsf{provs}(\mathit{cls})\right) \cup \mathsf{provs}(\mathit{cls}) \end{array}$$

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{provs}(\alpha) & \overset{\mathsf{def}}{=} & \emptyset & \mathsf{provs}([\nu]C\ \overline{\mathit{typ}}) & \overset{\mathsf{def}}{=} & \{\nu\} \cup \bigcup_{\mathit{typ} \in \overline{\mathit{typ}}} \mathsf{provs}(\mathit{typ}) \\ \mathsf{provs}(\mathtt{forall}\ \mathit{kenv}\mathtt{.}\ \mathit{typ}) & \overset{\mathsf{def}}{=} & \mathsf{provs}(\mathit{typ}) & \mathsf{provs}(\omega) & \overset{\mathsf{def}}{=} & \bigcup_{\mathit{fact} \in \omega} \mathsf{provs}(\mathit{fact}) \\ \mathsf{provs}([\nu]T\ \overline{\mathit{typ}}) & \overset{\mathsf{def}}{=} & \{\nu\} \cup \bigcup_{\mathit{typ} \in \overline{\mathit{typ}}} \mathsf{provs}(\mathit{typ}) & \mathsf{provs}(\mathit{kenv}.\mathit{cls} \mapsto \nu) & \overset{\mathsf{def}}{=} & \mathsf{provs}(\mathit{cls}) \cup \{\nu\} \end{array}$$

Located in package

$$\boxed{\mathsf{located}_{\Xi}(\nu)}$$

$$\mathsf{located}_{(\!|\, \Phi \,;\, \mathcal{L} \,|\!)}(\nu) \ \overset{\mathsf{def}}{\Leftrightarrow} \ \exists \ell, \alpha_\ell, \tau_\ell, \omega_\ell : \left(\begin{array}{l} \ell \mapsto \alpha_\ell \in \mathcal{L} \ \wedge \ \alpha_\ell{:}\tau_\ell{}^{-}@\,\omega_\ell \in \Phi \ \wedge \\ \left(\begin{array}{l} \alpha_\ell = \nu \\ \vee \ \exists \mathit{espc} : (\mathit{espc} \in \tau_\ell \ \wedge \ \mathsf{ident}(\mathit{espc}) = \nu) \\ \vee \ \exists \mathit{fact} : (\mathit{fact} \in \omega_\ell \ \wedge \ \mathsf{ident}(\mathit{fact}) = \nu) \end{array}\right) \end{array}\right)$$

Figure 8.3: Definition of thinning on package types and auxiliary definitions.

UNIFICATION ON SHAPES

UNIFYING MERGE ON PACKAGE SHAPES

$$\boxed{\Vdash\ \widehat{\Xi} + \widehat{\Xi} \Rightarrow \widehat{\Xi}}$$

$$\frac{\begin{array}{c}\widehat{\Xi}_1 = (\widehat{\Phi}_1; \mathcal{L}_1) \quad \widehat{\Xi}_2 = (\widehat{\Phi}_2; \mathcal{L}_2) \quad \overline{(\nu_1, \nu_2)} = ((\mathcal{L}_1(\ell), \mathcal{L}_2(\ell)) \mid \ell \in \mathsf{dom}(\mathcal{L}_1) \cap \mathsf{dom}(\mathcal{L}_2)) \\ \Vdash \left(\overline{\nu_1}; \widehat{\Phi}_1|_{\overline{\nu_1}}\right) \leftrightarrows \left(\overline{\nu_2}; \widehat{\Phi}_2|_{\overline{\nu_2}}\right) \rightsquigarrow \theta \quad \widehat{\Xi} = \mathsf{apply}(\theta; \widehat{\Xi}_1) \oplus \mathsf{apply}(\theta; \widehat{\Xi}_2)\end{array}}{\Vdash\ \widehat{\Xi}_1 + \widehat{\Xi}_2 \Rightarrow \widehat{\Xi}} \text{(UNIFMERGE)}$$

UNIFICATION OF PHYSICAL MODULE SHAPE CONTEXTS

$$\boxed{\Vdash \left(\overline{\nu}; \widehat{\Phi}\right) \leftrightarrows \left(\overline{\nu}; \widehat{\Phi}\right) \rightsquigarrow \theta}$$

$$\frac{\Vdash \overline{\nu_1} \leftrightarrows \overline{\nu_2} \rightsquigarrow \theta \qquad \Vdash \overline{\theta\widehat{\tau}_1.\mathit{espcs}} \leftrightarrows \overline{\theta\widehat{\tau}_2.\mathit{espcs}} \rightsquigarrow \theta' \qquad \Vdash \overline{\theta'\theta\widehat{\omega}_2} \leftrightarrows \overline{\theta'\theta\widehat{\omega}_2} \rightsquigarrow \theta''}{\Vdash \left(\overline{\nu_1}; \overline{\nu_1{:}\widehat{\tau}_1^{m_1} @ \widehat{\omega}_1}\right) \leftrightarrows \left(\overline{\nu_2}; \overline{\nu_2{:}\widehat{\tau}_2^{m_2} @ \widehat{\omega}_2}\right) \rightsquigarrow \theta'' \circ \theta' \circ \theta} \text{(UNIFPHYSCTX)}$$

UNIFICATION ON SINGLE OBJECTS

$$\boxed{\Vdash \nu \leftrightarrows \nu \rightsquigarrow \theta} \quad \boxed{\Vdash \mathit{espcs} \leftrightarrows \mathit{espcs} \rightsquigarrow \theta} \quad \boxed{\Vdash \widehat{\omega} \leftrightarrows \widehat{\omega} \rightsquigarrow \theta}$$

$$\Vdash \nu_1 \leftrightarrows \nu_2 \rightsquigarrow \theta \quad \overset{\text{def}}{\Leftrightarrow} \quad \mathsf{unify}(\nu_1 \doteq \nu_2) = \theta$$

$$\frac{\begin{array}{c}\mathsf{name}(\mathit{espc}_1) = \mathsf{name}(\mathit{espc}_2) \qquad \Vdash \mathsf{ident}(\mathit{espc}_1) \leftrightarrows \mathsf{ident}(\mathit{espc}_1) \rightsquigarrow \theta \\ \Vdash \left(\overline{\theta\mathit{espc}_1'}\right) \leftrightarrows \left(\overline{\theta\mathit{espc}_2'}\right) \rightsquigarrow \theta'\end{array}}{\Vdash \left(\mathit{espc}_1, \overline{\mathit{espc}_1'}\right) \leftrightarrows \left(\mathit{espc}_2, \overline{\mathit{espc}_2'}\right) \rightsquigarrow \theta' \circ \theta} \text{(UNIFESPCS)}$$

$$\frac{\forall\ (\mathit{espc}_1 \in \mathit{espcs}_1, \mathit{espc}_2 \in \mathit{espcs}_2):\ \mathsf{name}(\mathit{espc}_1) \neq \mathsf{name}(\mathit{espc}_2)}{\Vdash \mathit{espcs}_1 \leftrightarrows \mathit{espcs}_2 \rightsquigarrow \mathsf{id}} \text{(UNIFESPCSTRIV)}$$

$$\frac{\begin{array}{c}\mathsf{head}(\mathit{fact}_1) = \mathsf{head}(\mathit{fact}_2) \qquad \Vdash \mathsf{ident}(\mathit{fact}_1) \leftrightarrows \mathsf{ident}(\mathit{fact}_2) \rightsquigarrow \theta \\ \Vdash \left\{\overline{\mathsf{head}(\mathit{fact}_1') \mapsto \theta\mathsf{ident}(\mathit{fact}_1')}\right\} \leftrightarrows \left\{\overline{\mathsf{head}(\mathit{fact}_2') \mapsto \theta\mathsf{ident}(\mathit{fact}_2')}\right\} \rightsquigarrow \theta'\end{array}}{\Vdash \left\{\mathit{fact}_1, \overline{\mathit{fact}_1'}\right\} \leftrightarrows \left\{\mathit{fact}_2, \overline{\mathit{fact}_2'}\right\} \rightsquigarrow \theta' \circ \theta} \text{(UNIFWORLDSHP)}$$

$$\frac{\forall\ (\mathit{fact}_1 \in \omega_1, \mathit{fact}_2 \in \omega_2):\ \mathsf{head}(\mathit{fact}_1) \neq \mathsf{head}(\mathit{fact}_2)}{\Vdash \omega_1 \leftrightarrows \omega_2 \rightsquigarrow \mathsf{id}} \text{(UNIFWORLDSHPTRIV)}$$

SEQUENTIAL UNIFICATION ON VECTORS

$$\boxed{\Vdash \overline{\mathcal{U}} \leftrightarrows \overline{\mathcal{U}} \rightsquigarrow \theta} \quad \text{for } \mathcal{U} \in \{\nu, \mathit{espcs}, \widehat{\omega}\}$$

$$\frac{}{\Vdash \cdot \leftrightarrows \cdot \rightsquigarrow \mathsf{id}} \text{(UNIFVECNIL)} \qquad \frac{\Vdash \mathcal{U}_1 \leftrightarrows \mathcal{U}_2 \rightsquigarrow \theta \qquad \Vdash \overline{\theta\mathcal{U}_1'} \leftrightarrows \overline{\theta\mathcal{U}_2'} \rightsquigarrow \theta'}{\Vdash \mathcal{U}_1, \overline{\mathcal{U}_1'} \leftrightarrows \mathcal{U}_2, \overline{\mathcal{U}_2'} \rightsquigarrow \theta' \circ \theta} \text{(UNIFVECCONS)}$$

Figure 8.4: Definition of unification on various shape objects. All unification is predicated on a core unification definition, unify(−), on module identities, which performs unification on first-order recursive terms (A.1.1).

Package Shaping Judgments

Shaping package bindings

$$\boxed{\Delta;\widehat{\Gamma} \Vdash B \Rightarrow \widehat{\Xi}}$$

$$\frac{\ell' \mapsto \nu \in \widehat{\Gamma}}{\Delta;\widehat{\Gamma} \Vdash \ell = \ell' \Rightarrow (\, \cdot \,;\, \ell \mapsto \nu \,)}\ (\textsc{ShPkgAlias})$$

$$\frac{\nu_0 = \mathsf{mkident}(M;\widehat{\Gamma}.\mathcal{L}) \qquad \widehat{\Gamma};\nu_0 \Vdash M \Rightarrow \widehat{\tau} @ \widehat{\omega}}{\Delta;\widehat{\Gamma} \Vdash \ell = [M] \Rightarrow (\, \nu_0{:}\widehat{\tau}^{+} @ \widehat{\omega} \,;\, \ell \mapsto \nu_0 \,)}\ (\textsc{ShPkgMod})$$

$$\frac{\alpha,\overline{\beta}\ \mathsf{fresh} \qquad \widehat{\Gamma};\overline{\beta} \Vdash S \Rightarrow \widehat{\sigma} @ \widehat{\omega} \mid \widehat{\Phi}_{\mathsf{sig}} \qquad \widehat{\Phi}' = \alpha{:}\widehat{\sigma}^{-} @ \widehat{\omega},\, \widehat{\Phi}_{\mathsf{sig}}}{\Delta;\widehat{\Gamma} \Vdash \ell :: [S] \Rightarrow (\, \widehat{\Phi}' \,;\, \ell \mapsto \alpha \,)}\ (\textsc{ShPkgSig})$$

$$\frac{\overline{\alpha}\ \mathsf{fresh} \qquad (P : \forall\overline{\alpha}.\Xi) \in \Delta \qquad \vdash \Xi \xrightarrow{t} \Xi' \qquad \Xi'' = \mathsf{rename}(r;\Xi')}{\Delta;\widehat{\Gamma} \Vdash \textbf{include}\ P\ t\ r \Rightarrow \mathsf{shape}(\Xi'')}\ (\textsc{ShPkgInc})$$

Shaping sequences of package bindings

$$\boxed{\Delta \Vdash \overline{B} \Rightarrow \widehat{\Xi}}$$

$$\frac{}{\Delta \Vdash \cdot \Rightarrow (\,\cdot\,;\,\cdot\,)}\ (\textsc{ShPkgNil})$$

$$\frac{\Delta \Vdash \overline{B_1} \Rightarrow \widehat{\Xi}_1 \qquad \Delta;\widehat{\Xi}_1 \Vdash B_2 \Rightarrow \widehat{\Xi}_2 \qquad \Vdash \widehat{\Xi}_1 + \widehat{\Xi}_2 \Rightarrow \widehat{\Xi}}{\Delta \Vdash \overline{B_1}, B_2 \Rightarrow \widehat{\Xi}}\ (\textsc{ShPkgSeq})$$

Logical renaming

$$\boxed{\mathsf{rename}(r;\Xi)\ :_{\mathsf{par}}\ \Xi} \qquad \boxed{\mathsf{rename}(r;\widehat{\Xi})\ :_{\mathsf{par}}\ \widehat{\Xi}}$$

$$\begin{aligned}
\mathsf{rename}(r;\llparenthesis\, \Phi\,;\,\mathcal{L}\,\rrparenthesis) &\overset{\mathsf{def}}{=} \llparenthesis\, \Phi\,;\,\mathsf{rename}(r;\mathcal{L})\,\rrparenthesis\\
\mathsf{rename}(r;(\, \widehat{\Phi}\,;\,\mathcal{L}\,)) &\overset{\mathsf{def}}{=} (\, \widehat{\Phi}\,;\,\mathsf{rename}(r;\mathcal{L})\,)
\end{aligned}$$

$$\begin{aligned}
\mathsf{rename}(\langle\cdot\rangle;\mathcal{L}) &\overset{\mathsf{def}}{=} \mathcal{L}\\
\mathsf{rename}(\langle \ell_1 \mapsto \ell_2, \overline{\ell_1' \mapsto \ell_2'}\rangle;\mathcal{L}) &\overset{\mathsf{def}}{=} \mathsf{rename}(\langle\overline{\ell_1' \mapsto \ell_2'}\rangle;\mathcal{L}') \quad \text{if } \begin{cases} \mathcal{L} = \ell_1 \mapsto \nu, \overline{\ell' \mapsto \nu'}\\ \mathcal{L}' = \ell_2 \mapsto \nu, \overline{\ell' \mapsto \nu'} \end{cases}
\end{aligned}$$

Figure 8.5: Definition of package shaping judgments.

PACKAGE TYPING JUDGMENTS

TYPING PACKAGE BINDINGS $\boxed{\Delta; \Gamma; \hat{\Xi}_{\mathsf{pkg}} \vdash B : \Xi}$

$$\frac{\ell' \mapsto \nu \in \Gamma}{\Delta; \Gamma; \hat{\Xi}_{\mathsf{pkg}} \vdash \ell = \ell' : \llparenthesis \cdot\ ;\ \ell \mapsto \nu \rrparenthesis}\ (\textsc{TyPkgAlias})$$

$$\frac{\ell \mapsto \nu_0 \in \hat{\Xi}_{\mathsf{pkg}} \qquad \Gamma; \nu_0 \vdash M : \tau @ \omega}{\Delta; \Gamma; \hat{\Xi}_{\mathsf{pkg}} \vdash \ell = [M] : \llparenthesis \nu_0{:}\tau^{+} @ \omega\ ;\ \ell \mapsto \nu_0 \rrparenthesis}\ (\textsc{TyPkgMod})$$

$$\frac{\begin{array}{c}\ell \mapsto \nu_0 \in \hat{\Xi}_{\mathsf{pkg}} \qquad (\nu_0{:}\hat{\tau}_0^{m} @ \hat{\omega}_0) \in \hat{\Xi}_{\mathsf{pkg}} \\ \Gamma;\ (\hat{\tau}_0 @ \hat{\omega}_0) \vdash S : \sigma @ \omega \mid \Phi_{\mathsf{sig}} \qquad \Phi' = \nu_0{:}\sigma^{-} @ \omega \oplus \Phi_{\mathsf{sig}}\ \mathsf{defined}\end{array}}{\Delta; \Gamma; \hat{\Xi}_{\mathsf{pkg}} \vdash \ell :: [S] : \llparenthesis \Phi';\ \ell \mapsto \nu_0 \rrparenthesis}\ (\textsc{TyPkgSig})$$

$$\frac{\begin{array}{c}\overline{\alpha}\ \mathsf{fresh} \qquad (P : \forall\overline{\alpha}.\Xi) \in \Delta \qquad \vdash \Xi \xrightarrow{t} \Xi' \qquad \Xi'' = \mathsf{rename}(r; \Xi') \\ \overline{\alpha}' = \overline{\alpha} \cap \mathsf{dom}(\Xi''.\Phi) \qquad \vdash \hat{\Xi}_{\mathsf{pkg}} \leqslant_{\overline{\alpha'}} \Xi'' \rightsquigarrow \theta \qquad \mathsf{apply}(\theta; \Xi'')\ \mathsf{defined}\end{array}}{\Delta; \Gamma; \hat{\Xi}_{\mathsf{pkg}} \vdash \textbf{include}\ P\ t\ r : \mathsf{apply}(\theta; \Xi'')}\ (\textsc{TyPkgInc})$$

MATCHING PACKAGE TYPES AGAINST SHAPES $\boxed{\vdash \hat{\Xi}_{\mathsf{pkg}} \leqslant_{\overline{\alpha}} \Xi \rightsquigarrow \theta}$

$$\frac{\theta = \{\overline{\alpha := \nu}\} \qquad \mathsf{fv}(\overline{\nu}) \mathbin{\#} \overline{\alpha} \qquad \hat{\Xi}_{\mathsf{pkg}} \leqslant \mathsf{apply}(\theta; \mathsf{shape}(\Xi))}{\vdash \hat{\Xi}_{\mathsf{pkg}} \leqslant_{\overline{\alpha}} \Xi \rightsquigarrow \theta}\ (\textsc{PkgMatch})$$

TYPING SEQUENCES OF PACKAGE BINDINGS $\boxed{\Delta; \hat{\Xi}_{\mathsf{pkg}} \vdash \overline{B} : \Xi}$

$$\frac{}{\Delta; \hat{\Xi}_{\mathsf{pkg}} \vdash \cdot : \llparenthesis \cdot\,;\, \cdot \rrparenthesis}\ (\textsc{TyPkgNil})$$

$$\frac{\Delta; \hat{\Xi}_{\mathsf{pkg}} \vdash \overline{B_1} : \Xi_1 \qquad \Delta; \Xi_1; \hat{\Xi}_{\mathsf{pkg}} \vdash B_2 : \Xi_2 \qquad \Xi = \Xi_1 \oplus \Xi_2\ \mathsf{defined}}{\Delta; \hat{\Xi}_{\mathsf{pkg}} \vdash \overline{B_1}, B_2 : \Xi}\ (\textsc{TyPkgSeq})$$

TYPING PACKAGE DEFINITIONS $\boxed{\Delta \vdash D : \forall\overline{\alpha}.\Xi}$

$$\frac{\Delta \Vdash \overline{B} \Rightarrow \hat{\Xi}_{\mathsf{pkg}} \qquad \Delta; \hat{\Xi}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \qquad \vdash \Xi \xrightarrow{t} \Xi' \qquad \overline{\alpha} = \mathsf{fv}(\Xi'.\Phi)}{\Delta \vdash \textbf{package}\ P\ t\ \textbf{where}\ \overline{B} : \forall\overline{\alpha}.\Xi'}\ (\textsc{TyPkg})$$

TYPING PACKAGE REPOSITORIES $\boxed{\Delta \vdash R}$

$$\frac{}{\Delta \vdash \cdot}\ (\textsc{TyPkgRepoNil}) \qquad \frac{\Delta \vdash D : \forall\overline{\alpha}.\Xi \qquad \Delta, P : \forall\overline{\alpha}.\Xi \vdash \overline{D'}}{\Delta \vdash D, \overline{D'}}\ (\textsc{TyPkgRepoSeq})$$

Figure 8.6: Definition of package typing judgments.

9 ELABORATION

The Backpack formalization thus far has comprised typing judgments and metatheory for three conceptual levels of the system. Notably absent from this formalization has been any presentation of the *elaboration semantics* described in §3.4. In that section I painted the overall picture of the elaboration semantics: how the elaboration *compiles away* the new package level of Backpack by *rewriting* all the module expressions into Haskell module files and using *binary interface files* to represent the module types of signatures. That presentation lacked the *worlds* introduced in Chapter 4, however; in that chapter was a sketch of how to extend the IL.

All of the formalization presented so far has been the Backpack *External Language* (EL), which is a language for expressing modular programs. In this chapter I extend the Backpack formalization with the *Internal Language* (IL), which is a formal model of GHC Haskell modules in a file sytem. Concretely, I define the *elaboration semantics* for translating (well-typed) EL expressions into IL expressions; and, as the primary technical validation of this semantics, I state and prove an *Elaboration Soundness* theorem, conveying that well-typed EL (package) expressions elaborate to well-typed IL (directory) expressions whose (package) types have identical structure.

9.1 SYNTAX AND SEMANTIC OBJECTS OF THE IL

The syntax of the IL is presented in Figure 9.1. Many forms use the metavariables from the corresponding forms in the EL; they are syntactically identical except that module identities ν are replaced with module file names f. In order to disambiguate we sometimes use a circle above the metavariable to mean the IL variant, *e.g.*, $\mathring{dspc}$ for an IL core entity spec. Auxiliary definitions for IL syntax are defined in the appendix (§A.2).

Recall from §3.4 that import and export syntax in the IL is a restriction of that of the EL: all imports and exports are explicit. This restriction is sufficient to capture the target of our elaboration semantics, and it's necessary to ensure certain metatheoretic properties, like a *Weakening* property on module typing in the IL.

While the EL contained three levels—core, module, and package—the IL contains forms corresponding only to the former two. The entire point of the elaboration of the EL into the IL is to shed the package level. In its place are *directories of module source files*, whose main syntactic form is the *directory expression* (*dexp*). The stratification of the EL into the three levels was primarily conceptual and for presentation; the IL doesn't necessitate this stratification since it's mostly a replica of the EL. The IL is presented here all at once, including core, module, and directory.

An IL directory expression does not model only the Haskell source files which the user writes; it models those source files *along with their types*, and the types of any as-yet undefined modules. As a consequence, and because they may refer to each other as a mutually recursive knot, directory expressions do not need to be statically analyzed in any dependency order. This is not unlike the typing of, say, top-level value definitions in a fully type-annotated Haskell module.

The primary addition from the initial presentation in §3.4 is the syntactic forms for *worlds*. How exactly to model them in the formalization was a tricky matter that took some time to find a satisfying answer. The reason for the trickiness is the tension between, on one hand, making the IL syntax and semantics directly correspond to those of the EL, and on the other hand, making the IL syntax and semantics model the actually-existing module system of

Backpack IL Syntax and Semantic Objects

Module File Names	f, g	$\in$	*ILModNames*	
Module Source Files	*hsmod*	$::=$	$\texttt{module}\ \mathsf{f}\ \mathring{expdecl}\ \texttt{where}\ \mathring{impdecls}\texttt{;}\ \mathring{defs}$	
Module Source Types	*ftyp*	$::=$	$\langle\!\vert\ \mathring{dspcs}\ ;\ \mathring{espcs}\ ;\ \overline{\mathsf{f}}\ \vert\!\rangle$	
Typed Source Expressions	*tfexp*	$::=$	$hsmod : ftyp$	Module Source Files
		$\vert$	$- : ftyp$	Stubbed Modules
Directory Expressions	*dexp*	$::=$	$\{\ \overline{\mathsf{f} \mapsto tfexp\ @\ \mathring{\omega}}\ \}$	
File Environments	*fenv*	$::=$	$\{\!\vert\ \mathsf{f}{:}ftyp^{\mathsf{m}} @\ \mathring{\omega}\ \vert\!\}$	
Import Declarations	$\mathring{impdecls}$	$::=$	$\texttt{import}\ \mathsf{f}\ \texttt{as}\ \mathsf{f}'\ \mathring{impspec}$	Unqualified Import
		$\vert$	$\texttt{import qualified}\ \mathsf{f}\ \texttt{as}\ \mathsf{f}'\ \mathring{impspec}$	Qualified Import
Entity Import Lists	$\mathring{impspec}$	$::=$	$\texttt{(}\overline{\mathring{import}}\texttt{)}$	
Entity Import Specs	$\mathring{import}$	$::=$	χ	Simple Entity
		$\vert$	$\chi\texttt{(}\overline{\chi}'\texttt{)}$	Entity with Subordinates
Export Declarations	$\mathring{expdecl}$	$::=$	$\texttt{(}\overline{\mathring{export}}\texttt{)}$	
Entity Export Specs	$\mathring{export}$	$::=$	$\mathring{eref}$	Simple Entity
		$\vert$	$\mathring{eref}\texttt{(}\overline{\chi}'\texttt{)}$	Entity with Subordinates

Figure 9.1: Syntax and semantic objects of Backpack's IL. Disambiguation between EL and IL metavariables for the same concept are done by annotating the *IL* metavariables with $\mathring{A}$ and keeping A as the *EL* metavariable, but this annotation is often elided for simplicity.

(GHC) Haskell. In the end, in order to facilitate the development of the formalization and the proof of Elaboration Soundness, the former was chosen.

Worlds in the IL are semantic objects that carry the $\mathsf{consistent}(-)$ invariant just like in the EL. The key difference lies in their construction and in their role in module typings in a context, as we'll soon see.

9.2 SEMANTICS OF THE IL

The semantics of the IL are mostly similar to those of the core and module levels of the EL, considering only the *typing* (not shaping) and *module* (not signatures) fragment of the EL. The two main points of departure with (this fragment of) the EL semantics are the *directories* in the IL, which are an (intentionally) dumbed-down shadow of packages, and the type annotations (*ftyp*) on expressions.

TYPING OF DIRECTORY EXPRESSIONS Whereas EL package-level expressions operate on a physical structure of modules—and abstract, undefined modules—through manipulation of logical module names, IL directory expressions fully describe the overall physical structure *in situ*. As a result, the well-formedness of IL directory expressions is considerably simpler than that of EL packages: one simply type-checks the module files in the directory in any order.

The typing judgment for directory expressions, $fenv \vdash dexp$, characterizes those expressions whose files are all well-formed, with types and worlds matching their annotations, in the ambient context *extended with themselves*. The context in the premise of the (ILTYDEXP) reveals that last part. Like with the well-formedness of physical module contexts in the EL, well-formedness of a directory expression must "assume itself" (§7.7.1).

TYPING OF TYPED FILE EXPRESSIONS The judgment for typed file expressions, $fenv; \mathsf{f}_0 \vdash tfexp\ @\ \omega$, characterizes those module source files whose synthesized type and world, from a

IL Typing Judgments

Typing of directory expressions

$\boxed{\mathit{fenv} \vdash \mathit{dexp}}$

$$\frac{\mathit{fenv} \vdash \lfloor \mathit{dexp} \rfloor\ \mathsf{wf} \qquad \forall \mathsf{f} \mapsto \mathit{tfexp} \,@\, \mathring{\omega} \in \mathit{dexp} \;:\; \mathit{fenv} \oplus \lfloor \mathit{dexp} \rfloor ; \mathsf{f} \vdash \mathit{tfexp} \,@\, \mathring{\omega}}{\mathit{fenv} \vdash \mathit{dexp}}\ (\textsc{IlTyDexp})$$

Typing of typed file expressions

$\boxed{\mathit{fenv}; \mathsf{f}_0 \vdash \mathit{tfexp} \,@\, \mathring{\omega}}$

$$\frac{\mathsf{name}(\mathit{hsmod}) = \mathsf{f}_0 \qquad \mathit{fenv} \vdash \mathit{hsmod} : \mathit{ftyp} \,@\, \mathring{\omega}' \qquad \mathring{\omega}' \sqsupseteq \mathring{\omega} \sqsupseteq \mathsf{extworld}(\mathsf{f}_0; \mathit{ftyp})}{\mathit{fenv}; \mathsf{f}_0 \vdash (\mathit{hsmod} : \mathit{ftyp}) \,@\, \mathring{\omega}}\ (\textsc{IlTyHsmod})$$

$$\frac{\mathit{fenv} \vdash \mathit{ftyp}\ \mathsf{wf} \qquad \mathit{fenv} \vdash \mathring{\omega}\ \mathsf{wf} \qquad \mathring{\omega} \sqsupseteq \mathsf{extworld}(\mathsf{f}_0; \mathit{ftyp})}{\mathit{fenv}; \mathsf{f}_0 \vdash (- : \mathit{ftyp}) \,@\, \mathring{\omega}}\ (\textsc{IlTyStub})$$

Flattening directories to environments

$\boxed{\lfloor \mathit{dexp} \rfloor \;:\; \mathit{fenv}}$

$$\lfloor \{ \overline{\mathsf{f} \mapsto \mathit{tfexp} \,@\, \mathring{\omega}} \} \rfloor \stackrel{\text{def}}{=} \{\!\!\{ \overline{\mathsf{f}{:}\mathsf{typ}(\mathit{tfexp})^{\mathsf{pol}(\mathit{tfexp})} \,@\, \mathring{\omega}} \}\!\!\}$$

World extraction

$\boxed{\mathsf{extworld}(\mathsf{f}; \mathit{ftyp}) \;:\; \mathring{\omega}}$

$$\mathsf{extworld}(\mathsf{f}; \langle\!\langle\, \mathit{dspcs} \,;\, \mathit{espcs} \,;\, \overline{\mathsf{f}} \,\rangle\!\rangle) \stackrel{\text{def}}{=} \{\!\!\{ \mathsf{head}(\mathit{dspc}) \mapsto \mathsf{f} \mid \mathring{\mathit{dspc}} \in \mathring{\mathit{dspcs}} \text{ s.t. } \mathring{\mathit{dspc}} = \mathtt{instance} \ldots \}\!\!\}$$

Auxiliary definitions

$\boxed{\mathsf{pol}(\mathit{tfexp}) \;:\; \mathfrak{m}}$ $\boxed{\mathsf{typ}(\mathit{tfexp}) \;:\; \mathfrak{m}}$

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{pol}(\mathit{hsmod} : \mathit{ftyp}) & \stackrel{\text{def}}{=} & + & \mathsf{typ}(\mathit{hsmod} : \mathit{ftyp}) & \stackrel{\text{def}}{=} & \mathit{ftyp} \\
\mathsf{pol}(- : \mathit{ftyp}) & \stackrel{\text{def}}{=} & - & \mathsf{typ}(- : \mathit{ftyp}) & \stackrel{\text{def}}{=} & \mathit{ftyp}
\end{array}$$

Figure 9.2: Definition of well-formedness judgments for IL expressions and some required auxiliary functions.

module typing judgment (explained shortly), match those of its annotations. Like in the EL, part of the judgment is the identity of the module being analyzed; here in the IL, that's some module name f_0 that this file was assigned in some ambient directory expression.

Moreover, in both forms of typed file expressions, the world annotation ω must extend the *extracted world* of the type annotation *ftyp*. The extracted world, an operation defined by $\mathsf{extworld}(\mathsf{f}_0; \mathit{ftyp})$, is the world constructed from the locally declared `instance` specs in *ftyp.dspcs*, with each identifying f_0 as its defining module.[1] This part of both rules is necessary for the proofs of the metatheory; in particular, the proof of *Invariance under Substitution* (Lemma A.104) requires this condition of well-formed typed file expressions.

Rule (IlTyHsmod) for source files appeals to the (IL) typing judgment for the Haskell module (*hsmod*), checking that its synthesized module type, *ftyp*, is identical to the annotation on the typed file expression. The world annotation ω, however, must be some parent world of the synthesized one, ω'; we'll return to this distinction shortly. As just mentioned, the world annotation must extend the extracted world of the type.

Rule (IlTyStub) is straightforward.

1 This operation existed also in the EL, *e.g.*, in the definition of $\mathsf{apply}(\theta; \Phi)$ (Figure 7.4), but it was merely left undefined then.

TYPING OF MODULE SOURCE FILES Figure 9.3 presents the typing judgment for module source files. This judgment is structurally identical to *module typing* in the Backpack EL (*e.g.*, see that definition in Figure 7.10). The only distinction is in the exact "passing of arguments" to the core environment construction judgment, which in the IL avoids the complication of a realizer.

As in the EL, the typing of core definitions is undefined and axiomatized.

The figure also presents the core environment construction, import resolution, and export resolution judgments, all of which similarly resemble their EL counterparts, with the exception of the construction of the world $\mathring{\omega}$; again, we'll return to this point shortly. The syntax of imports and exports has been restricted to be entirely explicit:

- Rules (IlImpDeclUnqual) and (IlImpDeclQual) no longer allow for the absence of aliases.

- Rules (IlImpSimple) and (IlImpSub) allow a more rigid form of entity imports than in the EL. A single entity reference χ cannot refer to the name of an entity with subordinates, and there's no rule (or syntax) to designate all an entity's subordinate names to be imported.

- Export declarations are fully explicit; there's no rule (or syntax) to implicitly export local entities or all entities in some logical module `scope`.

- Rules (IlExpSimple) and (IlExpSub) are like the import rules: a single entity name cannot designate entities with subordinates, and entity references with subordinates must designate an explicit list.

As we will soon see, these judgments have been defined with the Elaboration Soundness theorem (and proof) in mind.

DIFFERENT WORLD SEMANTICS The fundamental distinction in EL and in IL typing is the determination of module worlds. The reader may have even missed that tiny notational distinction in (IlCoreEnv). In the IL, worlds are determined, as in Haskell, *from the worlds of all transitive dependencies of the module*, denoted $\mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{imps}(\mathring{\mathit{impdecls}}))$.[2] Back in the EL, however, worlds were determined from simply the worlds of the module's immediate imports, denoted $\mathsf{world}_{\Gamma}(\mathsf{imps}(\mathit{impdecls}))$. Figure 9.4 presents this new definition, as a straightforward application of existing definitions elsewhere in the formalization.

A few questions about this different world semantics naturally arise:

- What's the reason for the different semantics in the IL? Well, if the elaboration should give meaning to Backpack in terms of actually-existing Haskell, the IL should then faithfully model (a slight syntactic restriction of) actually-existing Haskell. And in Haskell, the instances known and available to a module are all those defined in any module in the transitive closure of the module's imports.

- What's the point of keeping the "annotated worlds" on typed file expressions (*tfexp*)? (Recall that in (IlTyHsmod) the world constructed for the module from its transitive imports, $\mathring{\omega}'$, is not used as the type of the *tfexp* that contains the module; that world is instead the annotated world $\mathring{\omega}$ which must be extended by $\mathring{\omega}'$.) The primary purpose of annotated worlds is to regularize the intentional correspondence between EL semantic objects and IL semantic objects. Whether it's an EL physical module context (Φ) or an IL file environment (*fenv*), or an EL module type (τ) or an IL module source type (*ftyp*), the whole Backpack formalization was set up so that these objects could be defined once

2 To be more precise, $\mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{f})$ isn't defined as the transitive imports of f in *fenv*, but rather as the closure of all module file names—read: module identities—that f's type and world mention in *fenv*, *i.e.*, the (translation of the) $\mathsf{depends}^{+}_{\Phi}(\nu)$ definition for thinning back in Figure 8.3. That's because stub files in an IL environment do not record their direct imports, so there's no way to accurately walk the module-and-signature import chains—just as how EL module types of holes do not record imports, thereby necessitating this notion of dependencies.

and reused between the two languages, along with their attendant well-formedness and algebraic definitions. Indeed, the proof of Elaboration Soundness rests on various metatheory about this direct connection—most notably in the fact that EL objects relate to their translations in the IL by replacing module identities, ν, with the injection into IL file names, $\nu^{\star}$; see Appendix Ch. D for example.

- When typing a module in the EL it has only the worlds of its immediate imports to consider for possible inconsistency, but when typing its elaboration in the IL it has all the worlds of its *transitive* imports to consider—a much stronger requirement. So where does the proof burden for that come from if not the original EL typing? An astute question indeed. As we'll see in the next section, the metatheory of the IL imposes various *world preservation* conditions that are readily discharged by my heavyweight *Package-Level Consistency* design choice in the EL.

9.3 METATHEORY OF THE IL

Whereas the EL's metatheory was concentrated on properties about the various *well-formedness* judgments, the IL's metatheory is concentrated on properties about the *typing* judgments, *i.e.*, about *terms* rather than *types*. That's because the key desired property in the elaboration semantics is that IL terms, after transformations are performed on them in lock-step with transformations on the EL terms that elaborate to them, are still well-typed and with structurally identical types. This will become more clear in the proof of *Elaboration Soundness*.

The structural properties about well-formedness presented in §8.7.1 have direct analogues in the IL, but about the typing judgments. Thematically, they are the same properties, but technically, since they're about *the typing of terms* rather than *the well-formedness of types*, they are substantially more complicated, sometimes requiring abstruse side conditions to make the proofs work. Most require axioms on the core-level typing judgment, axioms that weren't necessary in the more limited metatheory of the EL, like *Weakening* on core typing.

In particular, the proliferation of *world preservation* side conditions on each of the properties highlights a precise, technical obstacle that type classes pose in developing type systems for modules with type classes. Explaining the folklore idea that "type classes are anti-modular" by invoking this technical requirement could be considered a technical contribution of this thesis.

These side conditions refer to an auxiliary definition, $\mathsf{modworld}_{\mathit{fenv}}(\mathit{hsmod};\mathit{ftyp})$, which describes the reconstructed or *reinterpreted* world of *hsmod* in some context. This reinterpreted world is determined from (1) the worlds of modules in *fenv* that *hsmod* transitively depends on and (2) the *fact*s from local instances defined in *hsmod* and whose semantic *head*s are derived from *ftyp*. See the definition in Figure 9.4. Crucially, the reinterpreted world of a module source file and type is a *partial* function: if *hsmod* witnessed any sort of inconsistency among the instances it derives within (1), within (2), or between (1) and (2), then this function is undefined. When the world preservation conditions on the IL metatheory stipulate that a reinterpreted world is defined, that's a substantial proof burden.

In this section I present each key structural property for IL typing, all of which come into play in the proof of Elaboration Soundness, followed by a noteworthy metatheoretic property of the IL that connects back to the original introduction of world semantics. More details, including precise statements, can be found in the Appendix (§C.2).

9.3.1 *Weakening*

Weakening (Lemma A.93) states that if a term is well-typed in environment *fenv* and if $\mathit{fenv} \oplus_{?} \mathit{fenv}'$, then the term is also well-typed in $\mathit{fenv} \oplus \mathit{fenv}'$. But there's an additional condition on this statement, across almost all terms—the *world preservation* condition. Essentially, this condition says that fenv' cannot introduce any additional type class instances to any imported modules such that the importing module ceases to inhabit a consistent world.

For example, for the module typing judgment $\mathit{fenv} \vdash \mathit{hsmod} : \mathit{ftyp} \mathbin{@} \mathring{\omega}$, *Weakening* states: if the module is well-typed with some world $\mathring{\omega}$ *and if the new world the module would inhabit in the extended environment is consistent—i.e.*, world preservation—then the module will continue to be well-typed, with the same type, in the extended environment but with the (extended) new world.

The proof of *Weakening* is stuck without the world preservation premise on each part of the statement. That matches the intuition that, by merging in the additional environment fenv' (which might link in implementations for module stubs), a module might now be importing *more* type class facts than it did before, and those additional facts might clash with those defined locally in the module or with those defined in other modules it imports.

Moreover, the proof of *Weakening* requires two axioms about the core typing judgment in the IL: an obvious analogue of *Weakening* (Axiom A.89) and also a different property I call *Monotonicity* (Axiom A.90). The typing judgment for core definitions in the IL looks just like core typing in the EL; in particular, it's parameterized by the world of the surrounding module $\mathring{\omega}$. *Monotonicity* states that if some definitions are well-typed in some world, then they're still well-typed—with the exact same static specifications—in any extension of that world. This axiom models the actually-existing behavior of GHC: the type-checking of a program cannot depend on type class instances that don't exist; consequently, nothing about type-checking that program will change if more (*non-conflicting*) instances are known.[3]

9.3.2 *Merging*

Merging (Lemma A.99) states that if similarly-named tfexp_1 and tfexp_2 are well-typed in their respective environments fenv_1 and fenv_2, *and if the reinterpreted world of the merged expressions in the merged environment is consistent* (*i.e.*, if $\mathsf{modworld}_{\mathit{fenv}_1 \oplus \mathit{fenv}_2}(\mathit{tfexp}_i;\ \mathsf{defined})$), then that merged expression is well-typed in the merged environment. This property is only stated and proved for the *tfexp* typing judgment since that's the only form of it actually necessary to the metatheory—in the proof of *Cut* on *dexp*, which is the more useful property, as it turns out. *Merging* also demands no axiom about core typing.

9.3.3 *Strengthening*

Strengthening (Lemma A.100) states that if F is a set of module file names such that the dependencies[4] of F in *fenv* are contained within F itself, and if an IL term is well-typed in *fenv*, then it's also well-typed in $\mathit{fenv}|_{\mathsf{F}}$, the restriction of *fenv* to only the module files named in F. Notably, for this property, there is no world preservation premise because nothing about the file types in *fenv* actually *changes* in the conclusion of the property. More specifically, no module in the IL term sees any *conflicting* instances in the conclusion; it just sees fewer modules that it didn't transitively import.

The statement of the property for module source file typing has some noteworthy premises. It states that, for such a set F, if $\mathit{fenv} \vdash \mathit{hsmod} : \mathit{ftyp} \mathbin{@} \mathring{\omega}$ and and $\mathit{fenv}(\mathsf{f}_0) = \mathit{ftyp}$ and $\mathsf{f}_0 \in \mathsf{F}$ then $\mathit{fenv}|_{\mathsf{F}} \vdash \mathit{hsmod} : \mathit{ftyp} \mathbin{@} \mathring{\omega}$. The premise about $\mathit{fenv}(\mathsf{f}_0)$ states that *fenv*'s type annotation for this module must match the synthesized type of the judgment, *ftyp*. (Recall that, in IL module typing, a module source file has a full "recursive" view of itself and all other modules in some ambient directory expression.)

The proof of *Strengthening* on module typing requires an analogous axiom (Axiom A.91) on core-level typing of definitions. That axiom states that for some subset, F, of file names in *fenv*, if F contains the transitive closure of files mentioned in the entity environment $\mathit{e\mathring{e}nv}$

3 Edward Yang provided this key observation that led to the axiom.

4 Here the notion of dependencies follows that of thinning in the EL, as defined by the $\mathsf{depends}^+_\Phi(\nu)$ definition in Figure 8.3. A particular observation is that any orphan instances known to a module f will be represented in that module's world $\mathring{\omega}$ as a $\mathit{f\mathring{a}ct}$ whose defining file name f_o is exactly the module source file (or stub) that originally defined the orphan instance. That then means that $\mathsf{f}_o \in \mathsf{provs}(\mathring{\omega})\mathsf{depends}^+_{\mathit{fenv}}(\mathsf{f})$, and therefore modules providing orphan instances will be included in the dependencies as far as *Strengthening* is concerned.

and world $\mathring{\varpi}$, then the well-typing of definitions in *fenv* with $\mathring{\mathit{eenv}}$ and $\mathring{\varpi}$ leads to the well-typing of those definitions in $\mathit{fenv}|_{\mathsf{F}}$ and with the same $\mathring{\mathit{dspcs}}$. The axiom matches the intuition that dropping unrelated modules from the context should not change the type-checking of a module's definitions.

9.3.4 *Invariance under Substitution*

Invariance under Substitution (Lemma A.104) is the most technically complicated of the structural properties on IL typing. It states that if some file name substitution θ is applicable to well-formed environment *fenv*, then from the typing of a term in *fenv* we can derive the typing of the θ-substituted term in the θ-substituted *fenv*. Moreover, the file type in the new derivation is θ-substituted and the world is some extension of the θ-substituted original world. The intuition for that world extension (*i.e.*, (b) below) is that there may have been a stub (transitively) imported by *hsmod* that was linked by the substitution θ, and therefore θ*hsmod* would see any additional instances known to the implementation of that stub.

In this property is where side conditions, like world preservation, complicate the metatheory. Consider the statement on *hsmod* typing:

If

(i) $\mathsf{apply}(\theta; \mathit{fenv})$ defined
(ii) $\mathit{fenv} \vdash \mathit{hsmod} : \mathit{ftyp} \mathbin{@} \mathring{\varpi}$
(iii) $\mathsf{avoidaliases}(\theta; \mathit{hsmod})$
(iv) $\mathsf{apply}(\theta; \mathring{\varpi})$ defined
(v) $\exists\, \mathring{\varpi}_0 : \theta\mathit{fenv} \;\oplus_{?}\; (\theta\mathsf{name}(\mathit{hsmod})){:}(\theta\mathit{ftyp})^{+} \mathbin{@} \mathring{\varpi}_0$
(vi) $\mathsf{modworld}_{\theta\mathit{fenv}}(\theta\mathit{hsmod}; \theta\mathit{ftyp})$ defined

then

(a) $\theta\mathit{fenv} \vdash \theta\mathit{hsmod} : \theta\mathit{ftyp} \mathbin{@} \mathring{\varpi}'$
(b) $\mathring{\varpi}' \sqsupseteq \theta\mathring{\varpi}$

Premise (iii) states that the file name substitution θ does not change the import aliases in *hsmod*. Recall from §9.1 that the import syntax of the IL is restricted to always be of the form `import [qualified]` f_1 `as` f_2. This premise says that $\theta(\mathsf{f}_2) = \mathsf{f}_2$, *i.e.*, that no qualified entity reference $\mathsf{f}_2.\chi$ will be perturbed by applying θ throughout the module. And premise (vi) is the world preservation condition: if we reinterpret the world not of *hsmod* but of the substitution θ*hsmod* and in the substituted environment θ*fenv*, the resulting world is consistent.

Proving this property requires an analogous lemma on import resolution and core environment construction (Lemma A.110). In part (4) of that lemma, the potential dangers of applying θ to the well-typedness of a module are directly avoided with three premises stating that: reinterpreting the *imported* world of the substituted module in the substituted environment is okay; applying the substitution to the *locally defined* world is okay; and merging the above two worlds is okay.

9.3.5 *Cut*

Cut (Lemma A.111) states that from the well-typing of dexp_1, the mergeability of dexp_1 and dexp_2, and the well-typing of dexp_2 in the flattened $\lfloor \mathit{dexp}_1 \rfloor$, we can derive the well-typing of $\mathit{dexp}_1 \oplus \mathit{dexp}_2$. Here, world preservation manifests as a requirement that, for each module in each directory expression, its reinterpreted world in the combined environment $\lfloor \mathit{dexp}_1 \oplus \mathit{dexp}_2 \rfloor$ be consistent.

9.3.6 *World consistency*

The final noteworthy metatheoretic property of IL typing is now a straightforward statement based on all the machinery built up for the previous properties. When I first introduced the world semantics (§4.3), I described a desired *World Consistency* property for Haskell (not Backpack): every single module in a closed program should inhabit a consistent world. Not only was this property necessary to rule out misabstraction, it also gave a more modular, tighter framing to the *Global Uniqueness* property that says all type class instances defined across a closed program must be mutually consistent.

With the IL serving as a model of Haskell's type system augmented with the world semantics, we now have a foundation in place to definitively state that well-typed closed programs in the IL obey *World Consistency*.

As mentioned at the start of this section, the notion of a reinterpreted world of a module typing, $\mathsf{modworld}_{\mathit{fenv}}(\mathit{hsmod};\mathit{ftyp})$, denotes precisely the world inhabited by that module. (That's in stark contrast to the *annotated* world of the module as it exists in a closed *dexp*.) When a module's reinterpreted world is defined in a context, then the world it inhabits is consistent (since all world semantic objects are consistent). Lemma A.95, an auxiliary lemma developed for the proof of *Weakening*, shows that the reinterpreted world of a well-typed IL module is defined. Then *World Consistency* is a straightforward corollary of this lemma: if a closed *dexp* is well-typed in the empty context, then every module it defines inhabits a consistent world (Corollary A.96).

9.4 ELABORATION SEMANTICS

Up to this point the formalization has been focused on defining well-formedness of semantic objects and typing judgments characterizing expressions with those objects. But what's the *meaning* of a package expression? Here, I give the meaning of Backpack expressions, across the package and module levels, via an *elaboration semantics*.

The elaboration semantics is defined as an extension of the typing judgments of the EL package and module levels; the judgments for the package level are presented in Figure 9.5, while the single judgment and auxiliary definitions for the module level are presented in Figures 9.6 and 9.7. Each elaboration judgment has an additional " $\rightsquigarrow$ IL-expression" part that specifies the IL expression into which the well-typed EL expression elaborates.[5]

For the package level, the elaboration necessitates an extension of the package environment semantic object (Δ): it must map each package name P not just to a parameterized package type ($\forall\overline{\alpha}.\Xi$) but also to the parameterized IL directory expression ($\lambda\overline{\alpha}.\mathit{dexp}$) defining package P. The typing of a package repository must therefore store the package's elaboration along with its type when appending to its own ambient package. The absence of an expression form in the IL to which an entire package *repository* elaborates, *i.e.*, a " $\rightsquigarrow$ ..." in the consequent of (ElabPkgRepoSeq) is a gap within Backpack. Instead, package repositories internally shuffle around the elaborations of their constituent packages, observed in the premises to the same rule.

The key high-level ideas in this elaboration were presented in §3.4: naming IL modules *structurally* according to *module identities*; rewriting `import` statements to the new structured names while preserving local syntactic module names; and otherwise preserving core-level definitions.

That first idea has deep ramifications across the eleboration semantics: the embedding of module identities (ν) into IL module names (f). For this embedding we assume the existence of an *identity translation* function $(-)^{\star}$ that performs the embedding; see Appendix §A.1.1 for more details on this embedding. In addition to the image of $(-)^{\star}$, the set of IL module names, *ILModNames*, also contains the EL's logical module names (ℓ); we additionally assume that the two embeddings into IL module names are entirely distinct,[6] *i.e.*,

$$\mathit{IlModNames} \cong (\mathit{ModIdents})^{\star} + \mathit{ModNames} + \dots$$

5 Viewed differently, this judgment is a mapping from *typing derivations* in the EL to expressions in the IL.

6 In a real implementation of Backpack elaboration, this distinctness would be ensured by translating identities into internal names inexpressible in the surface syntax, like `#Foo(#Bar(),#Baz())`.

We'll soon see the details of each elaboration definition in the discussion of the *elaboration soundness* proof.

9.5 SOUNDNESS RELATION

The elaboration semantics describes the "compiling away" of the package level of the EL into plain Haskell, represented by IL terms. But what can we say about the relation between the former and the latter? Three key ideas characterize this relation:

- First, their physical structure is identical: the organization of both concrete and abstract modules is exactly the same, with a bijection between modules and their dependencies on each side of elaboration (between Ξ and *dexp*).
- Second, module types and worlds ($\tau \,@\, \omega$) correspond precisely with module file types ($\mathit{ftyp} \,@\, \mathring{\omega}$): they define, export, and import the same core entities and modules, and they inhabit worlds with the same facts.
- And third, the core-level definitions inside modules are the same: the definitions are syntactically identical modulo local imported module names, and all core entities (*phnm*) referred to in the definitions are the same.

The *soundness relation* formalizes this connection, defined as the binary relations $\Xi \sim \mathit{dexp}$ and $(\nu{:}\tau^{m} \,@\, \omega) \sim (f \mapsto \mathit{tfexp} \,@\, \mathring{\omega})$, and presented in Figure 9.8. This judgment states that (the physical part of) the package type Ξ and the directory expression *dexp* have *related* module contents according to the three key ideas above.

The definition of the relation is mostly straightforward, but three points about the definition of the relation between concrete modules and module source files deserve mention.

First, the injective translation function from EL identities to IL names, $(-)^{\star}$, plays a key role in relating EL objects to IL objects: they're the same but with the congruent extension of the injective translation function.

Second, the EL side of the relation has no actual *terms* (*i.e.*, no package or module *expressions*). The singleton module context is related to a singleton directory expression containing a module source file *hsmod*. What does it mean for that module to be related to $\nu{:}\tau^{+} \,@\, \omega$? The module must simply be in the image of the elaboration of modules, as defined via the $\mathsf{mkmod}(-)$ function, for *some* objects and, critically, the same module identity ν and export specs $\tau.\mathit{espcs}$. In this way, the relation is tied to Backpack's particular elaboration semantics.

Third, here we can see the elegance of the annotated worlds on typed file expressions in the IL: they preserve the symmetry between EL typings/objects and IL typings/objects. (Recall from §9.2 that these are not the same as the worlds actually inhabited by their modules.)

9.6 ELABORATION SOUNDNESS

With the elaboration semantics and relation in mind, we are finally equipped to state and prove the key Elaboration Soundness Theorem of Backpack's formalization.

The Elaboration Soundness Theorem is the central technical validation of the formalization and of the whole idea of packages as logical structure affixed on top of physical structure. Proving it drove much of the development of the formalization, as demonstrated by the sometimes abstruse qualification on earlier metatheory, *e.g.*, the world preservation conditions on the structural properties of IL typing (§9.3).

Elaboration Soundness is defined on the module typing and package-level typing judgments as follows:

Theorem (Elaboration Soundness).

Assume $\Gamma = (\!|\, \Phi \,;\, \mathcal{L} \,|\!)$, *where applicable.*
(1) *If* $\cdot \vdash \Gamma$ *wf and* $\Gamma; \nu_0 \vdash M : \tau \,@\, \omega \rightsquigarrow$ *hsmod and* $\Phi \oplus_{?} \nu_0{:}\tau^{+} \,@\, \omega$, *then* $\Phi^{\star} \vdash \mathit{hsmod} : \tau^{\star} \,@\, \mathring{\omega}'$ *and* $\mathring{\omega}' \sqsupseteq \omega^{\star}$.

(2) *If* $\vdash \Delta$ *wf and* $\cdot \vdash \Gamma$ *wf and* $\Delta;\Gamma;\hat{\Xi}_{pkg} \vdash B : \Xi \rightsquigarrow$ *dexp and* $\Gamma \oplus_? \Xi$, *then* $\Phi^\star \vdash$ *dexp and* $\Xi \sim$ *dexp.*

(3) *If* $\vdash \Delta$ *wf and* $\Delta;\hat{\Xi}_{pkg} \vdash \overline{B} : \Xi \rightsquigarrow$ *dexp, then* $\cdot \vdash$ *dexp and* $\Xi \sim$ *dexp.*

(4) *If* $\vdash \Delta$ *wf and* $\Delta \vdash D : \forall\overline{\alpha}.\Xi \rightsquigarrow \lambda\overline{\alpha}.$*dexp, then* $\cdot \vdash$ *dexp and* $\Xi \sim$ *dexp.*

The statement acts as an extension of the corresponding Regularity statement. Whereas that statement concludes that the object classifying a well-typed term is itself well-formed, the Elaboration Soundness statement concludes that a well-typed term's elaboration is itself well-typed and related to its EL type via the soundness relation. (Indeed, the proofs of this statement rely on Regularity.) The translation function from EL module identities to IL module names, $(-)^\star$, plays a central role in stating the various parts of the Soundness Theorem and in establishing the relationship between EL and IL.

Each part of Elaboration Soundness is described and its proof is sketched in order in the rest of this section. Each proof makes heavy use of technical lemmas about the elaboration and about the soundness relation; see Appendix D for the complete listing. For example, we presume yet another axiom about core-level typing: if *defs* are well-typed in the EL then their translation, via $\mathsf{refs}^\star_{\nu_0}(-)$, is well-typed in the IL (Axiom A.132).

In particular, the proofs employ *Package-Level Consistency* (PLC, Property A.3)—the property that in a single EL module context (Φ), there are no two worlds that conflict with each other—in a number of different places. The metatheory of the IL required meticulous side conditions like *world preservation*, and the proofs that follow knock them away, ungracefully, with PLC and its corollaries (*e.g.*, Corollary A.139).

9.6.1 *Soundness of module elaboration*

Proof. By inversion of the elaboration derivation $\Gamma; \nu_0 \vdash M : \tau\,@\,\omega \rightsquigarrow$ *hsmod*. Rule (ElabMod) defines the elaborated module *hsmod* via the definitions presented in Figure 9.6.

- By Lemma A.129(4) we know the elaborated imports resolve to the translated environment *eenv** and world $\mathring{\omega}'$ that extends the translated one $\omega^\star$. But for this lemma we need to satisfy the following subgoals:
  - (a) $\mathsf{world}^+_\Phi(\mathcal{L}(\mathsf{imps}(\mathit{impdecls})))$ defined
  - (b) $\mathsf{world}^+_\Phi(\mathcal{L}(\mathsf{imps}(\mathit{impdecls}))) \oplus_? \omega$
- (a):
  - Immediate by a corollary to *Package-Level Consistency* used in the elaboration (Corollary A.139)—the first instance in the Soundness proofs where this sledgehammer is swung. The reason we need it here is because we're moving from the EL semantics of "use the worlds of my *direct* imports" to the IL semantics of "use the worlds of all my *transitive* imports," and the latter is a heftier requirement.
- (b):
  - By definition of $\mathsf{world}^+_-(-)$ and $\mathsf{depends}^+_-(-)$, the subgoal is rewritable to $\mathsf{world}_\Phi(N) \oplus \mathsf{world}_\Phi(\mathsf{depends}_\Phi(N)) \oplus_? \omega$, where $N = \mathcal{L}(\mathsf{imps}(\mathit{impdecls}))$.
  - By inversion on the core environment construction of the EL typing, we know that $\omega = \mathsf{world}_\Phi(N) \oplus \ldots$. Now it sufficies to show that $\mathsf{world}_\Phi(\mathsf{depends}_\Phi(N)) \oplus_? \omega$.
  - Let $\nu_i \in N$; then $\nu_i = \mathcal{L}(\mathsf{imps}(\mathit{impdecl}_i))$ for some i. Now it suffices to show that $\mathsf{world}_\Phi(\mathsf{depends}_\Phi(\nu_i)) \oplus_? \omega$.
  - Let $\nu_i' \in \mathsf{depends}_\Phi(\nu_i)$. Now it suffices to show that $\mathsf{world}_\Phi(\nu_i') \oplus_? \omega$.
  - By assumption, $\Phi \oplus_? \nu_0{:}\tau^+@\,\omega$. Then by *Package-Level Consistency* in the definition of $\oplus$ on Φ, we know that every world in Φ is mergeable with ω.
  - Then we know that the world of ν_i' is also mergeable with it, achieving the subgoal.
- Now Lemma A.129(4) is applied.

- By the axiom on core definition checking in the IL (Axiom A.132), we know the well-typing of the translated definitions but only in the translated world $\omega^\star$.
- By monotonicity of core definition checking (Axiom A.90), we moreover know the well-typing of those translated definitions also in the extended world $\mathring{\omega}'$.
- By Lemma A.134 we know the elaborated exports resolve to the translated *espcs**.
- By definition of $\mathsf{mkimpdecls}(-;-)$, $N^\star = \mathsf{imps}(\mathsf{mkimpdecls}(\Gamma; \mathit{impdecls}))$.
- Then by rule (IlTyMod) we have the result.

$\square$

9.6.2 *Soundness of package binding elaboration*

Proof. By case analysis on the elaboration derivation $\Delta;\Gamma;\hat{\Xi}_{\mathsf{pkg}} \vdash B : \Xi \rightsquigarrow \mathit{dexp}$. We need to show (1) $\Phi^\star \vdash \mathit{dexp}$ and (2) $\Xi \sim \mathit{dexp}$.

- Case (ElabPkgMod):
  - By Elaboration Soundness on module typing and rule (IlTyHsmod), we know $\Phi^\star; \nu_0^\star \vdash \mathit{hsmod} : \tau^\star \mathbin{@} \mathring{\omega}'$ and $\mathring{\omega}' \sqsupseteq \omega^\star$.
  - For (1), by rule (IlTyDexp) with annotated world $\omega^\star$, it suffices to show that $\Phi^\star \vdash \lfloor \mathit{dexp} \rfloor$ wf.
  - By *Weakening* on *tfexp* typing (Lemma A.93(2)), we know $\Phi^\star \oplus \lfloor \mathit{dexp} \rfloor; \nu_0^\star \vdash \ldots$. But first we must satisfy the following subgoal:
    - (a) $\mathsf{modworld}_{\Phi^\star \oplus \lfloor \mathit{dexp} \rfloor}(\mathit{hsmod}; \mathit{ftyp})$ defined
  - (a):
    - By *Package-Level Consistency* in the elaboration (Corollary A.139) and Property A.131, it suffices to show that $\mathsf{world}^+_{\Phi}(N)^\star \oplus \mathsf{world}^+_{\nu_0:\tau^+@\omega}(N)^\star \oplus_? \mathsf{extworld}(\nu_0^\star; \tau^\star)$, where $N = \mathcal{L}(\mathsf{imps}(\mathit{impdecls}))$.
    - Then by a similar approach as in the previous proof, of module soundness, it suffices to show that $\mathsf{world}^+_{(\nu_0:\tau^+@\omega)^\star}(N^\star) \oplus_? \mathsf{extworld}(\nu_0^\star; \tau^\star)$.
    - Since the LHS is either $\omega^\star$ (if $\nu_0 \in N$), which equals the RHS plus some imported worlds, or $\{\!\!\{ \cdot \}\!\!\}$ (if not), we have the subgoal.
  - By regularity of EL typing (Theorem A.1(3)), we know $\Phi \vdash \Xi$ wf.
  - By translating the well-formedness of $\Xi.\Phi$ (Property A.142), we know $\Phi^\star \vdash \Xi.\Phi^\star$ wf.
  - By definition of $\lfloor - \rfloor$ and this *dexp*, $\Xi.\Phi^\star = \lfloor \mathit{dexp} \rfloor$, which with the previous result gives us (1).
  - (2) is immediate.
- Case (ElabPkgSig):
  - For (1), we need to show $\Phi^\star \vdash \mathsf{mkstubs}(\Phi')$ wf.
  - By Regularity of S typing and then applying Cut on the two resulting Φ-wf judgments, we know $\Phi \vdash \Phi'$ wf.
  - By translation of Φ wf-ness (Property A.142), $\Phi^\star \vdash \Phi'^\star$ wf.
  - By Lemma A.112 since $\Phi'^\star = \lfloor \mathsf{mkstubs}(\Phi') \rfloor$, we have (1).
  - By the expansion of $\Phi'^\star$ and by Lemma A.150, we have (2).
- Case (ElabPkgAlias) is trivial.
- Case (ElabPkgInc): Since it involves both substitution and filtering, this is the trickiest case to prove.

- For (1), we need to show $\Phi^\star \vdash \mathsf{apply}(\theta^\star; \mathit{dexp}|_{\mathsf{dom}(\Xi''.\Phi)^\star})$ wf. The approach is to first apply Strengthening of *dexp* typing and then Invariance under Substitution.
- By Δ wf-ness, we know $\mathit{fenv} \vdash \mathit{dexp}$ and $\cdot \vdash \Xi$ wf and $\Xi \sim \mathit{dexp}$.
- By wf-ness preservation under thinning (Lemma A.9), we know $\cdot \vdash \Xi'$ wf and $\mathsf{depends}_{\Xi.\Phi}(N) \subseteq N$, where $N = \mathsf{dom}(\Xi'.\Phi)$.
- By injectivity of $(-)^\star$, we have $\mathsf{depends}_{\Xi.\Phi}(N)^\star \subseteq N^\star$.
- By Property A.153,

$$\mathsf{depends}_{\Xi.\Phi^\star}(N^\star) = \mathsf{depends}_{\Xi.\Phi}(N)^\star \subseteq N^\star.$$

- Since $\mathsf{depends}_{\mathit{fenv}}(F) \subseteq F$, where $F = N^\star$ and $\mathit{fenv} = \Xi.\Phi^\star$, then by Strengthening on *dexp* typing, we have $\cdot \vdash \mathit{dexp}|_F$.
- Moreover, by Lemma A.152, we have $\Xi.\Phi|_N \sim \mathit{dexp}|_F$.
- Next we need to apply Invariance under Substitution to the well-typing of $\mathit{dexp}|_F$. For that we have three subgoals:
  - (a) $\mathsf{apply}(\theta^\star; \mathit{dexp}|_F)$ defined
  - (b) $\mathsf{avoidaliases}(\theta^\star; \mathit{dexp}|_F)$
  - (c) $\forall f \mapsto \mathit{tfexp} \mathbin{@} \mathring{\omega} \in \mathit{dexp}|_F :\ \mathsf{modworld}_{\theta^\star\lfloor \mathit{dexp}|_F \rfloor}(\theta^\star f; \theta^\star \mathit{tfexp})$ defined
- (a):
- For (a), we already know $\mathsf{apply}(\theta; \Xi'')$ defined from (ElabPkgInc), which means

$$\mathsf{apply}(\theta; \Xi''.\Phi) \text{ defined.}$$

- Since $\Xi'' = \mathsf{rename}(r; \Xi')$, we moreover know

$$\mathsf{apply}(\theta; \Xi'.\Phi) \text{ defined.}$$

- By commutativity of substitution with translation (Lemma A.147), we know $\mathsf{apply}(\theta^\star; \Xi'.\Phi^\star)$ defined.
- Since $\Xi.\Phi|_N \sim \mathit{dexp}|_F$, by Lemma A.151, $\lfloor \mathit{dexp}|_F \rfloor = (\Xi.\Phi|_N)^\star = \Xi'.\Phi^\star$.
- Since $\Xi'.\Phi^\star = \lfloor \mathit{dexp}|_F \rfloor$, by Lemma A.147, $\mathsf{apply}(\theta^\star; \lfloor \mathit{dexp}|_F \rfloor)$ defined if and only if $\mathsf{apply}(\theta^\star; \mathit{dexp}|_F)$ defined.
- Finally, since $\mathsf{apply}(\theta^\star; \Xi'.\Phi^\star)$, we have (a).
- (b): Immediate by Lemma A.155, since we already know that $\Xi.\Phi|_N \sim \mathit{dexp}|_F$. That lemma expresses the fact that the $\mathsf{mkmod}(-)$ definition of Figure 9.6 produces `import` statements that always have import aliases that avoid $\theta^\star$, since $\mathit{ModIdents}^\star \cap \mathit{ModNames} = \emptyset$.
- (c): This is the *world preservation* condition described in §9.3; its proof burden should now be more clear.
- Suppose $f \mapsto \mathit{tfexp} \mathbin{@} \mathring{\omega} \in \mathit{dexp}|_F$. Then we must show that

$$\mathsf{modworld}_{\theta^\star\lfloor \mathit{dexp}|_F \rfloor}(\theta^\star f; \theta^\star \mathit{tfexp}) \text{ defined}$$

  Proceed by case analysis on the syntax of *tfexp*.
- Case $\mathit{tfexp} = (- : \mathit{ftyp})$:
  - By definition of $\mathsf{modworld}_{\ldots}(f; (- : \mathit{ftyp}))$, the goal is to show that

$$\mathsf{extworld}(\theta^\star f; \theta^\star \mathit{ftyp}) \text{ defined.}$$

- The side condition in the definition of $\mathsf{apply}(\mathring{\theta}; \mathit{fenv})$—a straightforward translation of the EL definition in Figure 7.4)—says that the substitution is only defined if $\mathsf{extworld}(\mathring{\theta}\mathsf{f}; \mathring{\theta}\mathit{ftyp})$ is defined for each file f in the *fenv*. Although it matches the intuition that a substitution on *fenv* cannot "break any worlds," the side condition exists entirely to support this proof.
- With that in mind, since $\mathsf{apply}(\theta^\star; \lfloor \mathit{dexp}|_\mathsf{F} \rfloor)$ defined, and since $\mathit{tfexp} \mathbin{@} \mathring{\omega} \in \mathit{dexp}|_\mathsf{F} \Rightarrow \mathit{tfexp} \mathbin{@} \mathring{\omega} \in \lfloor \mathit{dexp}|_\mathsf{F} \rfloor$, by the side condition we have $\mathsf{extworld}(\theta^\star \mathsf{f}; \theta^\star \mathsf{typ}(\mathit{tfexp}))$, which is the goal.

- Case $\mathit{tfexp} = (\mathit{hsmod} : \mathit{ftyp})$:
  - By definition of $\mathsf{modworld}_{\ldots}(\mathsf{f}; (\mathit{hsmod} : \mathit{ftyp}))$, we have three subgoals:
    - (i) $\mathring{\omega}'_\mathsf{ext} = \mathsf{extworld}(\theta^\star \mathsf{f}; \theta^\star \mathit{ftyp})$ defined, for some $\mathring{\omega}'_\mathsf{ext}$
    - (ii) $\mathring{\omega}'_\mathsf{imp} = \mathsf{world}^{+}_{\theta^\star \lfloor \mathit{dexp}|_\mathsf{F} \rfloor}(\mathsf{imps}(\theta^\star \mathit{hsmod}))$ defined, for some $\mathring{\omega}'_\mathsf{imp}$
    - (iii) $\mathring{\omega}'_\mathsf{ext} \oplus_? \mathring{\omega}'_\mathsf{imp}$
  - (i):
  - As above, due to the definition of substitution on *fenv*, we have $\mathsf{extworld}(\theta^\star \mathsf{f}; \theta^\star \mathit{ftyp})$ defined.
  - Since $\cdot \vdash \mathit{dexp}|_\mathsf{F}$ from above and $\mathsf{f} \cdots \in \mathit{dexp}|_\mathsf{F}$, then by (IlTyDexp), we have $\lfloor \mathit{dexp}|_\mathsf{F} \rfloor; \mathsf{f} \vdash (\mathit{hsmod} : \mathit{ftyp}) \mathbin{@} \mathring{\omega}$.
  - By (IlTyHsmod) we have $\mathring{\omega} \sqsupseteq \mathsf{extworld}(\mathsf{f}; \mathit{ftyp})$, and therefore we have $\mathsf{extworld}(\mathsf{f}; \mathit{ftyp})$ defined; call it $\mathring{\omega}_\mathsf{ext}$.
  - By Lemma A.121, we have

    $$\mathring{\omega}'_\mathsf{ext} = \mathsf{extworld}(\theta^\star \mathsf{f}; \theta^\star \mathit{ftyp}) = \theta^\star \mathsf{extworld}(\mathsf{f}; \mathit{ftyp}) = \theta^\star \mathring{\omega}_\mathsf{ext}$$

    and therefore the goal (i).
  - (ii):
  - By the well-typing of *hsmod* in $\lfloor \mathit{dexp}|_\mathsf{F} \rfloor$, $\mathsf{imps}(\mathit{hsmod}) \subseteq \mathsf{dom}(\lfloor \mathit{dexp}|_\mathsf{F} \rfloor)$.
  - We next need to show that the imports of $\theta^\star \mathit{hsmod}$ are contained in $\mathsf{dom}(\lfloor \mathit{dexp}|_\mathsf{F} \rfloor)$.

    $$\begin{aligned} \mathsf{imps}(\theta^\star \mathit{hsmod}) &= \theta^\star \mathsf{imps}(\mathit{hsmod}) \\ &\subseteq \theta^\star \mathsf{dom}(\lfloor \mathit{dexp}|_\mathsf{F} \rfloor) \\ &= \theta^\star \mathsf{dom}(\Xi'.\Phi^\star) \\ &= \mathsf{dom}(\theta^\star \Xi'.\Phi^\star) \;\; \text{(a substitution we know is defined)} \\ &= \mathsf{dom}(\theta \Xi'.\Phi^\star) \;\; \text{(by Lemma A.147)} \\ &= \mathsf{dom}(\Xi'.\Phi)^\star \;\; \text{(by injectivity of } (-)^\star\text{)} \end{aligned}$$

  - Since $\Xi.\Phi|_\mathsf{N} \sim \mathit{dexp}|_\mathsf{F}$ and $\mathit{hsmod} \in \mathit{dexp}|_\mathsf{F}$, we know that

    $$\mathit{hsmod} = \mathsf{mkmod}(\Gamma; \nu_0; \mathit{eenv}; \mathit{impdecls}; \mathit{defs}; \mathit{espcs}).$$

  - Then we additionally know that $\mathsf{imps}(\mathit{hsmod})$ is in the image of $(-)^\star$, and therefore $\theta^\star \mathsf{imps}(\mathit{hsmod})$ is too.
  - Let N_0 be the preimage of $\mathsf{imps}(\theta^\star \mathit{hsmod})$ under $(-)^\star$. Then $\mathsf{N_0}^\star = \mathsf{imps}(\theta^\star \mathit{hsmod})$ and $\mathsf{N}_0 \subseteq \mathsf{dom}(\Xi'.\Phi)$.
  - Then by *Package-Level Consistency* in the EL Property A.3, $\mathsf{world}^{+}_{\theta \Xi'.\Phi}(\mathsf{N}_0)$ is defined; let it be ω'.
  - By preservation of worlds under $(-)^\star$ (Property A.145), $\mathsf{world}^{+}_{\theta \Xi'.\Phi^\star}(\mathsf{N_0}^\star)$ is defined and equals $\omega'^\star$.

- By rewriting, $\mathsf{world}^{+}_{\lfloor dexp|_F \rfloor}(\mathsf{imps}(\theta^\star hsmod))$, which is the goal (ii).
- (iii):
- As above, via (ILTYHSMOD), we have $\mathring{\omega}' \sqsupseteq \mathring{\omega} \sqsupseteq \mathsf{extworld}(f;ftyp)$, where $\mathring{\omega}'$ is the actual world that *hsmod* is typed under.
- By preservation of world extension under substitutions (Property A.85), $\theta^\star\mathring{\omega} \sqsupseteq \theta^\star\mathsf{extworld}(f;ftyp)$.
- By rewriting using the definition of $\mathring{\omega}'_{\mathsf{ext}}$ as before, $\theta^\star\mathring{\omega} \sqsupseteq \mathring{\omega}'_{\mathsf{ext}}$.
- By definition of $\mathsf{world}^{+}_{fenv}(-)$, $\mathring{\omega} = \mathsf{world}^{+}_{\lfloor dexp|_F \rfloor}(f)$.
- By commutativity of substitution with contextual worlds (Property A.86), $\theta^\star\mathring{\omega} = \mathsf{world}^{+}_{\theta^\star\lfloor dexp|_F \rfloor}(\theta^\star f)$.
- By *Package-Level Consistency* in the EL Property A.3, every world in the same context $\Xi'.\Phi$ is mergeable, and therefore $\mathsf{world}^{+}_{\theta\Xi'.\Phi}(N_0) \oplus_? \mathsf{world}^{+}_{\theta\Xi'.\Phi}(\theta\nu)$, where ν is the inverse of f under $(-)^\star$.
- By commutativity of translation with contextual worlds (Property A.145) and by rewriting, $\mathring{\omega}'_{\mathsf{imp}} \oplus_? \theta^\star\mathring{\omega}$.
- Finally, since $\theta^\star\mathring{\omega} \sqsupseteq \mathring{\omega}'_{\mathsf{ext}}$, we have $\mathring{\omega}'_{\mathsf{imp}} \oplus_? \mathring{\omega}'_{\mathsf{ext}}$, goal (iii).

- By applying Invariance under Substitution on *dexp* typing, we have $\cdot \vdash \theta^\star dexp|_F$.
- At this point, we need to apply Weakening on *dexp* typing in order to get goal (1), $\Phi^\star \vdash \theta^\star dexp|_F$. For that we have two subgoals:
  - (a) $\Phi^\star \oplus_? \lfloor \theta^\star dexp|_F \rfloor$
  - (b) $\forall f \mapsto (hsmod : ftyp) @ \mathring{\omega} \in \theta^\star dexp|_F :\ \mathsf{modworld}_{\Phi^\star \oplus \lfloor \theta^\star dexp|_F \rfloor}(hsmod;ftyp)$ defined
- (a):
  - By assumption, $\Gamma \oplus_? \theta\Xi''$, which means $\Phi \oplus_? \theta(\Xi''.\Phi)$.
  - By distributivity of translation over merging (Property A.143), we know $\Phi^\star \oplus_? \theta(\Xi''.\Phi)^\star$, which by earlier result gives us goal (a).
- (b): Suppose $f \mapsto (hsmod : ftyp) @ \mathring{\omega} \in \theta^\star dexp|_F$.
  - We have three subgoals:
    - (i) $\mathring{\omega}'_{\mathsf{imp}} = \mathsf{world}^{+}_{\Phi^\star \oplus \lfloor \theta^\star dexp|_F \rfloor}(\mathsf{imps}(hsmod))$ defined
    - (ii) $\mathring{\omega}'_{\mathsf{ext}} = \mathsf{extworld}(f;ftyp)$ defined
    - (iii) $\mathring{\omega}'_{\mathsf{imp}} \oplus_? \mathring{\omega}'_{\mathsf{ext}}$
  - (i):
    - Rewriting this file environment, we have $(\Phi \oplus \theta\Xi''.\Phi)^\star$.
    - Since $hsmod \in \theta^\star dexp|_F$, $\mathsf{imps}(hsmod) \subseteq ModIdents^\star$.
    - Then by *Package-Level Consistency*, we have $\mathsf{world}^{+}_{\Phi \oplus \theta\Xi''.\Phi}(\mathsf{imps}(hsmod))$ defined, and then after translation, $\mathsf{world}^{+}_{(\Phi \oplus \theta\Xi''.\Phi)^\star}(\mathsf{imps}(hsmod))$.
    - By pushing the $(-)^\star$ back into the environment, we have the goal (i).
  - (ii) is immediate since $f \mapsto (hsmod : ftyp) @ \mathring{\omega}$ in well-typed $\theta^\star dexp|_F$ (in empty environment), by (ILTYHSMOD).
  - (iii):
    - Let N be the preimage of $\mathsf{imps}(hsmod)$ under $(-)^\star$ and let ν such that $\nu^\star = f$.
    - By *Package-Level Consistency*, since $\Phi \oplus_? \theta\Xi''.\Phi$, all worlds appearing in this context must be mergeable, *i.e.*, $\mathsf{world}^{+}_{\Phi \oplus \theta\Xi''.\Phi}(N) \oplus_? \mathsf{world}^{+}_{\Phi \oplus \theta\Xi''.\Phi}(\nu)$.

- By Property A.131, $\mathsf{world}^{+}_{\Phi^\star \oplus (\theta\Xi''.\Phi)^\star}(N^\star) \oplus_? \mathsf{world}^{+}_{\Phi^\star \oplus (\theta\Xi''.\Phi)^\star}(\nu^\star)$.
- By rewriting, $\mathsf{world}^{+}_{\Phi^\star \oplus \theta^\star \mathit{dexp}|_F}(\mathsf{imps}(\mathit{impdecls})) \oplus_? \mathsf{world}^{+}_{\Phi^\star \oplus \theta^\star \mathit{dexp}|_F}(f)$.
- By Property A.98 the RHS $\sqsupseteq \mathsf{world}^{+}_{\theta^\star \mathit{dexp}|_F}(f)$.
- Since $f \mapsto (\mathit{hsmod} : \mathit{ftyp}) @ \dot{\omega} \in \theta^\star \mathit{dexp}|_F$ the RHS is mergeable with $\mathsf{extworld}(f;\mathit{ftyp})$ since together they form $\dot{\omega}$ (by the core environment construction from the *hsmod* typing).
- Then taken together the last few steps mean that $\dot{\omega}'_{\mathsf{imp}} \oplus_? \dot{\omega}'_{\mathsf{ext}}$, which gives us goal (iii).

- Therefore we have subgoal (b) and can now apply Weakening to get goal (1).
- Now we must prove goal (2) $\theta\Xi'' \sim \theta^\star \mathit{dexp}|_F$.
- By applying preservation of soundness relation under substitution (Lemma A.154) to the earlier result $\Xi.\Phi|_N \sim \mathit{dexp}|_F$, we have goal (2).

□

9.6.3 *Soundness of sequence of package bindings elaboration*

Proof. By induction on the elaboration derivation $\Delta;\hat{\Xi}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \rightsquigarrow \mathit{dexp}$. We need to show (1) $\cdot \vdash \mathit{dexp}$ and (2) $\Xi \sim \mathit{dexp}$. The (ElabPkgNil) case is trivial. The proof for the (ElabPkgSeq) case proceeds.

- By inductive hypothesis, $\cdot \vdash \mathit{dexp}_1$ and $\Xi_1 \sim \mathit{dexp}_1$, and as a result, $\Xi_1.\Phi^\star = \lfloor \mathit{dexp}_1 \rfloor$.
- By Regularity on the typing of $\overline{B_1}$ (Theorem A.1), $\cdot \vdash \Xi_1$ wf.
- Since $\vdash \Delta$ wf, $\cdot \vdash \Xi_1$ wf, and $\Xi_1 \oplus_? \Xi_2$, by Elaboration Soundness on the typing of B_2, we have $\Xi_1.\Phi^\star \vdash \mathit{dexp}_2$ and $\Xi_2 \sim \mathit{dexp}_2$.
- We concentrate first on goal (1), which we need only to apply Cut on IL typing to get. That yields subgoals:
  - (a) $\cdot \vdash \mathit{dexp}_1$
  - (b) $\lfloor \mathit{dexp}_1 \rfloor \vdash \mathit{dexp}_2$
  - (c) $\mathit{dexp}_1 \oplus_? \mathit{dexp}_2$
  - (d) $\forall i \in [1,2] : \forall f \mapsto \mathit{tfexp} @ \dot{\omega} \in \mathit{dexp}_i : \mathsf{modworld}_{\lfloor \mathit{dexp}_1 \oplus \mathit{dexp}_2 \rfloor}(f;\mathit{tfexp})$ defined
- Subgoals (a), (b), and (c) are all immediate, leaving (d) as the meat of this proof.
- Suppose $i \in [1,2]$ and $f \mapsto \mathit{tfexp} @ \dot{\omega} \in \mathit{dexp}_i$.
- Case $i = 1$:
  - Case $\mathit{tfexp} = (- : \mathit{ftyp})$:
    - Then $\mathsf{modworld}_{\mathit{fenv}}(f;\mathit{tfexp}) \overset{\text{def}}{=} \mathsf{extworld}(f;\mathit{ftyp})$, for any *fenv*.
    - Since $\cdot \vdash \mathit{dexp}_1$ and $f \mapsto \mathit{tfexp} @ \dot{\omega} \in \mathit{dexp}_1$, by rule (IlTyStub) we have the existence of $\mathsf{extworld}(f;\mathit{ftyp})$ and therefore, with $\mathit{fenv} = \lfloor \mathit{dexp}_1 \oplus \mathit{dexp}_2 \rfloor$, the goal (d).
  - Case $\mathit{tfexp} = (\mathit{hsmod} : \mathit{ftyp})$:
    - Let $\Phi_1 = \Xi_1.\Phi$ and $\Phi_2 = \Xi_2.\Phi$. By earlier results we have $\Phi_1^\star = \lfloor \mathit{dexp}_1 \rfloor$ and $\Phi_2^\star = \lfloor \mathit{dexp}_2 \rfloor$.
    - Then the goal can be rewritten to $\mathsf{modworld}_{\Phi_1^\star \oplus \Phi_2^\star}(f;\mathit{tfexp})$ defined, which has three subgoals:
      - (i) $\mathsf{world}^{+}_{\Phi_1^\star \oplus \Phi_2^\star}(\mathsf{imps}(\mathit{hsmod}))$ defined; call it $\dot{\omega}_{\mathsf{imp}}$
      - (ii) $\mathsf{extworld}(f;\mathit{ftyp})$ defined; call it $\dot{\omega}_{\mathsf{ext}}$

- (iii) $\acute{\omega}_{\mathsf{imp}} \oplus_? \acute{\omega}_{\mathsf{ext}}$

- (i):
  - Since *hsmod* is well-typed in $\lfloor dexp_1 \rfloor$, we know $\mathsf{imps}(hsmod) \subseteq \mathsf{dom}(\lfloor dexp_1 \rfloor)$.
  - Then also $\mathsf{imps}(hsmod) \subseteq \mathsf{dom}(\lfloor dexp_1 \rfloor \oplus dexp_2)$, the latter of which can be rewritten to $\mathsf{dom}(\Phi_1^\star \oplus \Phi_2^\star)$.
  - Since $\mathsf{imps}(hsmod) \subseteq ModIdents^\star$ (because *hsmod* is from elaboration), by *Package-Level Consistency* we have such a $\acute{\omega}_{\mathsf{imp}}$.
- (ii) is immediate; by rule (ILTYHSMOD), $\acute{\omega} \sqsupseteq \mathsf{extworld}(f;ftyp) = \acute{\omega}_{\mathsf{ext}}$.
- (iii):
  - By the typing of $hsmod : ftyp \,@\, \acute{\omega}$ in $dexp_1$, we know that its imported worlds and local world merge: $\mathsf{world}^+_{\lfloor dexp_1 \rfloor}(\mathsf{imps}(hsmod)) \oplus_? \mathsf{extworld}(f;ftyp)$. That's almost the subgoal; we just need the environment on the LHS to be expanded to $\lfloor dexp_1 \oplus dexp_2 \rfloor$.
  - Let $F = \mathsf{imps}(hsmod)$. Because *hsmod* is the result of elaboration then $\mathsf{imps}(hsmod)$ are all contained in $ModIdent^\star$; let N be the preimage of F under $(-)^\star$ and, likewise, ν the preimage of f.
  - By *Package-Level Consistency* in the EL, $\mathsf{world}^+_{\Phi_1 \oplus \Phi_2}(N) \oplus_? \mathsf{world}^+_{\Phi_1 \oplus \Phi_2}(\nu)$.
  - Then by translating and distributing translation into $\mathsf{world}^+_-(-)$, $\mathsf{world}^+_{\Phi_1^\star \oplus \Phi_2^\star}(N^\star) \oplus_? \mathsf{world}^+_{\Phi_1^\star \oplus \Phi_2^\star}(\nu^\star)$.
  - By rewriting we then have $\acute{\omega}_{\mathsf{imp}} = \mathsf{world}^+_{\Phi_1^\star \oplus \Phi_2^\star}(\mathsf{imps}(hsmod)) \oplus_? \mathsf{world}^+_{\Phi_1^\star \oplus \Phi_2^\star}(f)$. Now it remains to show that $\acute{\omega}_{\mathsf{imp}}$ additionally merges with $\acute{\omega}_{\mathsf{ext}}$.
  - By the rules of *dexp* merging, $f \mapsto tfexp \,@\, \acute{\omega} \in dexp_1 \oplus dexp_2$ as well. Then $\acute{\omega} = \mathsf{world}_{\lfloor dexp_1 \rfloor}(f) = \mathsf{world}_{\lfloor dexp_1 \oplus dexp_2 \rfloor}(f)$.
  - By the typing of *hsmod* in $dexp_1$, $\acute{\omega} \sqsupseteq \acute{\omega}_{\mathsf{ext}} = \mathsf{extworld}(f;ftyp)$.
  - Taken together, $\acute{\omega}_{\mathsf{imp}} = \mathsf{world}^+_{\Phi_1^\star \oplus \Phi_2^\star}(\mathsf{imps}(hsmod)) \sqsupseteq \mathsf{world}_{\Phi_1^\star \oplus \Phi_2^\star}(\mathsf{imps}(hsmod)) = \acute{\omega} \oplus_? \acute{\omega}_{\mathsf{ext}}$, so that we have subgoal (iii).

- Case $i = 2$, *i.e.*, $f \mapsto tfexp \,@\, \acute{\omega} \in dexp_2$:
  - Since $\lfloor dexp_1 \rfloor \vdash dexp_2$, by soundness of reinterpreted module world (Lemma A.95), we have $\mathsf{modworld}_{\lfloor dexp_1 \rfloor \oplus \lfloor dexp_2 \rfloor}(f;tfexp)$ defined.
  - With simple rewriting, this is the goal (d).
- With subgoals (a) through (d) met, via Cut on *dexp* typing we have goal (1).
- Now we must prove (2) $\Xi_1 \oplus \Xi_2 \sim dexp_1 \oplus dexp_2$.
- Since we already proved $\Xi_1 \sim dexp_1$ and $\Xi_2 \sim dexp_2$, we simply apply the preservation of relation under merging (Lemma A.157) to get (2).

□

9.6.4 *Soundness of package definition elaboration*

The proof of this part of Elaboration Soundness is fairly straightforward given all the machinery built up for the earlier parts of the theorem. Notably, there's nothing interesting to say about the $\overline{\alpha}$ part of the parameterized package type and directory expression. The notation is, admittedly, a technical nuisance to the formalization.

Proof. By inversion of the elaboration derivation $\Delta \vdash D : \forall\overline{\alpha}.\Xi \rightsquigarrow \lambda\overline{\alpha}.dexp$, rule (ELABPKG).
- Let $N = \mathsf{dom}(\Xi'.\Phi)$ and $F = N^\star$.

- We need to show (1) $\cdot \vdash$ *dexp*$|_F$ and (2) $\Xi'.\Phi|_N \sim$ *dexp*$|_F$.
- By Elaboration Soundness on $\overline{B}$ elaboration, we have $\cdot \vdash$ *dexp* and $\Xi \sim$ *dexp*.
- By Regularity on $\overline{B}$ typing, we have $\cdot \vdash \Xi$ wf.
- Then by thinning preservation (Lemma A.9), we have $\cdot \vdash \Xi'$ wf and $\mathsf{depends}_{\Xi.\Phi}(N) \subseteq N$.
- We have (1) by Strengthening on *dexp* typing and (2) by Strengthening on soundness relation.

$\square$

IL Module Judgments

Typing of module source files

$$\boxed{\mathit{fenv} \vdash \mathit{hsmod} : \mathit{ftyp} \mathbin{@} \mathring{\omega}}$$

$$\frac{\begin{array}{c}\mathit{fenv};\, \mathsf{f}_0 \vdash \mathit{imp\mathring{d}ecls}; \mathit{d\mathring{e}fs} \rightsquigarrow \mathit{ee\mathring{n}v} \mathbin{@} \mathring{\omega} \qquad \mathit{ee\mathring{n}v} \vdash \mathit{exp\mathring{d}ecl} \rightsquigarrow \mathit{es\mathring{p}cs} \\ \mathit{fenv};\, \mathsf{f}_0;\, \mathit{ee\mathring{n}v};\, \mathring{\omega} \vdash \mathit{d\mathring{e}fs} : \mathit{ds\mathring{p}cs} \qquad \{\overline{\mathsf{f}}\} = \{\mathsf{imp}(\mathit{imp\mathring{d}ecl}) \mid \mathit{imp\mathring{d}ecl} \in \mathit{imp\mathring{d}ecls}\}\end{array}}{\mathit{fenv} \vdash (\texttt{module}\ \mathsf{f}_0\ \mathit{exp\mathring{d}ecl}\ \texttt{where}\ \mathit{imp\mathring{d}ecls}\texttt{;}\ \mathit{d\mathring{e}fs}) : \langle\!|\, \mathit{ds\mathring{p}cs} \,;\, \mathit{es\mathring{p}cs} \,;\, \overline{\mathsf{f}} \,|\!\rangle \mathbin{@} \mathring{\omega}}\ (\textsc{IlTyMod})$$

Environment construction

$$\boxed{\mathit{fenv};\, \mathsf{f}_0 \vdash \mathit{imp\mathring{d}ecls}; \mathit{d\mathring{e}fs} \rightsquigarrow \mathit{ee\mathring{n}v} \mathbin{@} \mathring{\omega}}$$

$$\frac{\begin{array}{c}\forall i \in [1..n] : \mathit{fenv} \vdash \mathit{imp\mathring{d}ecl}_i \rightsquigarrow \mathit{ee\mathring{n}v}_i \\ \begin{array}{lcll}\mathit{ee\mathring{n}v} & = & \bigoplus_{i \in [1..n]} \mathit{ee\mathring{n}v}_i & \oplus\ \ \mathsf{mklo\mathring{c}aleenv}(\mathsf{f}_0; \mathit{d\mathring{e}fs}) \\ \mathring{\omega} & = & \bigoplus_{i \in [1..n]} \mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{imp}(\mathit{imp\mathring{d}ecl}_i)) & \oplus\ \ \mathsf{mklo\mathring{c}alworld}(\mathsf{f}_0; \mathit{d\mathring{e}fs}; \mathit{ee\mathring{n}v})\end{array}\end{array}}{\mathit{fenv};\, \mathit{imp\mathring{d}ecl}_1, \ldots, \mathit{imp\mathring{d}ecl}_n \vdash \mathit{imp\mathring{d}ecls}; \mathit{d\mathring{e}fs} \rightsquigarrow \mathit{ee\mathring{n}v} \mathbin{@} \mathring{\omega}}\ (\textsc{IlCoreEnv})$$

Import resolution

$$\boxed{\mathit{fenv} \vdash \mathit{imp\mathring{d}ecl} \rightsquigarrow \mathit{ee\mathring{n}v}} \quad \boxed{\mathit{fenv};\, \mathsf{f} \vdash \mathit{imp\mathring{s}pec} \rightsquigarrow \mathit{es\mathring{p}cs}} \quad \boxed{\mathit{fenv};\, \mathsf{f} \vdash \mathit{im\mathring{p}ort} \rightsquigarrow \mathit{es\mathring{p}c}}$$

$$\frac{\begin{array}{c}\mathit{fenv};\, \mathsf{f} \vdash \mathit{imp\mathring{s}pec} \rightsquigarrow \mathit{es\mathring{p}cs} \\ \mathit{ee\mathring{n}v}_{\mathsf{base}} = \mathsf{mkeenv}(\mathit{es\mathring{p}cs}) \qquad \mathit{ee\mathring{n}v}_{\mathsf{qual}} = \mathsf{qualify}(\mathsf{f}';\, \mathit{ee\mathring{n}v}_{\mathsf{base}})\end{array}}{\mathit{fenv} \vdash (\texttt{import}\ \mathsf{f}\ \texttt{as}\ \mathsf{f}'\ \mathit{imp\mathring{s}pec}) \rightsquigarrow (\mathit{ee\mathring{n}v}_{\mathsf{base}} \oplus \mathit{ee\mathring{n}v}_{\mathsf{qual}})}\ (\textsc{IlImpDeclUnqual})$$

$$\frac{\begin{array}{c}\mathit{fenv};\, \mathsf{f} \vdash \mathit{imp\mathring{s}pec} \rightsquigarrow \mathit{es\mathring{p}cs} \\ \mathit{ee\mathring{n}v}_{\mathsf{base}} = \mathsf{mkeenv}(\mathit{es\mathring{p}cs}) \qquad \mathit{ee\mathring{n}v}_{\mathsf{qual}} = \mathsf{qualify}(\mathsf{f}';\, \mathit{ee\mathring{n}v}_{\mathsf{base}})\end{array}}{\mathit{fenv} \vdash (\texttt{import qualified}\ \mathsf{f}\ \texttt{as}\ \mathsf{f}'\ \mathit{imp\mathring{s}pec}) \rightsquigarrow \mathit{ee\mathring{n}v}_{\mathsf{qual}}}\ (\textsc{IlImpDeclQual})$$

$$\frac{\forall i \in [1..n] : \mathit{fenv};\, \mathsf{f} \vdash \mathit{im\mathring{p}ort}_i \rightsquigarrow \mathit{es\mathring{p}c}_i \qquad \mathit{es\mathring{p}cs} = \bigoplus_{i \in [1..n]} \{\mathit{es\mathring{p}c}_i\}}{\mathit{fenv};\, \mathsf{f} \vdash (\mathit{im\mathring{p}ort}_1, \ldots, \mathit{im\mathring{p}ort}_n) \rightsquigarrow \mathit{es\mathring{p}cs}}\ (\textsc{IlImpSpec})$$

$$\frac{\mathit{es\mathring{p}c} \in \mathit{fenv}(\mathsf{f}) \qquad \mathit{es\mathring{p}c} = [\mathsf{f}]\chi}{\mathit{fenv};\, \mathsf{f} \vdash \chi \rightsquigarrow \mathit{es\mathring{p}c}}\ (\textsc{IlImpSimple}) \qquad \frac{\mathit{es\mathring{p}c} \in \mathit{fenv}(\mathsf{f}) \qquad \mathit{es\mathring{p}c} \leqslant \mathit{es\mathring{p}c}' = [\mathsf{f}]\chi(\overline{\chi}')}{\mathit{fenv};\, \mathsf{f} \vdash \chi(\overline{\chi}') \rightsquigarrow \mathit{es\mathring{p}c}'}\ (\textsc{IlImpSub})$$

Export resolution

$$\boxed{\mathit{ee\mathring{n}v} \vdash \mathit{exp\mathring{d}ecl} \rightsquigarrow \mathit{es\mathring{p}cs}} \quad \boxed{\mathit{ee\mathring{n}v} \vdash \mathit{ex\mathring{p}ort} \rightsquigarrow \mathit{es\mathring{p}c}}$$

$$\frac{\begin{array}{c}\forall i \in [1..n] : \mathit{ee\mathring{n}v} \vdash \mathit{ex\mathring{p}ort}_i \rightsquigarrow \mathit{es\mathring{p}c}_i \\ \mathit{es\mathring{p}cs} = \bigoplus_{i \in [1..n]} \{\mathit{es\mathring{p}c}_i\} \qquad \mathsf{nooverlap}(\mathit{es\mathring{p}cs})\end{array}}{\mathit{ee\mathring{n}v} \vdash (\mathit{ex\mathring{p}ort}_1, \ldots, \mathit{ex\mathring{p}ort}_n) \rightsquigarrow \mathit{es\mathring{p}cs}}\ (\textsc{IlExpList})$$

$$\frac{\mathit{ee\mathring{n}v}(\mathit{e\mathring{r}ef}) = [\mathsf{f}]\chi : [\mathsf{f}]\chi}{\mathit{ee\mathring{n}v} \vdash \mathit{e\mathring{r}ef} \rightsquigarrow [\mathsf{f}]\chi}\ (\textsc{IlExpSimple}) \qquad \frac{\mathit{ee\mathring{n}v}(\mathit{e\mathring{r}ef}) = [\mathsf{f}]\chi : [\mathsf{f}]\chi(\overline{\chi}', \overline{\chi}'')}{\mathit{ee\mathring{n}v} \vdash \mathit{e\mathring{r}ef}\,(\overline{\chi}') \rightsquigarrow [\mathsf{f}]\chi(\overline{\chi}')}\ (\textsc{IlExpSub})$$

Figure 9.3: Definition of typing judgment and related definitions for IL module source files. The judgments constitute a restriction of their EL counterparts, with the notable distinction of using $\mathsf{world}^{\pm}(-)$ in (IlCoreEnv) instead of the EL's $\mathsf{world}_{-}(-)$.

Auxiliary Definitions on Worlds

Reinterpreted module world $\boxed{\mathsf{modworld}_{\mathit{fenv}}(\mathit{hsmod};\mathit{ftyp}) \;:_{\mathsf{par}}\; \mathring{\omega}}$ $\boxed{\mathsf{modworld}_{\mathit{fenv}}(\mathsf{f};\mathit{tfexp}) \;:_{\mathsf{par}}\; \mathring{\omega}}$

$$\mathsf{modworld}_{\mathit{fenv}}(\mathit{hsmod};\mathit{ftyp}) \overset{\mathsf{def}}{=} \mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{imps}(\mathit{hsmod})) \oplus \mathsf{extworld}(\mathsf{name}(\mathit{hsmod});\mathit{ftyp})$$

$$\begin{aligned} \mathsf{modworld}_{\mathit{fenv}}(\mathsf{f};\mathit{hsmod} : \mathit{ftyp}) &\overset{\mathsf{def}}{=} \mathsf{modworld}_{\mathit{fenv}}(\mathit{hsmod};\mathit{ftyp}) \\ \mathsf{modworld}_{\mathit{fenv}}(\mathsf{f};- : \mathit{ftyp}) &\overset{\mathsf{def}}{=} \mathsf{extworld}(\mathsf{f};\mathit{ftyp}) \end{aligned}$$

Worlds from a context $\boxed{\mathsf{world}_{\mathit{fenv}}(\mathsf{f}) \;:_{\mathsf{par}}\; \mathring{\omega}}$ $\boxed{\mathsf{world}_{\mathit{fenv}}(\mathsf{F}) \;:_{\mathsf{par}}\; \mathring{\omega}}$

$$\begin{aligned} \mathsf{world}_{\mathit{fenv}}(\mathsf{f}) &\overset{\mathsf{def}}{=} \mathring{\omega} \quad \text{if } \mathsf{f}{:}\mathit{ftyp}^{\mathsf{m}}@\,\mathring{\omega} \in \mathit{fenv} \\ \mathsf{world}_{\mathit{fenv}}(\mathsf{F}) &\overset{\mathsf{def}}{=} \bigoplus\nolimits_{\mathsf{f}\in\mathsf{F}\cap\mathsf{dom}(\mathit{fenv})} \mathsf{world}_{\mathit{fenv}}(\mathsf{f}) \end{aligned}$$

Transitive worlds from a context $\boxed{\mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{f}) \;:_{\mathsf{par}}\; \mathring{\omega}}$ $\boxed{\mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{F}) \;:_{\mathsf{par}}\; \mathring{\omega}}$

$$\begin{aligned} \mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{f}) &\overset{\mathsf{def}}{=} \mathsf{world}_{\mathit{fenv}}(\mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{f})) \\ \mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{F}) &\overset{\mathsf{def}}{=} \bigoplus\nolimits_{\mathsf{f}\in\mathsf{F}\cap\mathsf{dom}(\mathit{fenv})} \mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{f}) \end{aligned}$$

Figure 9.4: Definition of the *reinterpreted world* of a module source file in some file environment. Used in the statement of world preservation conditions in structural properties of the IL typing judgments.

Elaboration Semantics at Package Level

Elaboration of package bindings

$$\boxed{\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash B : \Xi \rightsquigarrow \mathit{dexp}}$$

$$\frac{\ell' \mapsto \nu \in \Gamma}{\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \ell = \ell' : \llparenthesis \cdot\ ;\ \ell \mapsto \nu \rrparenthesis \rightsquigarrow \{\,\}}\ (\textsc{ElabPkgAlias})$$

$$\frac{\ell \mapsto \nu_0 \in \hat{\hat{\Xi}}_{\mathsf{pkg}} \qquad \Gamma;\nu_0 \vdash M : \tau\,@\,\omega \rightsquigarrow \mathit{hsmod}}{\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \ell = [M] : \llparenthesis \nu_0{:}\tau^{+}@\,\omega\ ;\ \ell \mapsto \nu_0 \rrparenthesis \rightsquigarrow \{\nu_0^{\star} \mapsto \mathit{hsmod} : \tau^{\star}\,@\,\omega^{\star}\}}\ (\textsc{ElabPkgMod})$$

$$\frac{\begin{array}{c}\ell \mapsto \nu_0 \in \hat{\hat{\Xi}}_{\mathsf{pkg}} \qquad (\nu_0{:}\hat{\tau}_0^{m}@\,\hat{\omega}_0) \in \hat{\hat{\Xi}}_{\mathsf{pkg}} \\ \Gamma;(\hat{\tau}_0\,@\,\hat{\omega}_0) \vdash S : \sigma\,@\,\omega \mid \Phi_{\mathsf{sig}} \qquad \Phi' = \nu_0{:}\sigma^{-}@\,\omega \oplus \Phi_{\mathsf{sig}} \text{ defined}\end{array}}{\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \ell :: [S] : \llparenthesis \Phi';\ \ell \mapsto \nu_0 \rrparenthesis \rightsquigarrow \{\nu^{\star} \mapsto - : \tau^{\star}\,@\,\omega^{\star} \mid \nu{:}\tau^{-}@\,\omega \in \Phi'\}}\ (\textsc{ElabPkgSig})$$

$$\frac{\begin{array}{c}\overline{\alpha}\text{ fresh} \qquad (P = \lambda\overline{\alpha}.\mathit{dexp} : \forall\overline{\alpha}.\Xi) \in \Delta \qquad \vdash \Xi \xrightarrow{t} \Xi' \qquad \Xi'' = \mathsf{rename}(r;\Xi') \\ \overline{\alpha}' = \overline{\alpha} \cap \mathsf{dom}(\Xi''.\Phi) \qquad \vdash \hat{\hat{\Xi}}_{\mathsf{pkg}} \leqslant_{\overline{\alpha'}} \Xi'' \rightsquigarrow \theta\end{array}}{\Delta;\Gamma;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \textbf{include}\ P\ t\ r : \mathsf{apply}(\theta;\Xi'') \rightsquigarrow \mathsf{apply}(\theta^{\star};\mathit{dexp}|_{\mathsf{dom}(\Xi''.\Phi)^{\star}})}\ (\textsc{ElabPkgInc})$$

Elaboration of sequences of package bindings

$$\boxed{\Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \rightsquigarrow \mathit{dexp}}$$

$$\frac{}{\Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \cdot : \llparenthesis \cdot\,;\cdot \rrparenthesis \rightsquigarrow \{\,\}}\ (\textsc{ElabPkgNil})$$

$$\frac{\Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \overline{B_1} : \Xi_1 \rightsquigarrow \mathit{dexp}_1 \qquad \Delta;\Xi_1;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash B_2 : \Xi_2 \rightsquigarrow \mathit{dexp}_2 \qquad \Xi = \Xi_1 \oplus \Xi_2 \text{ defined}}{\Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \overline{B_1}, B_2 : \Xi \rightsquigarrow \mathit{dexp}_1 \oplus \mathit{dexp}_2}\ (\textsc{ElabPkgSeq})$$

Elaboration of package definitions

$$\boxed{\Delta \vdash D : \forall\overline{\alpha}.\Xi \rightsquigarrow \lambda\overline{\alpha}.\mathit{dexp}}$$

$$\frac{\Delta \Vdash \overline{B} \Rightarrow \hat{\hat{\Xi}}_{\mathsf{pkg}} \qquad \Delta;\hat{\hat{\Xi}}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \rightsquigarrow \mathit{dexp} \qquad \vdash \Xi \xrightarrow{t} \Xi' \qquad \overline{\alpha} = \mathsf{fv}(\Xi'.\Phi)}{\Delta \vdash \textbf{package}\ P\ t\ \textbf{where}\ \overline{B} : \forall\overline{\alpha}.\Xi' \rightsquigarrow \lambda\overline{\alpha}.\mathit{dexp}|_{\mathsf{dom}((\Xi'.\Phi))^{\star}}}\ (\textsc{ElabPkg})$$

Elaboration of package repositories

$$\boxed{\Delta \vdash R}$$

$$\frac{}{\Delta \vdash \cdot}\ (\textsc{ElabPkgRepoNil})$$

$$\frac{\Delta \vdash D : \forall\overline{\alpha}.\Xi \rightsquigarrow \lambda\overline{\alpha}.\mathit{dexp} \qquad \Delta, P = \lambda\overline{\alpha}.\mathit{dexp} : \forall\overline{\alpha}.\Xi \vdash \overline{D'}}{\Delta \vdash D, \overline{D'}}\ (\textsc{ElabPkgRepoSeq})$$

Figure 9.5: Definition of the elaboration semantics, defined as judgments that augment earlier typing judgments with elaborations. The shaded parts are the only changes from the corresponding typing judgments.

Elaboration Semantics at Module Level

Elaboration of modules

$$\boxed{\Gamma;\, \nu_0 \;\vdash\; M \;:\; \tau @ \omega \;\rightsquigarrow\; \mathit{hsmod}}$$

$$\frac{\begin{array}{c}\Gamma \;\vdash\; \mathit{impdecls}; (\nu_0, \ldots, \nu_0 \mid \mathit{defs}) \;\rightsquigarrow\; \mathit{eenv} \,@\, \omega \qquad \mathit{eenv} \;\Vdash\; \mathit{expdecl} \;\rightsquigarrow\; \mathit{espcs} \\ \Gamma.\Phi;\, \nu_0;\, \mathit{eenv};\, \omega \;\vdash\; \mathit{defs} \;:\; \mathit{dspcs} \qquad \mathsf{N} = \{\Gamma(\mathsf{imp}(\mathit{impdecl})) \mid \mathit{impdecl} \in \mathit{impdecls}\}\end{array}}{\begin{array}{c}\Gamma;\, \nu_0 \;\vdash\; (\mathit{impdecls}; \mathit{expdecl}; \mathit{defs}) \;:\; \langle\!|\, \mathit{dspcs}\,;\, \mathit{espcs}\,;\, \mathsf{N} \,|\!\rangle \,@\, \omega \\ \rightsquigarrow\; \mathsf{mkmod}(\Gamma; \nu_0; \mathit{eenv}; \mathit{impdecls}; \mathit{defs}; \mathit{espcs})\end{array}} \;(\textsc{ElabMod})$$

Module patching

$$\boxed{\mathsf{mkmod}(\Gamma; \nu_0; \mathit{eenv}; \mathit{impdecls}; \mathit{defs}; \mathit{espcs}) \;:_{\mathsf{par}}\; \mathit{hsmod}}$$

$$\mathsf{mkmod}(\Gamma; \nu_0; \mathit{eenv}; \mathit{impdecls}; \mathit{defs}; \mathit{espcs}) \;\stackrel{\mathsf{def}}{=}\; \left(\begin{array}{l}\texttt{module}\ \nu_0^\star\ \mathsf{mkexpdecl}(\mathit{espcs}; \mathit{eenv})\ \texttt{where} \\ \quad \mathsf{mkimpdecls}(\Gamma; \mathit{impdecls}) \\ \quad \mathsf{refs}^\star_{\nu_0}(\mathit{defs})\end{array}\right)$$

Module patching, imports

$$\boxed{\mathsf{mkimpdecls}(\Gamma; \mathit{impdecls}) \;:_{\mathsf{par}}\; \mathring{\mathit{impdecls}}} \quad \boxed{\mathsf{mkimpdecl}(\Gamma; \mathit{impdecl}) \;:_{\mathsf{par}}\; \mathring{\mathit{impdecl}}}$$

$$\boxed{\mathsf{mkimpspec}(\Gamma; \ell; \mathit{impspec}) \;:_{\mathsf{par}}\; \mathring{\mathit{impspec}}} \quad \boxed{\mathsf{mkentimp}(\Gamma; \ell; \mathit{import}) \;:_{\mathsf{par}}\; \mathring{\mathit{import}}}$$

$$\boxed{\mathsf{mkentimp}(\mathit{espc}) \;:\; \mathring{\mathit{import}}}$$

$$\mathsf{mkimpdecls}(\Gamma; \mathit{impdecls}) \;\stackrel{\mathsf{def}}{=}\; (\mathsf{mkimpdecl}(\Gamma; \mathit{impdecl})\texttt{;} \;\mid \mathit{impdecl} \in \mathit{impdecls})$$

$$\begin{array}{l}\mathsf{mkimpdecl}(\Gamma; \texttt{import}\ [\texttt{qualified}]\ \ell\ \mathit{impspec}) \;\stackrel{\mathsf{def}}{=} \\ \qquad \texttt{import}\ [\texttt{qualified}]\ (\Gamma.\mathcal{L})(\ell)^\star\ \texttt{as}\ \ell\ \mathsf{mkimpspec}(\Gamma; \ell; \mathit{impspec})\end{array}$$

$$\begin{array}{l}\mathsf{mkimpdecl}(\Gamma; \texttt{import}\ [\texttt{qualified}]\ \ell\ \texttt{as}\ \ell'\ \mathit{impspec}) \;\stackrel{\mathsf{def}}{=} \\ \qquad \texttt{import}\ [\texttt{qualified}]\ (\Gamma.\mathcal{L})(\ell)^\star\ \texttt{as}\ \ell'\ \mathsf{mkimpspec}(\Gamma; \ell; \mathit{impspec})\end{array}$$

$$\begin{array}{lcl}\mathsf{mkimpspec}(\Gamma; \ell; \cdot) & \stackrel{\mathsf{def}}{=} & \texttt{(}\mathsf{mkentimp}(\mathit{espc}) \mid \mathit{espc} \in \Gamma(\ell).\mathit{espcs}\texttt{)} \\ \mathsf{mkimpspec}(\Gamma; \ell; \texttt{(}\overline{\mathit{import}}\texttt{)}) & \stackrel{\mathsf{def}}{=} & \texttt{(}\mathsf{mkentimp}(\Gamma; \ell; \mathit{import})\texttt{)}\end{array}$$

$$\begin{array}{lcll}\mathsf{mkentimp}(\Gamma; \ell; \chi) & \stackrel{\mathsf{def}}{=} & \mathsf{mkentimp}(\mathit{espc}) & \text{if } \begin{cases}\mathit{espc} \in \Gamma(\ell) \\ \chi = \mathsf{name}(\mathit{espc})\end{cases} \\ \mathsf{mkentimp}(\Gamma; \ell; \chi\texttt{(..)}) & \stackrel{\mathsf{def}}{=} & \mathsf{mkentimp}(\mathit{espc}) & \text{if } \begin{cases}\mathit{espc} \in \Gamma(\ell) \\ \chi = \mathsf{name}(\mathit{espc}) \\ \mathsf{hasSubs}(\mathit{espc})\end{cases} \\ \mathsf{mkentimp}(\Gamma; \ell; \chi\texttt{(}\overline{\chi'}\texttt{)}) & \stackrel{\mathsf{def}}{=} & \chi\texttt{(}\overline{\chi'}\texttt{)} & \end{array}$$

$$\begin{array}{lcl}\mathsf{mkentimp}([\nu]\chi) & \stackrel{\mathsf{def}}{=} & \chi \\ \mathsf{mkentimp}([\nu]\chi(\overline{\chi'})) & \stackrel{\mathsf{def}}{=} & \chi\texttt{(}\overline{\chi'}\texttt{)}\end{array}$$

Figure 9.6: Definition of the elaboration semantics at the module level.

Elaboration Semantics at Module Level, Continued

Module patching, exports

$\boxed{\mathsf{mkexpdecl}(\mathit{espcs};\mathit{eenv}) \;:\; \mathring{\mathit{expdecl}}}$ $\boxed{\mathsf{mkexp}(\mathit{espc};\mathit{eenv}) \;:_{\mathsf{par}}\; \mathring{\mathit{export}}}$

$$\mathsf{mkexpdecl}(\mathit{espcs};\mathit{eenv}) \stackrel{\text{def}}{=} (\mathsf{mkexp}(\mathit{espc};\mathit{eenv}) \mid \mathit{espc} \in \mathit{espcs})$$

$$\begin{array}{lll} \mathsf{mkexp}([\nu]\chi;\mathit{eenv}) & \stackrel{\text{def}}{=} \mathsf{refs}^{\star}_{\nu_0}(\mathit{eref}) & \text{if} \;\; \exists\, \mathit{eref} :\; \mathit{eenv}(\mathit{eref}) = [\nu]\chi \\ \mathsf{mkexp}([\nu]\chi(\overline{\chi'});\mathit{eenv}) & \stackrel{\text{def}}{=} \mathsf{refs}^{\star}_{\nu_0}(\mathit{eref})(\overline{\chi'}) & \text{if} \;\; \exists\, \mathit{eref} :\; \mathit{eenv}(\mathit{eref}) = [\nu]\chi : [\nu]\chi(\overline{\chi'}) \end{array}$$

Entity patching

$\boxed{\mathsf{refs}^{\star}_{\nu}(\mathit{eenv}) \;:\; \mathring{\mathit{eenv}}}$ $\boxed{\mathsf{refs}^{\star}_{\nu}(\mathit{eref}) \;:\; \mathring{\mathit{eref}}}$ $\boxed{\mathsf{refs}^{\star}_{\nu}(\cdots) \;:\; \cdots}$

$$\mathsf{refs}^{\star}_{\nu}(\mathit{eenv}) \stackrel{\text{def}}{=} \{\mathsf{refs}^{\star}_{\nu}(\mathit{eref}) \mapsto \mathit{phnm}^{\star} \mid \mathit{eref} \mapsto \mathit{phnm} \in \mathit{eenv}\}\,;\; \mathsf{locals}(\mathit{eenv})^{\star}$$

$$\begin{array}{lll} \mathsf{refs}^{\star}_{\nu}(\chi) & \stackrel{\text{def}}{=} & \chi \\ \mathsf{refs}^{\star}_{\nu}(\ell.\chi) & \stackrel{\text{def}}{=} & \ell.\chi \\ \mathsf{refs}^{\star}_{\nu}(\mathtt{Local}.\chi) & \stackrel{\text{def}}{=} & (\nu^{\star}).\chi \end{array} \qquad \mathsf{refs}^{\star}_{\nu}(\cdots\mathit{eref}\cdots) \stackrel{\text{def}}{=} \cdots\mathsf{refs}^{\star}_{\nu}(\mathit{eref})\cdots$$

Figure 9.7: Definition of the elaboration semantics at the module level, continued.

Elaboration Soundness Relation

Relation of module context to directory expression

$\boxed{\Xi \sim \mathit{dexp}}$

$$\begin{array}{lll} \Xi \sim \mathit{dexp} & \stackrel{\text{def}}{\Leftrightarrow} & \Xi.\Phi \sim \mathit{dexp} \\ \Phi \sim \mathit{dexp} & \stackrel{\text{def}}{\Leftrightarrow} & \forall i \in [1..n] :\; (\nu_i{:}\tau_i{}^{m_i} @\, \omega_i) \sim (f_i \mapsto \mathit{tfexp}_i @\, \mathring{\omega}_i) \\ & \text{where} & \begin{cases} \Phi = \nu_1{:}\tau_1{}^{m_1} @\, \omega_1, \ldots, \nu_n{:}\tau_n{}^{m_n} @\, \omega_n \\ \mathit{dexp} = \{\, f_1 \mapsto \mathit{tfexp}_1 @\, \mathring{\omega}_1, \ldots, f_n \mapsto \mathit{tfexp}_n @\, \mathring{\omega}_n \,\} \end{cases} \end{array}$$

Relation of singleton module context to file

$\boxed{(\nu{:}\tau^{m} @\, \omega) \sim (f \mapsto \mathit{tfexp} @\, \mathring{\omega})}$

$$\begin{array}{lll} (\nu{:}\tau^{+} @\, \omega) \sim (\nu^{\star} \mapsto \mathit{hsmod} : \tau^{\star} @\, \omega^{\star}) & \stackrel{\text{def}}{\Leftrightarrow} & \exists\, \Gamma, \mathit{eenv}, \mathit{impdecls}, \mathit{defs} : \\ & & \quad \mathit{hsmod} = \mathsf{mkmod}(\Gamma;\nu;\mathit{eenv};\mathit{impdecls};\mathit{defs};\tau.\mathit{espcs}) \\ (\nu{:}\tau^{-} @\, \omega) \sim (\nu^{\star} \mapsto - : \tau^{\star} @\, \omega^{\star}) & \stackrel{\text{def}}{\Leftrightarrow} & \text{always} \end{array}$$

Figure 9.8: Definition of the soundness relation between EL module contexts and IL directory expressions.

Part III

EPILOGUE

10 RELATED WORK

In the introduction I tied together the constellation of concepts pertaining to *modularity* that would be necessary to understand the core contributions of Backpack. In this chapter I flesh out in more detail some of the comparisons to related work in module systems and mixins.

10.1 MIXIN MODULES

ML modules provide a very expressive and convenient language for programming with abstract data types. However, due to a variant of the double vision problem (Section 2.6), functors are fundamentally incompatible with recursive linking.[1] There have therefore been several attempts to synthesize aspects of ML modules and mixin modules in a single system, including Owens and Flatt's typed unit calculus[2] and Duggan's type system for recursive DLLs.[3] Arguably the most advanced system in this space is Rossberg and Dreyer's MixML,[4] which will be discussed in the next section.

Our focus has been on mixin-based strong modularity in the setting of a typed *functional* language. In the object-oriented community, mixins have already seen significant uptake. Both Scala[5] and J&[6], for instance, incorporate mixin-style composition into the very fabric of their designs. However, as we have explained, we are particularly interested in the question of how to retrofit existing languages with mixin-based strong modularity, and to our knowledge there is relatively little work on that.

In their FJig line of work,[7] Servetto and Zucca imbue abstract classes in Java with mixin behavior: various mixin operators like merge combine abstract classes to form new ones. Whereas Backpack retrofits Haskell modules with a whole new layer on top, the FJig languages redefine the existing abstract class layer of a Java-like language; they are implemented via "flattening" all the mixin operations into ordinary abstract class definitions. In comparison, the Backpack retrofitting is a somewhat more modular extension to Haskell in that we essentially do not touch the existing language of Haskell modules at all.

The SmartJavaMod/*component* systems of Ancona *et al.*[8] define a new level of mixin modules to encapsulate existing Java classes. Because they do not redefine the existing language constructs, these component systems are more closely related to Backpack than other mixin systems for OO languages.

A component contains *defined* classes and *deferred* classes—essentially just the difference between implementations and holes in Backpack. Deferred classes are specified as abstract class names with constraints on their inheritance and methods and fields. The "bind" construct mixin-merges two components by plugging in definitions from one component for deferred classes in the other; moreover, bind instantiates the components generatively, producing a unique copy of all the classes inside the merged result (and thus fresh abstract types). In contrast, Backpack supports an applicative semantics of instantiation.

The SmartJavaMod/component languages are implemented with a translation into "polymorphic bytecode,"[9] essentially an extension of JVM bytecode with markers and constraints

1 Rossberg and Dreyer (2013), "Mixin' Up the ML Module System".
2 Owens and Flatt (2006), "From Structures and Functors to Modules and Units".
3 Duggan (2002), "Type-safe linking with recursive DLLs and shared libraries".
4 Rossberg and Dreyer (2013), "Mixin' Up the ML Module System".
5 Odersky and Zenger (2005), "Scalable component abstractions".
6 Nystrom *et al.* (2006), "J&: Nested Intersection for Scalable Software Composition".
7 Corradi *et al.* (2011) and Lagorio *et al.* (2012).
8 Ancona *et al.* (2005b, 2006).
9 Ancona *et al.* (2005a), "Polymorphic bytecode: compositional compilation for Java-like languages".

for the deferred classes (*i.e.*, holes) in the component. For that reason, their internal language resembles Backpack's, although they present no formal definition of their elaboration into this language. Instead, they define a reduction semantics on components that, like the flattening in FJIG languages, produces fully instantiated class definitions that can be understood by the underlying Java-like language. In Backpack, following much work on ML module systems,[10] packages do not have a direct reduction semantics—rather, their meaning is given by a formal translation into a typed internal language, in our case based on Haskell.

As is the tradition in object-oriented languages, the aforementioned systems emphasize dynamic binding, virtual methods, overriding, etc., and do not consider the issue of the double vision problem. In contrast, Backpack supports only static binding, does not permit overriding, and invests great effort to avoid double vision.

10.2 MIXML

The most conceptually and technically relevant work to Backpack is Rossberg and Dreyer's MixML[11], a highly expressive foundational calculus of mixin modules for an ML-like core language. Backpack's design and semantics are inspired by those of MixML, but our design decisions, driven by our goal of retrofitting Haskell with *strong, large* modularity, have led to considerable simplifications.[12]

MixML supports first-class and higher-order *units* (*i.e.*, instantiable mixins), whereas Backpack's units—packages—only exist at the top-level. We find Backpack's units to be sufficient for practical programming, and restricting them to top-level streamlines the semantics of applicative identities. MixML also supports hierarchical mixin modules with "deep linking", but Backpack restricts packages to be flat namespaces of modules. Deep linking lets MixML express many different ML constructs (*e.g.*, n-ary signatures) using just the single form of linking. Since our focus is on practicality rather than expressiveness, we sacrifice features like first-class units and deep linking for simplicity of syntax and semantics, optimizing instead for common usage patterns. In particular, Backpack's **include** construct is syntactically more straightforward to program with than MixML's binary linking/binding construct, and fits better with the "feel" of the Haskell module language (*e.g.*, its **import** statements).

MixML lacks some features that Backpack provides for practical reasons. In particular, renaming and thinning allow programmers greater control over the automatic mixin linking behavior. Moreover, our thinning semantics also allows programmers to completely discard irrelevant holes from a package. Module projection in MixML similarly allows programmers to selectively use certain parts of modules while ignoring the rest, but there is no way to ignore undefined parts of those modules even when the selected parts do not depend on them, *i.e.*, like thinning in Backpack.

Concerning the double vision problem, MixML solves it through the use of a two-pass algorithm for typechecking linked modules: the first pass computes all information about type components in the modules, and the second pass performs full typechecking. In MixML, these two passes are defined using a single set of inference rules, with the first pass defined by conveniently ignoring certain premises. Backpack adopts the same two-pass idea in order to compute the physical module identities involved in a package before typechecking it. However, Backpack distinguishes the two passes—shaping and typing—using completely separate judgments and rules. Although this leads to a doubling of rules, the rules themselves are (we feel) much easier to understand. In particular, the account of linking given by the sequencing rules (ShPkgSeq) and (TyPkgSeq) feels simpler than MixML's formidable linking rule, though much of the complexity in Backpack's linking stems from an admittedly complicated unification specification (Figure 8.4). Moreover, Backpack stages the shaping pass over a whole package entirely before the typing pass, leading to a clearer conceptual split

10 Harper and Stone (2000), Rossberg and Dreyer (2013), and Rossberg *et al.* (2010).
11 Rossberg and Dreyer (2013), "Mixin' Up the ML Module System".
12 Backpack's complexity is largely due to its core and module levels, not the package level, which is the actual point of comparison with mixin modules.

between the two phases of package typechecking than in MixML, where the two passes are interleaved.

A key reason we can get by with a simpler semantics of linking is that we are deliberately less ambitious than MixML in a certain sense: unlike MixML, we do not aim to completely subsume the functionality of ML modules. MixML does, and this means that its semantics must deal with nested uses of *translucent sealing* (*i.e.*, the ability to define types that are "transparent" inside a module but "opaque" outside), a defining feature of ML modules which compounds the already-tricky double-vision problem. In contrast, Backpack does not attempt to support translucent sealing—and thus does not suffer the attendant complexities—for the simple reason that Haskell, our target of elaboration, cannot support it.

In MixML, instantiations happen in the full domain of *core-language types*. Unfortunately, the (potential) presence of higher-kinded types in the core language rules out the use of ordinary unification to determine instantiations, as higher-order unification is undecidable in general.[13] Therefore to implement instantiation without unification, MixML employs semantic metadata called "locators" and a "bidirectional type lookup" judgment that together complicate the linking rule. In Backpack, on the other hand, instantiations happen in the (closed) domain of module identities and are determined with ordinary unification on recursive first-order terms.

An additional simplification of Backpack over MixML arises in our use of context shapes in the typing judgment, based on MixML's "realizers". Realizers in MixML tell the typing judgment what the interpretation of an undefined type component should be. They are defined hierarchically and directly correspond to the structure of the concerning expression in the typing judgment. As a result, MixML's linking rule involves a complicated, *nondeterministic* splitting of the realizer in the conclusion in order to give realizers to the constituent expressions. Similarly, context shapes in Backpack tell the typing judgment what the identities for a signature binding should be. However, context shapes are flat and always describe the entire package rather than only the concerning binding expression; the TyPkgSeq rule consequently requires no such splitting and is deterministic.

Because MixML's typing judgment (and thus also its "static pass" judgment) are defined nondeterministically, implementing them with a deterministic algorithm is nontrivial. They define an additional semantic object called "templates" that, essentially, determinize the typing judgment by specifying, for a given expression, (1) the necessary realizer and abstract type variables to feed into the typing judgment and (2) the aforementioned realizer splitting in the linking rule. Templates are synthesized according to a separate (but similar) template computation judgment. We circumvent all this work in Backpack's type system by unifying template computation and the static typing pass into a single judgment—shaping. This unification is made possible by the unique way in which we stage the static pass entirely before the typing pass.

Finally, MixML is defined by elaboration into an internal language, LTG, which was designed specifically to capture all the necessary features of MixML. (LTG is an extension, with linear kinds, of an earlier internal language, similarly specialized for recursive ML module systems, called RTG[14].) LTG's tricky metatheory underscores MixML's status as a foundational calculus rather than a practical language design, in contrast to Backpack, whose internal language (IL) is a formalization of an existing implementation artifact, the GHC module system. A major benefit of our approach is that the semantics (via elaboration) of a Backpack package may be understood by Haskell programmers essentially in terms of a reshuffling of import and export lists in their Haskell modules. The elaboration in Figure 3.7 is a prime example of this.

13 Goldfarb (1981), "The Undecidability of the Second-Order Unification Problem".
14 Dreyer (2007b), "Recursive Type Generativity".

10.3 LOGICAL MODULE NAMES VS. PHYSICAL MODULE IDENTITIES

Module identities, which establish canonical *physical* names for modules (as distinct from program-level *logical* names), serve two important roles in Backpack's semantics: (1) they simplify and regularize the elaboration into Haskell modules (and its soundness proof), and (2) they are the principal component of our solution for how to support applicative mixin linking.

Concerning the first point: The distinction between logical and physical names is a central technical element enabling—and conceptually reinforcing—the elaboration into our Haskell-based IL. In particular, a key invariant of elaboration is that the physical part of a package's EL type gives a precise description of the IL modules that it elaborates to; the logical part of its type is only relevant for namespace management during Backpack-level typechecking.

Concerning the second point: The idea of distinguishing between logical and physical names is not new. A number of prior formalisms for ML-style modules—including the *Definition of Standard ML* itself—rely on a similar distinction.[15] The key advantage of this approach (as opposed to more direct, syntactic type systems for modules[16]) is that physical identities greatly simplify the treatment of type equality in the presence of aliasing: no matter their logical names, two types are equal iff they have the same physical identity. This eliminates the need for fancier mechanisms for handling type sharing, like translucent sums or singleton kinds (see Rossberg *et al.*[17] for further discussion). Moreover, for recursive and mixin module extensions of ML,[18] the logical/physical distinction has enabled clean solutions to the double vision problem, as discussed above. (There is some more recent work by Im *et al.* on solving double vision "syntactically"—*i.e.*, using only logical names—but it does not account for separate typechecking of mutually recursive modules in general.[19])

What distinguishes Backpack from these prior systems is its support for *both* separate typechecking of recursive modules *and* an applicative semantics of instantiation, as appropriate for a pure language like Haskell. To handle the combination, we needed to enrich the language of module identities with both (equi-)recursive μ-binders and constructor applications, and employ (standard) unification and equivalence-checking algorithms that work for these recursive identities.[20] To see why, consider the example from Section 2.6, in which the modules A and B in package ab-rec-sep have the recursive identities ν_A and ν_B defined on the subsequent page. If one were to define another package ab-rec-sep2 in the same way, the identities of A and B would be exactly the same. In contrast, were we to code up this example in MixML, each distinct package defined like ab-rec-sep would produce modules with "fresh" (distinct) identities, as one would expect given MixML's *generative* semantics of instantiation. Nevertheless, we observe that recursive identities do not complicate the semantics much, a testament to the scalability of the logical/physical approach.

10.4 MODULE IDENTITIES AND SHARING

Whereas Backpack largely concerns itself with the organization of *module identities*, conventional type system for ML modules largely concern themselves with the organization of *types*—abstract types in particular, whose creation exhibits generative and/or applicative semantics. However, some other works in the literature of module systems have employed a mechanism somewhat like Backpack's module identities.

MacQueen and Tofte developed a term representation of the semantics of functors, which they called *stamps*.[21] Using stamps they solved the "technical challenge in defining a semantics of higher-order functors [arising] from the way static identity information is propagated

15 Milner *et al.* (1997), Rossberg *et al.* (2010), and Russo (1998).
16 Harper and Stone (2000) and Leroy (1995).
17 Rossberg *et al.* (2010), "F-ing Modules".
18 Dreyer (2007a) and Rossberg and Dreyer (2013).
19 Im *et al.* (2011), "A Syntactic Type System for Recursive Modules".
20 Gauthier and Pottier (2004) and Huet (1976).
21 MacQueen and Tofte (1994), "A semantics for higher-order functors".

in Standard ML." Their usage of stamps was inspired by the earlier *Definition of Standard ML*,[22] which called the same notion *names* (but which didn't cover higher-order functors, an innovation of MacQueen and Tofte's system).

In the original SML system, the module system included *structure sharing*, which allowed signatures to not only express equivalence of *types* but also equivalence of *entire modules*, like in the following:

```
signature NUM = sig
  structure Eq : EQ
  structure Ord : ORD
  type t = Eq.t
  val plus : t * t -> t

  (* Ord's Eq component must be identical to Eq *)
  sharing Eq = Ord.Eq
end
```

In Russo's assessment,[23] "sharing of structures is a stronger property than mere sharing of types: it provides a static guarantee of the identity of values." SML used a form of stamps (*names*) to give meaning to such sharing constraints by requiring that the stamp of the `Eq` component be *identical to* that of the `Ord.Eq` component. But years later the revised *Definition of Standard ML* of 1997 dropped structure sharing altogether since the feature introduced technical complexity to the semantics and didn't seem, to the authors, to provide much practical usage beyond what type sharing provided.[24] (Further explanation of the presence and then absence of structure sharing in Standard ML can be found in (Rossberg *et al.*, 2014, p. 585–586).)

Leroy compared stamps vs. syntactic names by defining two different module systems, one with each of these two semantics, that supported translucent signatures (a.k.a. manifest types) and generative functors (but not higher-order functors).[25] In his TypModL system, stamps represented identities of core-level names rather than of modules, so it supported sharing of *types* but not of *structures*. In his TypModL′ system, syntactic names for core types derived from *paths*, —sequences of structure identifiers projecting out a core type, *e.g.*, `X.Y.t`—provided the basis for determing type equivalence: two types were identical if they could be expressed with the same syntactic path. This approach relied heavily on syntactic names and on "self-ification" module typing rules whereby the type of a structure name would be imbued with the path representing its own name; this formed the basis for syntactic, path-based semantics to module systems. Leroy showed that with additional syntactic rewriting the TypModL and TypModL′ systems were equivalent, even going so far as to sketch the following loose relationship to connect the ideas of stamps and names:

$$\text{type generativity and sharing} \;=\; \text{path equivalence} + \text{A-normalization} + \text{S-normalization}$$

Russo's thesis provided the first comprehensive treatment of a type system for ML modules by modeling generativity and applicativity of abstract types through existential and universal quantification in *System F*. The transitive influence of that work on Backpack's formalization is tremendous. Although Russo speculated that his system's mechanism for tracking abstract types could be readily extended to track module identity, his work did not support structure sharing.[26]

The *F-ing modules* system of Rossberg, Russo, and Dreyer subsumed stamps—and therefore module identity—into its comprehensive organization of abstract types, bringing the ideas of

22 Milner *et al.* (1990), "The Definition of Standard ML".
23 Russo (1998), "Types for Modules".
24 Milner *et al.* (1997), "The Definition of Standard ML (Revised)," §G.3.
25 Leroy (1996), "A syntactic theory of type generativity and sharing".
26 Russo (1998), "Types for Modules," p. 347.

Russo's thesis into fruition.[27] Rather than stamps they model structure sharing using *phantom types* that are generated for each *value* component; these phantom types for value components, taken together with the types for type components, approximate a module identity. Rossberg continued this same approach in his 1ML system.[28] 1ML unites the core and module levels that are generally stratified and kept semantically distinct, as in the *F-ing* system, Backpack, and countless other systems.

Backpack's development and usage of module identities is unique. To my knowledge, no account of module identity or stamps has supported both applicativity and recursion. Furthermore, a major benefit in tracking module identities, rather than individual core-level types like MixML, lies in the very straightforward elaboration into the underlying module language that this enables.

10.5 LINKSETS AND SEPARATE COMPILATION

In order to formally study linking and separate compilation of program fragments, Cardelli introduced a foundational calculus called *linksets*[29]. In this framework a program fragment represents (the typing derivation of) a module value together with interfaces for its internal dependencies on other values, while a linkset represents a collection of these fragments, each designated with a unique name, together with an interface for their collective external dependencies. Separate compilation then describes the translation from (typing derivations of) modules into these linksets. Through a reduction relation and algorithm, linking is implemented on linksets by substituting a named fragment's value for each reference to it in the other fragments until no more such substitutions are possible.

Cardelli's framework gives the impression of mixins and thus the two systems share some cosmetic similarity. Linksets resemble the directory expressions of Backpack's IL: each file of a directory explicitly provides its type, and the interfaces of internal dependencies are provided elsewhere in the directory. Like linksets, directories may be checked for individual well-typedness of files with respect to both internal and external dependencies. Unlike linksets, directories have no reduction or merging semantics: linking in the Backpack IL is realized instead through extra-linguistic means during elaboration, *i.e.*, by appealing to the $dexp \oplus dexp$ meta operation.

Despite its apparent similarity to Backpack, the linkset framework does not support recursive modules (or values or types) or user-defined abstract data types—two prominent features that drive the complexity of state-of-the-art module systems. As an example of how supporting these features steers Backpack away from the linkset framework, consider abstract data types: the lack of reduction semantics for the Backpack IL stems from the need to faithfully preserve the meaning of abstract data types (with the module identity mechanism) defined in the EL package.

10.6 SEPARATE COMPILATION FOR ML

Setting aside the lack of support for recursive linking, ML functors are not by themselves really a practical mechanism for strong modularity due to the proliferation of "sharing" constraints that are known to arise when programming in a "fully functorized" style[30] (*i.e.*, in which modules are parameterized explicitly, via the functor mechanism, over all their dependencies). Consequently, a number of systems have been proposed for building a better strong-modular framework on top of the existing ML module system.

Before discussing these systems in more detail, let us observe two important ways in which they all differ from Backpack. First, unlike Backpack, the separate compilation systems for ML build improved strong, *large* modularity support on top of the already-powerful ML

27 Rossberg *et al.* (2014), "F-ing modules".
28 Rossberg (2015), "1ML - Core and modules united (F-ing first-class modules)".
29 Cardelli (1997), "Program fragments, linking, and modularization".
30 Harper and Pierce (2005), "Design Considerations for ML-Style Module Systems".

module system, which offers instantiation, reuse, and strong (albeit *smaller*) modularity via functors. In contrast, Backpack is built on top of Haskell, which lacks those features, and thus the expressiveness boost it offers over the underlying language is in some sense more significant. Second, we realize this boost not through functors but through mixins. As a result, Backpack supports recursive linking and it avoids the need for any separate notion of sharing constraints, appealing instead to the *implict, by-name linking* of packages' abstract dependencies rather than to *explicit sharing expressions* in addition to those dependencies.

Building on Cardelli's linkset foundation, Swasey *et al.* designed a typed language of program fragments, SMLSC, that organizes lists of top-level SML definitions (*e.g.*, modules) into what they call *units*.[31] Linking happens automatically by name when unit definitions are considered in the same linkset. In particular, when multiple units in a linkset have "interface imports" on some common name, those dependencies unify automatically without extra annotations. SMLSC units therefore eliminate the need for sharing constraints on dependencies—as mixin modules do—but they do not permit recursive linking. Finally, as they are intended to be units of compilation rather than units of reuse, SMLSC units may not be instantiated in a single program with multiple different implementations of their imports, unlike Backpack.

In a different vein, targeting "open" modular programming, the Acute language of Sewell *et al.*[32] and the Alice ML language of Rossberg *et al.*[33] support not only separate compilation, but dynamic linking, marshalling/pickling, and (in the case of Acute) versioning of components, all of which are beyond the scope of Backpack. While Acute repurposes modules (with new primitive operations) as a mechanism for compilation units and linking, Alice ML defines "components" by reduction to a simpler construct of "packages" (modules as first-class core values). Linking in Acute consists of (non-recursive) chains of module definitions and imports, whereas Alice ML employs a more flexible and dynamic "component manager" approach based on Java class-loading (rather than linksets). Neither Acute nor Alice ML supports recursive modules.

As part of the OCaml module system,[34] the `ocamlc` compilation tool performs separate compilation on files that contain module components. The tool treats the file system rather like a mixin: each component (*i.e.*, a file) can be defined as an implementation (*i.e.*, a `.ml` file) or a hole (*i.e.*, a `.mli` file), and components can be recursively linked. Like SMLSC but unlike Backpack, though, these "mixins" cannot be instantiated and reused: a separately-compiled file cannot be linked with multiple implementations of its dependencies. In essence, `ocamlc` implements something similar to the target IL of Backpack's elaboration, albeit for OCaml (obviously) and extended with full separate compilation rather than just separate typechecking. It does not, however, provide a *language* for building and linking components, as Backpack does.

10.7 modular type classes

Dreyer *et al.*'s *modular type classes*[35] (MTC) sought to integrate type classes into the ML module system as "a particular mode of use" of modules and functors. Their key insight was to separate the definition of a type class instance and its *canonicalization*, or usage, in a particular scope. The MTC system requires that every "scope" of implicit instance resolution be consistent (which they call "coherence"); there cannot be two distinct ways to resolve the same desired class constraint. This requirement corresponds to the world consistency property of Backpack, that every module inhabit a consistent world.

In one particular way, Backpack goes further in ensuring consistency. Backpack's *world preservation* conditions (*e.g.*, in the definition of substitution on module contexts, or in the metatheory of the IL) guarantee that consistency is preserved by modular linking. The MTC system, however, does not—or at least it seems not to—guarantee that consistency is pre-

31 Swasey *et al.* (2006), "A separate compilation extension to Standard ML".
32 Sewell *et al.* (2007), "Acute: High-level programming language design for distributed computation".
33 Rossberg (2006), "The missing link – Dynamic components for ML".
34 Leroy *et al.* (2017), "The OCaml System release 4.06: Documentation and user's manual".
35 Dreyer *et al.* (2007), "Modular Type Classes".

served by modular linking, *i.e.*, functor application. That's because of the ability to define and use instances on abstract types, as in the following:

```
functor F (
    A : sig type t; ... end,
    B : sig type t; ... end) = struct
  structure EqAt = ... A.t ...
  structure EqBt = ... B.t ...
  using EqAt, EqBt in
    ... usage of EqAt ... usage of EqBt ...
end
F(IntModule, IntModule)
```

When this functor is statically analyzed, the MTC system accepts it by elaborating each usage of `EqAt` or `EqBt` into calls on the respective modules. But an application of the functor such that the two abstract types are actually the same will result in the functor body, within the `using` declaration, witnessing the equivalence of the two purportedly distinct instances.

It's not clear whether this behavior is actually problematic in the context of MTC. In this system, the elaboration of the functor body would explicitly fix the particular instance chosen; the body would not need to be later *re-evaluated* to choose a now-ambiguous instance. In the context of Backpack, this would be a more serious problem, so this linking would be statically ruled out. See the `uncertain` package from §4.5.3 for example.

10.8 BACKPACK'17

As I hinted in the intro, Backpack'17, presented in Yang's recent thesis,[36] is a continuation of the Backpack research project—of the original publication in particular, which he refers to as "Backpack'14." The reader will surely wonder how Backpack, as I've presented it in this thesis, differs from that of Yang's thesis.

Essentially, Backpack'17 is a *practical* implementation of the original Backpack presentation without type classes, *i.e.*, the system described in Chapters 2 and 3. Indeed, Yang has implemented Backpack'17 in the GHC Haskell infrastructure: in the GHC compiler (version 8.2) and in the `cabal-install` package manager (version 2.0).

But Backpack'17 is also a *re-design* of Backpack's semantics, at times in collaboration with myself and my Backpack co-authors, guided by an even more practical mission than my own: to retrofit not just a model of Haskell but *the entire actually-existing Haskell infrastructure*, *i.e.*, not just the actual compiler but the actual package management system. Through this effort Yang identified a key flaw in Backpack (both the original Backpack'14 presentation and the one presented in this thesis):

> *[D]espite its emphasis on being a practical design, Backpack'14 could not be implemented in a real world compiler like GHC. Why not? The semantics of Backpack'14, especially its package-level semantics, were closely entwined with the semantics of Haskell itself, violating the traditional abstraction barrier between the compiler and package manager. Without tightly coupling GHC (the Haskell compiler) and Cabal (the Haskell package manager), there was no way to directly implement Backpack'14.*[37]

Recognizing that abstraction barrier between compiler and package manager, and assigning to the former the responsibility of mixin linking and to the latter the responsibility of typechecking, is perhaps the key innovation behind Backpack'17.

36 Yang (2017), "Backpack: Toward Practical Mix-in Linking in Haskell".
37 Ibid., p.3.

10.8.1 *Language and features*

Backpack'17 considers the full Haskell/Cabal specification as the notion of packages. Inside that specification are *components*, which are named bundles of Haskell code—either libraries or executables—provided by the package. Resolved components are the analogue to packages in Backpack: they contain module declarations, signature declarations, and dependencies (*i.e.*, **include** bindings, with thinning and renaming). These are then *mixin linked* into *mixed* components, objects which resemble Backpack packages annotated with their shapes—in particular, inclusion bindings are annotated with their linking substitutions.

Backpack'17 incorporates a number of Haskell features missing in Backpack:

- the full language of types, kinds, and *roles* in modern Haskell;
- type synonyms and the attendant notion of (core-level) type equality;
- the ability for type synonyms to "implement" abstract type declarations in signatures, a feature of GHC Haskell with boot files in place of signatures; and
- type families and their instances.

Additionally, Backpack'17 does not require that concretely defined core entities have the same syntactic name as the abstractly declared entities they implement.

The primary Backpack feature missing from Backpack'17 is full recursive linking, *i.e.*, support for mutually recursive module bindings, although Yang sketches an approach based on Backpack's recursive module identities and on GHC Haskell's "boot files" approach.[38]

More minor deviations include syntactic restrictions for the sake of simplicity—and to only cover the kinds of components that generalize what people write today. As in Haskell, modules and signatures name themselves (with logical module names) and there are no alias bindings at the package level. Moreover, there is no merging among the modules and signatures defined within a single component. This means certain kinds of contrived examples in Backpack are inexpressible in Backpack'17.

Finally, thinning in Backpack'17 seems not to allow for thinning out unused holes,[39] *i.e.*, the ability to require some subset of a component's holes when the remaining holes are not transitively imported by any of the included modules. The lack of tracking all upstream modules in module types (except the orphan worlds; see below) provides further evidence for that omission.

10.8.2 *Identity*

Backpack'17 represents identity not principally at the module level but at the component level, called "unit identities." (In Backpack this would be like representing package identities.) This representation of identity allows (component) shapes to be synthesized without peering into the modules themselves, thus allowing shaping, *i.e.*, the determination of *physical* modular structure as part of mixin linking, to remain within the domain of the package manager and not the compiler. Backpack, on the other hand, represents identity at the module level, a technical design decision motivated in part by the elaboration semantics. In Backpack, identity at the module level elucidated the elaboration, whereas in Backpack'17 identity at the component level elucidates the package manager vs. compiler distinction.

As a result, Backpack'17 defines module identity as a unit identity plus a logical module name, *e.g.*, $\mathtt{q}[\mathtt{C} = \mathtt{p}[] : \mathtt{B}]$, or, for a hole, as a specially denoted logical module name, *e.g.*, $\langle\mathtt{A}\rangle$. This schema resembles Backpack's notion of physical names of core entities. Indeed, Backpack'17's "original names" are pairings of module identities with syntactic core entity

38 Yang (2017), "Backpack: Toward Practical Mix-in Linking in Haskell," Appendix B.

39 The definitions of rnthin and of mixin linking on dependencies (p. 36) does not impose any conditions on the designated requirements. Moreover, the typing rules for mixed components do not actually require that a unit identifier $\mathsf{P} = \mathsf{p}[\overline{m = M}]$ includes all of p's holes among the designated $\overline{m}$.

names. But there's an interesting second case for core-level original names: so-called "name holes" {`A.x`}. The distinction between name holes and module holes is very much like the distinction between βs and αs in Backpack: name holes denote the core bindings declared in a signature. Rather than being substituted by module identities of modules that define a matching core entity with the same syntactic name, however, they may be substituted by any arbitrary original name of a core-level entity.

A seemingly strange aspect of unit identities is that, like Backpack's module identities, they're parameterized by the identities of their *module holes*. In other words, unit identities constitute a coarser granularity of identity, but they're parameterized with the same granularity of identity. To see why module-level parameterization is necessary, in both systems, consider the following example:

```
component p where          component q where
  signature A where ...      module A1 where ...
  module B where             module A2 where ...
    import A
    ...
```

One might instantiate component `p` in two distinct ways: `p` with `q requires (A as A1)`, and `p with q requires (A as A2)`. If unit identity were parameterized at the unit level, then both instantiations would have some identity like `p[q]`. In order to observe the distinction—a distinction that will exist in the evaluation of `B`—we need the unit identities to reflect the different modules *within* a single component, *e.g.*, `p[A=q[]:A1]` vs. `p[A=q[]:A2]`.

10.8.3 *Substitution*

Substitution of module identities for holes is a key part of Backpack'17 as in Backpack, but the former is lazy whereas the latter is eager. In Backpack, a linking substitution is applied to an included package's type, θΞ, which then gets merged into the module context for subsequent modules to import. In Backpack'17, on the other hand, the linking substitution is more like an annotation on an included component; in modules that import those included modules, the substitution is applied lazily to the module type of the imported (substituted) module.

This decision guides Yang's implementation, as the type-checker of a module need only modify the interface of the looked-up module; what it never needs to do is apply a substitution to an entire component type in order to produce a new one to be processed.

10.8.4 *Type classes*

Backpack defines a world semantics of type classes in which the entire set of type class instances known to any module are reified as semantic objects. Backpack'17 defines a different semantics—GHC's orphan semantics described in Chapter 4, whereby orphanhood of instances is known to the type system. As a result, the module types (*i.e.*, "interfaces") of Backpack'17 contain the set of all upstream modules that define orphans.

Like in GHC Haskell, the orphan vs. non-orphan distinction is critical to the semantics and representation of type class instances in Backpack'17. The manifestation of that distinction is perhaps most evident in signature matching—called "module subtyping"—defined in the rule (SubMod).[40] The way that *non-orphan* instances are treated with respect to signature matching in Backpack'17 precisely corresponds to the way that *all* instances are treated with respect to signature matching in Backpack (§4.5.1).

In Backpack'17, in order for an implementing module to match a signature, two conditions must be met:

1. The instances known to the implementation must be able to "solve"—in the core-level instance resolution sense—each instance declaration in the signature. Translated to

40 Yang (2017), "Backpack: Toward Practical Mix-in Linking in Haskell," Fig. 6.5, p. 51.

Backpack's world semantics, this first condition would be like requiring that an implementation's world extend merely the smaller world of the signature's directly declared instances.

2. The set of orphan modules (*i.e.*, module identities that define orphan instances) known to the implementation must be *a superset* of those known to the signature. (In Backpack'17, orphan modules in the transitive closure of imports of a module are recorded in that module's type.)

Condition (2) is like the world extension requirement in my world semantics. But since my semantics doesn't make the distinction of orphan instances, thereby optimizing for the common case of *non*-orphan instances, it has to generalize from (2).

Finally, because of the lazy application of linking substitutions, Backpack'17 does *not* prohibit conflicting type class instances that result from linking.[41] In my formalization of Backpack, however, prohibiting such invalid linking was a core concern for the world semantics; recall the *world preservation* side conditions of the IL. For example, the uncertain package in Backpack from §4.5.3 defines conflicting instances if it's linked in such a way that the types implementing X_1.T and X_2.T are equal.

10.8.5 *Formalization*

Backpack'17's formalization is considerably more limited than that of Backpack. Notably, due in large part to the design decision to abstract mixin linking from the compiler, Backpack'17 does not define a semantics for the Haskell module level. In Backpack, that portion of the formalization drove perhaps the majority of the complexities detailed in this thesis.

Backpack'17 assumes some judgments from the Haskell core language, as does Backpack, albeit without clearly stated axioms to accompany them.[42] Noteworthy additions to these assumed judgments are the core-level type equality judgment and the instance resolution judgments. The former enables Backpack'17 to support type synonyms, which Backpack lacks. The latter gives meaning to the orphan semantics for type classes. Backpack, on the other hand, has no such judgment, even among its assumed judgments, as type class instance resolution is left up to the assumed judgment of typing of core-level bindings.

41 Yang (2017), "Backpack: Toward Practical Mix-in Linking in Haskell," p.83.
42 Ibid., §6.2.1.

11 CONCLUSION AND FUTURE WORK

11.1 CONCLUSION

The research project that led to Backpack started with the question, shared between Simon Peyton Jones and Derek Dreyer, "what if Haskell had mixin modules *à la* MixML?" Originally that question sought mixins at Haskell's *module level*. Over the subsequent years the practical challenges of "retrofitting Haskell with interfaces" pushed the focus one level up, to the *package level*. The accumulation of questions and solutions, experiments in a new kind of modularity for an existing language, became Backpack.

Over the course of my work developing the Backpack research project, I have

- brought the rich research tradition of type systems for ML modules into the world of Haskell and the less robustly researched world of package management systems;
- staked out an entirely new kind of modular programming in practice;
- formalized the Haskell module system in all its complexity and developed novel metatheory for that system, for example, to give a precise technical justification for a folklore assessment of type classes;
- applied my modeling of type classes to verify that the vast majority of widely used Haskell packages in the wild obey the key world consistency property; and
- stated and proved a key soundness theorem that both guides a potential implementation strategy and validates the design.

The conclusions one should draw from my work on Backpack can be categorized into those regarding Backpack's *design* (Part I) and those regarding its *formalization* (Part II).

I see the key *design* conclusions of Backpack as twofold. First, since *packages* have become an ubiquitous part of modular programming practice, they deserve the same attention that *modules* tend to receive—primarily, with the research tradition and concepts of *types*. Second, mixin modules offer a fertile ground for imbuing packages with the status of expression language with a type system, more so than ML-style functors do.

Moreover, thanks to the work of Edward Z. Yang in continuing and engineering Backpack, the Haskell community is already beginning to operationalize these design conclusions with new packages in Hackage.

As for the conclusions from Backpack's *formalization*, the primary one is that it acts as motivation and proof of concept for the design: by stating and proving soundness and attending to all the complexities of that process, the concrete design of Backpack—as a stratified system across core, module, and package levels—was fleshed out. That's true in particular for the addition of type classes to Backpack, a challenging project that required substantive changes to the formalization.

While my formalization of Haskell modules is certainly a technical contribution of my work, its exact takeaways for the research community around Haskell aren't so certain. I have presented a type system for Haskell modules that incorporates their possibly recursive nature, and in so doing, I have established some conventional metatheory around the core typing judgments. Some of that metatheory is predicated on axioms about the behavior of Haskell's core typechecking; that axiomatization also offers some insights into how one should formalize Haskell.

A particularly notable conclusion to draw from the formalization is the inherent complexity posed by (my semantics for) Haskell's type classes. It's a common refrain in the programming languages research community that type classes are "antimodular." By actually stating and proving concrete properties about Haskell's module system, I can now point to particular awkward side conditions, for example, as justification for that intuition.

In the end, however, my research does not present any clear conclusions about the *design* of type classes, *i.e.*, what a more modular alternative might look like, as I have only undertaken the challenge of formalizing (the primary features of) type classes as they exist in Haskell.

11.2 FUTURE WORK

My work on Backpack has formalized the bulk of Haskell's features that concern modularity. However, a few features have remained unincorporated into Backpack. On top of additional Haskell features, there are a couple aspects of the formalization that deserve consideration. These are discussed below.

11.2.1 *Filling gaps in Backpack*

TYPE SYNONYMS AND `newtype` One could straightforwardly extend Backpack with both type synonyms and Haskell's `newtype` mechanism for defining abstract data types. Both would be separate entities along with datatypes and values, with accompanying defined entity specs (*dspc*); because they are core entities, they would be imported, exported, and recorded in module types just like datatypes and values. However, for compatibility with GHC, neither would be declarable "abstractly" in signatures (*i.e.*, by omitting the "right-hand sides"), unlike regular data types.

Type synonyms are different, though, as they violate two assumptions baked into the `data` types of the Backpack core language. First, the latter generate fresh abstract types, whereas the former are synonyms for existing ones. And second, the representation of core-level types in Backpack are simply as the *physical names* of those types, including the *syntactic* name, like `T`. Type synonyms necessitate a representation of types that can treat two different syntactic type constructors as synonymous.

One way to accomplish this would be to simply expand type synonyms as part of Backpack elaboration, ensuring that they never appeared in our semantic objects (the "F-ing modules" approach of Rossberg *et al.*[1] works similarly). Since the synonyms themselves would never appear in any semantic types (*typ*), they would not complicate type equality. In contrast, `newtypes` would not be automatically equated with their defining types; in semantic types (*typ*) they would look and behave essentially like regular data types.

Another way to handle type synonyms might be to bake syntactic type equalities into the type system of the core level. Such is the approach of Im *et al.*[2] Alternatively, Yang's Backpack implementation supports type synonyms by assuming a (core-level) type equivalence judgment that is defined in part by looking up type synonym entities in module types.[3]

TYPE FAMILIES As *open* definitions of type-level functions that can be extended modularly, type families introduce a whole new category of problems for the type system. A concrete example can be found in the misabstraction problem presented in §4.2. There, the problem resulted in the downstream module observing "the wrong" values at runtime. But if the same example were rewritten with type *family* instances—with a type-level function `F I = Bool` on the left side and `F I = Int` on the right—then the problem results in *segmentation faults*, *i.e.*, breaking type safety. That's why the GHC Haskell implementation enforces *global uniqueness* for type family instances.

Backpack's worlds describe static knowledge about type class instances which is propagated implicitly to downstream modules. But there's no reason to limit that static knowledge to type *class* instances; type *family* instances could also fit nicely into the worlds framework.

1 Rossberg *et al.* (2010), "F-ing Modules".
2 Im *et al.* (2011), "A Syntactic Type System for Recursive Modules".
3 Yang (2017), "Backpack: Toward Practical Mix-in Linking in Haskell," §6.2.1.

And since Backpack already imposes package-level consistency, which was argued to be a modularization of global uniqueness (§4.3), the uniqueness requirement necessitated by type family instances would already be enforced. What's left then is to support the resulting core-level type computation and equality that type families introduce, perhaps using similar mechanisms as needed to support type synonyms.

MECHANIZATION OF THE FORMALIZATION The Backpack formalization has a profound shortcoming: it was not mechanized in a proof system like Coq. As a result, the details of the proofs of the myriad lemmas presented in the Appendix are mostly confined to my own paper notebooks. Although I took great care to develop the entire formalization with constructive proofs, I did not undertake the massive effort to mechanize them.

If I can impart one piece of wisdom for future PhD students in the field, it would be to develop your systems in Coq. Pen and paper is more expedient but the days, weeks, months, and years of extending and refactoring your system will surely become far easier when a machine can help you re-check all the details.

FURTHER DEVELOPMENT OF SHAPING My definitions of shaping judgments and "shapey" objects are validated not by proofs and metatheory but by their pseudo-formal resemblance to the corresponding definitions of typing judgments and "typey" objects. That's because my efforts developing the formalization were driven entirely by the Elaboration Soundness proof, which had no need for anything interesting about shaping.

I did not state, let alone prove, any of the structural properties of the shaping judgments and objects as I did with the typing judgments and objects. Moreover, I merely conjecture that shaping can be straightforwardly implemented as if the judgments designate deterministic algorithms. Indeed, my definition of shaping judgments are inspired not just by the "static" typing judgments of Dreyer's systems but also the "templates" of his and Rossberg's MixML.[4] But I do not argue that conjecture in formal terms.

11.2.2 *Practical implementation of world semantics*

In this dissertation I aimed to establish a more modular foundation for type classes: *world semantics* (§4.3). This semantics has the benefit of conceptual simplicity on paper, but it poses an interesting challenge for practical implementations: what should be the representation of a world? While I don't have a complete answer to this question, I can nonetheless describe the problem with the *canonical representation* of worlds demonstrated so far, along with an approach for a potential solution.

BLOAT IN THE CANONICAL REPRESENTATION As described in §4.3, and as defined formally in §4.4, worlds are canonically represented as mappings from class constraints to defining module names. Such objects would grow quite large in practice. For example, our prototype implementation (§4.3.3) revealed that the world of the automatically-imported Prelude module (as defined in GHC 7.8.4) contains 1,375 instances, defined among 64 other modules.[5]

When one world extends another, the former is an object at least as large as the latter; *i.e.*, worlds, as objects, will grow larger as the chain of module imports grows larger. Since I intend worlds to be embedded into module *types* (as in the extension to Backpack), this accumulative effect would introduce bloat in the file system, as Haskell implementations would surely represent module types on disk in order to implement modular typechecking. For example, GHC uses "binary interface files" (*i.e.*, `.hi` files) to represent module types. When typechecking any particular module, it must read and write interface files on disk. Such I/O operations necessitated the orphan instances distinction in the first place.

4 Rossberg and Dreyer (2013), "Mixin' Up the ML Module System," §9.2.
5 The analysis of GHC 7.8.4's standard Prelude module in particular: `https://gitlab.mpi-sws.org/backpack/class-struggle/blob/v3/data/worlds/prelude-readme.md`

On top of the accumulative effect of worlds, redundancy causes additional bloat. If multiple modules inhabit the same (large) world, each of their module types would, naively, contain the full contents of that world. This redundancy would cause bloat in the file system for the same reasons.

Therefore, in the worst possible case, where each module defines at most c instances and in which each new module imports all previous modules, the size of the world of the nth module is the sum of the sizes of all previous $n-1$ worlds (the imported instances) plus c (the locally defined instances). And, worse still, if each new module's world were represented distinctly from the worlds of all previous ones, then that number is added to the combined size of all $n-1$ worlds. The result is that the total size of all worlds together grows, at worst, exponentially in the number of modules.[6] Clearly, then, it's not sensible for an implementation to represent the types of modules in the naive way derived from the syntax and semantic objects of the Backpack formalization.

Indeed, the actually-existing implementation of (a variant of) Backpack, Yang's Backpack'17, does not adhere to the world semantics *per se* and therefore does not suffer any sort of exponential growth in the representation of type classes within a package. At the heart of that approach is GHC Haskell's existing representation of type class instances by considering *orphan instances* as distinct from the others. I discussed that approach in §10.8.

CHECKING MUTUAL CONSISTENCY When synthesizing the world inhabited by a module, an implementation must check for definedness of the $\oplus$ operation in three ways: among locally defined instances, between the world of those instances and the worlds of imports, and among the worlds of imports.

As discussed in §4.2, GHC only checks the first two merges; the omission of the third check is what allows global uniqueness to be broken. GHC omits this check as the result of a tradeoff: the damage done by not enforcing global uniqueness is deemed less severe than the expense of checking for conflicting instances among imported modules. In contrast to GHC's decision, I have argued that the former is severe enough to warrant the latter.

11.2.3 *Broader scope for Backpack and modularity*

Finally, by expanding the terrain of types into packages, Backpack introduces a number of potential trajectories for further research, which I sketch below.

MODULES VS. PACKAGES VS. TYPE CLASSES As I argued in the introduction, with my distinction between *small* and *large* modularity, both modules and packages have their uses to modular programming. But do the different *uses* necessitate different *constructs*?

Backpack's elaboration shows that packages are "compiled away" into just plain modules. But that's an explanation of their semantics, not an assertion that, to the programmer of the external language (EL), only modules and not packages are available to structure her programs. Modules and packages are, sadly, two different mechanisms with two very different semantics.

Recent work by Rossberg on 1ML has investigated how one system of modularity could express both the core language (of values and types) and the module language (of structures and signatures and data abstraction). One then wonders if something similar might unify modules and packages.

I suspect the answer is no. Because of the different use cases of small and large modularity, and in particular the different ways in which instantiation is conveniently expressed—functions and application in small modularity; mixins and linking in large modularity—they will necessitate different constructs with different semantics.

As for type classes, sadly, Backpack has little to say about evolving the design.

6 Let $T(n)$ be the combined size of all worlds among n modules. Then $T(n) \leqslant T(n-1) + T(n-1) + c$, where c is an arbitrary upper bound on the number of local instances any module can define. That yields the upper bound $T(n) \leqslant c2^{n-1} + n - 1$.

Over the course of my research I toyed with different ways to bring type classes into the original Backpack system.[7] One such way was to introduce syntax at the module level that explicitly "used" particular sets of type class instances within the scope of a module, *à la* the modular type classes of Dreyer *et al.*[8] Another way involved module-level expressions for reconciling conflicting instances within the body of a module that otherwise would observe an inconsistent world. All of these ideas ran up against the brick wall of my then-existing Backpack formalization and my research goal of keeping Backpack grounded in actually-existing Haskell.

The Package-Level Consistency (PLC) property (§4.4.5) that Backpack imposes is a very strong property, perhaps too heavyweight for many use cases of type classes. In the case of type classes that model algebraic properties of a program or a constellation of user-defined types, the total assurance of non-conflicting instances is perhaps necessary. At the design level, relying on PLC is how Backpack rules out *misabstraction* (§4.2). And at the formalization level, relying on PLC is also how I managed, over the time-bounded course of this work, to augment Backpack's semantics to handle the kinds of interaction between type classes and modular abstraction presented in §4.5. Indeed, PLC was crucially instrumentalized in the proof of Elaboration Soundness. As I wrote in §9.6, "[the soundness proofs] employ *Package-Level Consistency* [...] in a number of different places. The metatheory of the IL required meticulous side conditions like *world preservation*, and the proofs that follow knock them away, ungracefully, with PLC and its corollaries."

Sometimes, though, instead of encoding invariants about abstract data, type classes are just useful as a means of implicit programming, *i.e.*, *ad hoc* polymorphism and overloading. In these latter cases, one might want to weaken the consistency property, which might then necessitate a distinction in the world semantics between world facts that must be consistent and those that might not be.

In general, Backpack leaves open the possibility, if not the need, for further research on reconciling module systems and type classes.

PACKAGE VERSIONING AND DEPENDENCY RESOLUTION Lastly, while the support for versioning in Cabal does not obviate interfaces and mixins, neither do interfaces and mixins obviate versioning. An important direction for future work is to investigate how best to integrate versioning into Backpack. Versioning necessitates the concept of a named lineage of packages, and therefore some kind of relation beyond structural matching based on types—or at least those types need to be aware of names and lineage themselves.

If versioning were a first-class citizen in the package level and its type system, what might modular type matching and mixin linking look like? One direction might be the constraining of types by package lineages. For example, a way to express "give me any matching implementation of containers-sig-1.2 from the containers-impl lineage." But that would also require that the types of holes—at the *module* level—be associated with the identities and lineages of the *packages* that define them. This would be a substantial departure from Backpack's semantics.

Moreover, the key idea of *dependency resolution* in package management systems could be explored. Backpack sets up packages as fully explicit expressions assigned to unique names, but in practice, packages often involve a combination of *explicitly*-requested dependencies and *implicitly*-satisfied ones. The capability to express—and characterize with a type system—both explicit ("link this module for this hole") and implicit ("find me any module for this hole") mixin linking would be an interesting avenue for future work.

7 Kilpatrick *et al.* (2014), "Backpack: Retrofitting Haskell with Interfaces".
8 Dreyer *et al.* (2007), "Modular Type Classes".

Part IV

APPENDIX

A AUXILIARY DEFINITIONS

A.1 SEMANTIC OBJECTS

A.1.1 *Module Identity*

Each (non-variable) module identity uniquely identifies a module and its dependencies. Like a closure in the λ-calculus, stamps contain the code and, recursively, the closures (stamps) of its free variables (imports, resp.).

Identity Variables	α, β	$\in$	*IdentVars*
Identity Constructors	$\mathcal{K}$	$\in$	*IdentCtors*
Identities	ν	$::=$	$\alpha \mid \mu\alpha.\mathcal{K}\,\overline{\nu}$
Identity Substitutions	ϕ, θ	$::=$	$\{\overline{\alpha := \nu}\}$

• Identity variables α correspond to identities of signatures without corresponding implementations, and to the provenances of code entities specified in signatures.

• Each application $\mu\alpha.\mathcal{K}\,\overline{\nu}$ represents the closure of a particular module $\mathcal{K}$ with the identities of all its imported modules $\overline{\nu}$. We assume a bijection between identity constructors $\mathcal{K}$ and module source code M. The μ-binder is used for recursive modules that (transitively) import themselves. When α does not appear in $\overline{\nu}$ we leave off the binder:

$$\text{(Identities)}\quad \mathcal{K}\,\overline{\nu} \;\overset{\mathsf{def}}{=}\; \mu\alpha.\mathcal{K}\,\overline{\nu} \quad \text{for some } \alpha \notin \mathsf{fv}(\overline{\nu})$$

• Because module identities are recursive we cannot simply compare them for *syntactic* equality. Instead we need a coinductive interpretation of equivalence in which μ-binders are rolled or unrolled as needed:

$$\frac{}{\alpha \equiv_\mu \alpha} \qquad \frac{\{\alpha_1 := \mu\alpha_1.\mathcal{K}\,\overline{\nu}_1\}\,\overline{\nu}_1 \equiv_\mu \{\alpha_2 := \mu\alpha_2.\mathcal{K}\,\overline{\nu}_2\}\,\overline{\nu}_2}{\mu\alpha_1.\mathcal{K}\,\overline{\nu}_1 \equiv_\mu \mu\alpha_2.\mathcal{K}\,\overline{\nu}_2}$$

This definition directly follows that of the recursive types in (Gauthier and Pottier, 2004), albeit without their notion of "atoms" or ∀ quantifiers.

• This coinductive definition of equivalence doesn't lend itself naturally to an algorithm, so we appeal to unification of first-order recursive terms to decide equivalence. We write this unification as

$$\mathsf{unify}(\nu_1 \doteq \nu_2) = \theta$$

This statement means that, when free variables in ν_1 and ν_2 are substituted according to θ, the resulting identities are equivalent, *i.e.*, $\theta\nu_1 \equiv_\mu \theta\nu_2$. Unification on first-order recursive terms is an old problem in the literature of compilers.[1] The algorithm for implementing this unification is quite efficient: it requires $O(n\alpha(n))$ time, which is "almost linear" in the number of nodes in the graph representation of the terms. (This is more or less the module dependency graphs represented by the module identities.)

1 Knight (1989), "Unification: A Multidisciplinary Survey," §3.

- Due to a well-known correspondence of recursive types to DFAs and DFA minimization,[2] there exists a normalization function

$$\mathsf{norm}(-) : \mathit{ModIdents} \to \mathit{ModIdents}$$

on module identities, which is used for the elaboration. Normalization has the property that $\nu_1 \equiv_\mu \nu_2$ if and only if $\mathsf{norm}(\nu_1) = \mathsf{norm}(\nu_2)$, *i.e.*, the normalized forms are syntactically equivalent (modulo α-conversion).

- Throughout this thesis, particularly in definitions related to the *typing* pass, equality on module identities is written like ordinary syntactic equality, as $\nu_1 = \nu_2$; this notation papers over the true property in mind, $\nu_1 \equiv_\mu \nu_2$. And when module identities are treated as an *index* into a mapping, like in Φ, an actual implementation would likely index said mapping by normalizing identities with $\mathsf{norm}(-)$.

- For the elaboration into the IL, we assume an injection from module identities (up to $\equiv_\alpha$ equivalence) to plain Haskell module names,

$$(-)^\star \quad : \quad \mathit{ModIdents}/{\equiv_\alpha} \rightarrowtail \mathit{ILModNames}$$

We lift this to an injection from module types τ (and other semantic objects) to plain Haskell interfaces *ftyp* which translates each stamp occurring in τ (and other semantic objects) to its injected module name,

$$(-)^\star \quad : \quad \mathit{ModTypes}/{\equiv_\alpha} \rightarrowtail \mathit{ILModTyps}$$

- The creation of a module identity, performed in the rule, has a straightforward mechnical definition, based on the module expression (M) and the ambient logical module context $(\mathcal{L})$. Creating a module identity from a logical module context:

$$\mathsf{mkident}(M;\mathcal{L}) \quad \overset{\text{def}}{=} \quad \mathcal{K}\ \nu_1\ \ldots\ \nu_n$$

$$\text{where} \quad \begin{cases} \mathcal{K} \text{ encodes } M \\ M \text{ imports } \ell_1,\ldots,\ell_n \\ \forall i \in [1..n] : \ \mathcal{L}(\ell_i) = \nu_i \end{cases}$$

2 Considine (2000) and Gauthier and Pottier (2004).

A.1.2 *Semantic Signatures*

- Coercing typey things into shapey things:

$$
\begin{array}{lcl}
\mathsf{shape}(\llparenthesis\, \Phi \,;\, \mathcal{L} \,\rrparenthesis) & \stackrel{\text{def}}{=} & (\, \mathsf{shape}(\Phi) \,;\, \mathsf{shape}(\mathcal{L}) \,) \\
\mathsf{shape}(\{\!\!\{ \overline{\nu{:}\tau^{m} @\, \omega} \}\!\!\}) & \stackrel{\text{def}}{=} & \{\, \overline{\nu{:}\mathsf{shape}(\tau)^{m} @\, \mathsf{shape}(\omega)} \,\} \\
\\
\mathsf{shape}(\langle\!| \, \mathit{dspcs} \,;\, \mathit{espcs} \,;\, \overline{\nu} \,|\!\rangle) & \stackrel{\text{def}}{=} & \langle\, \mathsf{shape}(\mathit{dspcs}) \,;\, \mathit{espcs} \,;\, \overline{\nu} \,\rangle \\
\mathsf{shape}(\{\!\!\{ \overline{\mathit{fact} \mapsto \nu} \}\!\!\}) & \stackrel{\text{def}}{=} & \{\, \overline{\mathit{fact} \mapsto \nu} \,\} \\
\\
\mathsf{shape}(x :: \mathit{typ}) & \stackrel{\text{def}}{=} & x \\
\mathsf{shape}(\mathtt{data}\ \mathsf{T}\ \mathit{kenv}) & \stackrel{\text{def}}{=} & \mathsf{T} \\
\mathsf{shape}(\mathtt{data}\ \mathsf{T}\ \mathit{kenv} = \overline{\mathsf{K}\ \overline{\mathit{typ}}}) & \stackrel{\text{def}}{=} & \mathsf{T}(\overline{\mathsf{K}}) \\
\mathsf{shape}(\mathtt{class}\ \mathsf{C}\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \overline{x :: \mathit{typ}}) & \stackrel{\text{def}}{=} & \mathsf{C}(\overline{x}) \\
\mathsf{shape}(\mathtt{instance}\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \mathit{cls}) & \stackrel{\text{def}}{=} & \mathtt{instance}
\end{array}
$$

- The domain of a binding signature or context, and the restriction to a given prefix:

$$
\begin{array}{lcl}
\mathsf{dom}(\{\!\!\{ \overline{\nu{:}\tau^{m} @\, \omega} \}\!\!\}) & \stackrel{\text{def}}{=} & \{\overline{\nu}\} \\
\mathsf{dom}(\overline{\ell \mapsto \nu}) & \stackrel{\text{def}}{=} & \{\overline{\ell}\} \\
\\
\mathsf{dom}_{p}(\mathcal{L}) & \stackrel{\text{def}}{=} & \{p' \mid p' = p.p'',\ p' \in \mathsf{dom}(\mathcal{L})\} \\
\mathsf{dom}_{p}((\Phi;\mathcal{L})) & \stackrel{\text{def}}{=} & \mathsf{dom}_{p}(\mathcal{L})
\end{array}
$$

defined similarly for shapes and contexts

A.1.2.1 *Local well-formedness and augmented entity environments*

- $\boxed{\mathit{aenv} \Vdash \mathit{eenv}\ \mathsf{loc\text{-}wf}}$ Determine whether the physical names in the range of the entity environment are indeed exported from their defining modules, or from a local definition. Also check that the locally available export specs all make sense.

$$
\mathit{aenv} \Vdash \mathit{eenv}\ \mathsf{loc\text{-}wf} \ \stackrel{\text{def}}{\Leftrightarrow} \
\begin{cases}
\forall \mathit{espc} \in \mathit{eenv} : & \begin{cases} \mathit{aenv} \Vdash \mathit{espc}\ \mathsf{loc\text{-}wf} \\ \mathsf{allphnms}(\mathit{espc}) \subseteq \mathsf{rng}(\mathit{eenv}) \end{cases} \\
\forall \mathit{eref} \mapsto \mathit{phnm} \in \mathit{eenv} : & \begin{cases} \mathit{aenv} \Vdash \mathit{eref} \mapsto \mathit{phnm}\ \mathsf{loc\text{-}wf} \\ \mathit{phnm} \in \mathsf{allphnms}(\mathsf{locals}(\mathit{eenv})) \end{cases}
\end{cases}
$$

- $\boxed{\mathit{aenv} \Vdash \mathit{eref} \mapsto \mathit{phnm}\ \mathsf{loc\text{-}wf}}$ Determine whether the physical name in the range of the mapping is indeed exported from its defining module, or from a local definition.

$$
\frac{\begin{array}{c} \mathit{eref} = \chi \text{ or } \mathit{mref}\mathtt{.}\chi \\ (\mathit{eref} = \mathtt{Local.}\chi) \ \Rightarrow\ \mathsf{islocal}(\mathit{aenv}; [\nu]\chi) \end{array}}{\mathit{aenv} \Vdash \mathit{eref} \mapsto [\nu]\chi\ \mathsf{loc\text{-}wf}}
$$

- $\boxed{\mathit{aenv} \Vdash \mathit{espc}\ \textsf{loc-wf}}$ $\boxed{\mathit{aenv} \Vdash \mathit{espcs}\ \textsf{loc-wf}}$ Check that a locally available export name spec is well formed with respect to a local definition and/or a spec in the context.

$$\frac{\mathit{espc}_0 = \textsf{locmatch}(\mathit{aenv}; \mathit{espc}) \quad \mathit{espc}_1 = \textsf{ctxmatch}(\mathit{aenv}; \mathit{espc}) \quad \mathit{espc}_0 \oplus \mathit{espc}_1 \leqslant \mathit{espc}}{\mathit{aenv} \Vdash \mathit{espc}\ \textsf{loc-wf}}$$

$$\frac{\mathit{espc}_0 = \textsf{locmatch}(\mathit{aenv}; \mathit{espc}) \quad \textsf{noctxmatch}(\mathit{aenv}; \mathit{espc}) \quad \mathit{espc}_0 \leqslant \mathit{espc}}{\mathit{aenv} \Vdash \mathit{espc}\ \textsf{loc-wf}}$$

$$\frac{\textsf{nolocmatch}(\mathit{aenv}; \mathit{espc}) \quad \mathit{espc}_1 = \textsf{ctxmatch}(\mathit{aenv}; \mathit{espc}) \quad \mathit{espc}_1 \leqslant \mathit{espc}}{\mathit{aenv} \Vdash \mathit{espc}\ \textsf{loc-wf}}$$

$$\mathit{aenv} \Vdash \mathit{espcs}\ \textsf{loc-wf} \overset{\text{def}}{\Leftrightarrow} \forall \mathit{espc} \in \mathit{espcs} : \mathit{aenv} \Vdash \mathit{espc}\ \textsf{loc-wf}$$

- $\boxed{\textsf{provs}(\omega) : \textsf{N}}$ $\boxed{\textsf{provs}(\mathit{fact}) : \textsf{N}}$ $\boxed{\textsf{provs}(\mathit{cls}) : \textsf{N}}$ Identities mentioned in worlds.

$$\textsf{provs}(\omega) \overset{\text{def}}{=} \bigcup_{\mathit{fact} \in \omega} \textsf{provs}(\mathit{fact})$$

$$\textsf{provs}(\mathit{kenv.cls} \mapsto \nu) \overset{\text{def}}{=} \textsf{provs}(\mathit{cls}) \cup \{\nu\}$$

$$\textsf{provs}([\nu]\textsf{C}\ \overline{\mathit{typ}}) \overset{\text{def}}{=} \{\nu\} \cup \bigcup_{\mathit{typ} \in \overline{\mathit{typ}}} \textsf{provs}(\mathit{typ})$$

- $\boxed{\mathit{aenv}; \mathit{eenv} \Vdash \mathit{fact}\ \textsf{loc-wf}}$ $\boxed{\mathit{aenv}; \mathit{eenv} \Vdash \widehat{\omega}\ \textsf{loc-wf}}$ Local well-formedness for world shapes and world facts.

$$\frac{\mathit{aenv}.\widehat{\Phi} \Vdash \mathit{head} \mapsto \nu\ \textsf{wf}}{\mathit{aenv}; \mathit{eenv} \Vdash \mathit{fact}\ \textsf{loc-wf}}\ (\textsc{ShWfLocWorldCtx})$$

$$\frac{\mathit{def} \in \mathit{defs} \quad \nu = \nu_0 \quad \mathit{eenv}(\textsf{head}(\mathit{def})) = \mathit{kenv.cls}}{(\widehat{\Phi}; \nu_0; \mathit{defs}); \mathit{eenv} \Vdash \mathit{kenv.cls} \mapsto \nu\ \textsf{loc-wf}}\ (\textsc{ShWfLocWorldDef})$$

$$\frac{\mathit{decl} \in \mathit{decls} \quad \mathit{kenv.cls} \mapsto \nu \in \widehat{\omega}_0 \quad \mathit{eenv}(\textsf{head}(\mathit{decl})) = \mathit{kenv.cls}}{(\widehat{\Phi}; \widehat{\tau}_0 \mathbin{@} \widehat{\omega}_0; \mathit{decls}); \mathit{eenv} \Vdash \mathit{kenv.cls} \mapsto \nu\ \textsf{loc-wf}}\ (\textsc{ShWfLocWorldDecl})$$

$$\mathit{aenv}; \mathit{eenv} \Vdash \widehat{\omega}\ \textsf{loc-wf} \overset{\text{def}}{\Leftrightarrow} \forall \mathit{fact} \in \widehat{\omega} : \mathit{aenv}; \mathit{eenv} \Vdash \mathit{fact}\ \textsf{loc-wf}$$

- $\boxed{\widehat{\Phi} \Vdash \mathit{head} \mapsto \nu\ \textsf{wf}}$ $\boxed{\widehat{\Phi} \Vdash \widehat{\omega}\ \textsf{wf}}$ Shapey well-formedness of facts and worlds. The former is *not* defined like its typey counterpart.

$$\frac{\mathit{kenv.cls} \mapsto \nu \in \textsf{world}_{\widehat{\Phi}}(\nu)}{\widehat{\Phi} \Vdash \mathit{kenv.cls} \mapsto \nu\ \textsf{wf}} \qquad \frac{\overline{\widehat{\Phi} \Vdash \mathit{head} \mapsto \nu\ \textsf{wf}}}{\widehat{\Phi} \Vdash \{\overline{\mathit{fact}}\}\ \textsf{wf}}$$

- $\boxed{\textsf{islocal}(\mathit{aenv}; \mathit{espc})}$ $\boxed{\textsf{islocal}(\mathit{aenv}; \mathit{phnm})}$ Is the given entity's identity the same as the local module? Or, in the case of signatures, is this entity one of the locally specified ones? (This does *not* check for a matching definition/declaration; it merely looks at the identity.)

$$\begin{aligned}
\textsf{islocal}((\widehat{\Phi}; \nu_0; \mathit{defs}); \mathit{espc}) &\overset{\text{def}}{\Leftrightarrow} \textsf{ident}(\mathit{espc}) = \nu_0 \\
\textsf{islocal}((\widehat{\Phi}; \widehat{\tau}_0 \mathbin{@} \widehat{\omega}_0; \mathit{decls}); \mathit{espc}) &\overset{\text{def}}{\Leftrightarrow} \exists \mathit{espc}' \in \widehat{\tau}_0 : \mathit{espc}' \oplus_? \mathit{espc} \\
\textsf{islocal}((\widehat{\Phi}; \nu_0; \mathit{defs}); [\nu]\chi) &\overset{\text{def}}{\Leftrightarrow} \nu = \nu_0 \\
\textsf{islocal}((\widehat{\Phi}; \widehat{\tau}_0 \mathbin{@} \widehat{\omega}_0; \mathit{decls}); [\nu]\chi) &\overset{\text{def}}{\Leftrightarrow} \exists \mathit{espc}' \in \widehat{\tau}_0 : \mathit{espc}' \sqsubseteq [\nu]\chi
\end{aligned}$$

- $\boxed{\mathsf{locmatch}(aenv; espc)}$ $\boxed{\mathsf{ctxmatch}(aenv; espc)}$ $\boxed{\mathsf{nolocmatch}(aenv; espc)}$ $\boxed{\mathsf{noctxmatch}(aenv; espc)}$ Find a matching export spec in the context or the local environment.

$$\mathsf{locmatch}(aenv; espc) \overset{\text{def}}{=} espc' \quad \text{if} \begin{cases} \mathsf{islocal}(aenv; espc), \\ \exists bnd \in aenv.bnds :\ bnd \sqsubseteq espc', \\ espc' \ \oplus_? \ espc \end{cases}$$

$$\mathsf{ctxmatch}(aenv; espc) \overset{\text{def}}{=} espc' \quad \text{if} \begin{cases} espc' \in aenv.\hat{\Phi}(\mathsf{ident}(espc)), \\ espc' \ \oplus_? \ espc \end{cases}$$

$$\mathsf{nolocmatch}(aenv; espc) \overset{\text{def}}{\Leftrightarrow} \mathsf{islocal}(aenv; espc) \Rightarrow \forall espc', bnd \in aenv.bnds :\ bnd \sqsubseteq espc' \Rightarrow espc' \not\oplus_? espc$$

$$\mathsf{noctxmatch}(aenv; espc) \overset{\text{def}}{\Leftrightarrow} \forall espc' \in aenv.\hat{\Phi}(\mathsf{ident}(espc)) : espc' \not\oplus_? espc$$

- $\boxed{\mathsf{haslocaleenv}(eenv; rbnds)}$ Specifies that *eenv* contains all the locally bound entities in *rbnds* and, moreover, that this subset is disjoint from the remainder of the environment.

$$\mathsf{haslocaleenv}(eenv; rbnds; \quad) \overset{\text{def}}{\Leftrightarrow} \exists eenv' : \begin{cases} eenv = \mathsf{mklocaleenv}(\mathsf{mkloceenv}(;) \oplus)eenv' \\ \forall eref \in \mathsf{dom}(eenv') :\ eref \neq \texttt{Local.}\chi \end{cases}$$

- $\boxed{\mathsf{allphnms}(espc)}$ Get the set of all physical names mentioned by the *espc*.

$$\begin{aligned} \mathsf{allphnms}(espc) &\overset{\text{def}}{=} \{[\mathsf{ident}(espc)]\chi \mid \chi \in \mathsf{allnames}(espc)\} \\ \mathsf{allphnms}(\overline{espc}) &\overset{\text{def}}{=} \textstyle\bigcup_{espc \in \overline{espc}} \mathsf{allphnms}(espc) \end{aligned}$$

- $\boxed{eenv \ \oplus \ eenv}$ $\boxed{eenv \ \oplus_? \ eenv}$ Merge two entity environments.

$$\begin{aligned} eenv_1 \ \oplus \ eenv_2 &\overset{\text{def}}{=} \{eref \mapsto phnm \mid eref \mapsto phnm \in eenv_1 \text{ or } eenv_2\}; \\ &\quad\ \langle\!|\mathsf{locals}(eenv_1) \ \oplus \ \mathsf{locals}(eenv_2)|\!\rangle \\ eenv_1 \ \oplus_? \ eenv_2 &\overset{\text{def}}{\Leftrightarrow} \mathsf{locals}(eenv_1) \ \oplus_? \ \mathsf{locals}(eenv_2) \end{aligned}$$

- $\boxed{eenv(eref)}$ Look up the (single!) physical name referred to by *eref* in the environment. See Figure 6.7.

- $\boxed{eenv(eref) = phnm : espc}$ Look up the (single!) physical name referred to by *eref* in the environment, and also extract the relevant local *espc* from the *eenv*. See Figure 6.7.

- $\boxed{\mathsf{rng}(eenv) \ :_{\mathsf{par}} \ \overline{phnm}}$ Get the set of all physical names that entities in the environment refer to.

$$\mathsf{rng}(eenv) \overset{\text{def}}{=} \{phnm \mid eref \mapsto phnm \in eenv\}$$

- $\boxed{\mathsf{name}(\mathit{dspc})}$ $\boxed{\mathsf{name}(\mathit{espc})}$ $\boxed{\mathsf{name}(\mathit{def})}$ $\boxed{\mathsf{name}(\mathit{decl})}$ Get the name of some entity.

$$\begin{aligned}
\mathsf{name}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}) &\stackrel{\text{def}}{=} \mathsf{T}\\
\mathsf{name}(\texttt{data}\ \mathsf{T}\ \mathit{kenv} = \overline{\mathsf{K}\ \overline{\mathit{typ}}}) &\stackrel{\text{def}}{=} \mathsf{T}\\
\mathsf{name}(x :: \mathit{typ}) &\stackrel{\text{def}}{=} x\\
\mathsf{name}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\{\mathit{cls}\}}\ \overline{x :: \mathit{typ}}) &\stackrel{\text{def}}{=} \mathsf{C}\\
\mathsf{name}(\texttt{instance}\ \mathit{kenv}\ \overline{\{\mathit{cls}\}}\ \mathit{cls}) &\quad \text{undefined}
\end{aligned}$$

$$\begin{aligned}
\mathsf{name}([\nu]\chi(\overline{\chi}')) &\stackrel{\text{def}}{=} \chi\\
\mathsf{name}([\nu]\chi) &\stackrel{\text{def}}{=} \chi
\end{aligned}$$

$$\begin{aligned}
\mathsf{name}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}\ \texttt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}}) &\stackrel{\text{def}}{=} \mathsf{T}\\
\mathsf{name}(x\ [\ \texttt{::}\ \mathit{utyp}]\ \texttt{=}\ \mathit{uexp}) &\stackrel{\text{def}}{=} x\\
\mathsf{name}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\texttt{<=}\,\mathit{ucls}}\ \texttt{where}\ \overline{x\ \texttt{::}\ \mathit{utyp}}) &\stackrel{\text{def}}{=} \mathsf{C}\\
\mathsf{name}(\texttt{instance}\ldots) &\quad \text{undefined}
\end{aligned}$$

$$\begin{aligned}
\mathsf{name}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}) &\stackrel{\text{def}}{=} \mathsf{T}\\
\mathsf{name}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}\ \texttt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}}) &\stackrel{\text{def}}{=} \mathsf{T}\\
\mathsf{name}(x\ \texttt{::}\ \mathit{utyp}) &\stackrel{\text{def}}{=} x\\
\mathsf{name}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\texttt{<=}\,\mathit{ucls}}\ \texttt{where}\ \overline{x\ \texttt{::}\ \mathit{utyp}}) &\stackrel{\text{def}}{=} \mathsf{C}\\
\mathsf{name}(\texttt{instance}\ldots) &\quad \text{undefined}
\end{aligned}$$

- $\boxed{\mathsf{names}(\mathit{dspc})}$ $\boxed{\mathsf{names}(\mathit{espc})}$ $\boxed{\mathsf{names}(\mathit{def})}$ $\boxed{\mathsf{names}(\mathit{decl})}$ Get the subordinate names of some entity, such as data constructors or class methods.

$$\begin{aligned}
\mathsf{names}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}) &\stackrel{\text{def}}{=} \emptyset\\
\mathsf{names}(\texttt{data}\ \mathsf{T}\ \mathit{kenv} = \overline{\mathsf{K}\ \overline{\mathit{typ}}}) &\stackrel{\text{def}}{=} \{\overline{\mathsf{K}}\}\\
\mathsf{names}(x :: \mathit{typ}) &\quad \text{undefined}\\
\mathsf{names}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\{\mathit{cls}\}}\ \overline{x :: \mathit{typ}}) &\stackrel{\text{def}}{=} \{\overline{x}\}\\
\mathsf{names}(\texttt{instance}\ \mathit{kenv}\ \overline{\{\mathit{cls}\}}\ \mathit{cls}) &\quad \text{undefined}
\end{aligned}$$

$$\begin{aligned}
\mathsf{names}([\nu]\chi(\overline{\chi}')) &\stackrel{\text{def}}{=} \{\overline{\chi}'\}\\
\mathsf{names}([\nu]\chi) &\quad \text{undefined}
\end{aligned}$$

$$\begin{aligned}
\mathsf{names}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}\ \texttt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}}) &\stackrel{\text{def}}{=} \{\overline{\mathsf{K}}\}\\
\mathsf{names}(x\ [\ \texttt{::}\ \mathit{utyp}]\ \texttt{=}\ \mathit{uexp}) &\quad \text{undefined}\\
\mathsf{names}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\texttt{<=}\,\mathit{ucls}}\ \texttt{where}\ \overline{x\ \texttt{::}\ \mathit{utyp}}) &\stackrel{\text{def}}{=} \{\overline{x}\}\\
\mathsf{names}(\texttt{instance}\ldots) &\quad \text{undefined}
\end{aligned}$$

$$\begin{aligned}
\mathsf{names}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}) &\stackrel{\text{def}}{=} \{\}\\
\mathsf{names}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}\ \texttt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}}) &\stackrel{\text{def}}{=} \{\overline{\mathsf{K}}\}\\
\mathsf{names}(x\ \texttt{::}\ \mathit{utyp}) &\quad \text{undefined}\\
\mathsf{names}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\texttt{<=}\,\mathit{ucls}}\ \texttt{where}\ \overline{x\ \texttt{::}\ \mathit{utyp}}) &\stackrel{\text{def}}{=} \{\overline{x}\}\\
\mathsf{names}(\texttt{instance}\ldots) &\quad \text{undefined}
\end{aligned}$$

- $\boxed{\mathsf{hasSubs}(\mathit{dspc})}$ $\boxed{\mathsf{hasSubs}(\mathit{espc})}$ $\boxed{\mathsf{hasSubs}(\mathit{def})}$ $\boxed{\mathsf{hasSubs}(\mathit{decl})}$ Determine whether an entity has subordinate names.

$$\begin{aligned}
\mathsf{hasSubs}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}(\texttt{data}\ \mathsf{T}\ \mathit{kenv} = \overline{\mathsf{K}\ \overline{\mathit{typ}}}) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}(x\ \texttt{::}\ \mathit{typ}) &\stackrel{\mathsf{def}}{=} \mathsf{false}\\
\mathsf{hasSubs}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \overline{x\ \texttt{::}\ \mathit{typ}}) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}(\texttt{instance}\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \mathit{cls}) &\stackrel{\mathsf{def}}{=} \mathsf{false}
\end{aligned}$$

$$\begin{aligned}
\mathsf{hasSubs}([\nu]\chi(\overline{\chi}')) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}([\nu]\chi) &\stackrel{\mathsf{def}}{=} \mathsf{false}
\end{aligned}$$

$$\begin{aligned}
\mathsf{hasSubs}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}\ \texttt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}}) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}(x\ [\ \texttt{::}\ \ \mathit{utyp}]\ \texttt{=}\ \mathit{uexp}) &\stackrel{\mathsf{def}}{=} \mathsf{false}\\
\mathsf{hasSubs}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\texttt{<=}\mathit{ucls}}\ \texttt{where}\ \overline{x\ \texttt{::}\ \mathit{utyp}}) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}(\texttt{instance}\ldots) &\stackrel{\mathsf{def}}{=} \mathsf{false}
\end{aligned}$$

$$\begin{aligned}
\mathsf{hasSubs}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}(\texttt{data}\ \mathsf{T}\ \mathit{kenv}\ \texttt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}}) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}(x\ \ \texttt{::}\ \ \mathit{utyp}) &\stackrel{\mathsf{def}}{=} \mathsf{false}\\
\mathsf{hasSubs}(\texttt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\texttt{<=}\mathit{ucls}}\ \texttt{where}\ \overline{x\ \texttt{::}\ \mathit{utyp}}) &\stackrel{\mathsf{def}}{=} \mathsf{true}\\
\mathsf{hasSubs}(\texttt{instance}\ldots) &\stackrel{\mathsf{def}}{=} \mathsf{false}
\end{aligned}$$

- $\boxed{\mathsf{allnames}(\mathit{dspc})}$ $\boxed{\mathsf{allnames}(\mathit{espc})}$ $\boxed{\mathsf{allnames}(\mathit{def})}$ $\boxed{\mathsf{allnames}(\mathit{decl})}$ Get the set containing the name and any subordinate names of the given entity.

$$\mathsf{allnames}(A) \stackrel{\mathsf{def}}{=} \begin{cases} \{\mathsf{name}(A)\} \cup \mathsf{names}(A) & \text{if } \mathsf{hasSubs}(A)\\ \{\mathsf{name}(A)\} & \text{otherwise}\end{cases} \quad \text{for } A = \mathit{dspc}, \mathit{espc}, \mathit{def}, \mathit{decl}$$

- $\boxed{\mathsf{nooverlap}(\overline{\mathit{dspc}})}$ $\boxed{\mathsf{nooverlap}(\overline{\mathit{espc}})}$ $\boxed{\mathsf{nooverlap}(\overline{\mathit{def}})}$ $\boxed{\mathsf{nooverlap}(\overline{\mathit{decl}})}$ Get the set containing the name and any subordinate names of the given entity.

$$\mathsf{nooverlap}(A_1, \ldots, A_n) \stackrel{\mathsf{def}}{\Leftrightarrow} \forall i, j \in [1..n] \text{ s.t. } i \neq j : \ \mathsf{allnames}(A_i) \# \mathsf{allnames}(A_j)$$

for $A = \mathit{dspc}, \mathit{espc}, \mathit{def}, \mathit{decl}$

- $\boxed{\mathit{def} \sqsubseteq \mathit{dspc}}$ $\boxed{\mathit{def} \sqsubseteq \mathit{espc}}$ $\boxed{\mathit{dspc} \sqsubseteq \mathit{espc}}$ $\boxed{\mathit{espc} \sqsubseteq \mathit{phnm}}$ This relation specifies that two core entities are syntactically similar; that a definition syntactically matches a specification, that a specification syntactically matches an export specification, and that an export specification matches a physical name. This is a very weak statement; for example, the relation includes the $(\mathit{def}, \mathit{dspc})$ pair

$$(x :: \texttt{Int} = 5) \ \sqsubseteq\ (x :: [\nu_0]\texttt{Bool})$$

despite the obvious problem with typing.

$$\boxed{\mathit{def} \sqsubseteq \mathit{decl}}$$

$$\begin{array}{rcl}
\mathtt{data}\ \mathsf{T}\ \mathit{kenv}\ \mathtt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}} & \sqsubseteq & \mathtt{data}\ \mathsf{T}\ \mathit{kenv}\ \mathtt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}} \\
x\ [\ \mathtt{::}\ \mathit{utyp}]\ \mathtt{=}\ \mathit{uexp} & \sqsubseteq & x\ \mathtt{::}\ \mathit{utyp} \\
\mathtt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\mathtt{<=}\,\mathit{ucls}}\ \mathtt{where}\ \overline{x\ \mathtt{::}\ \mathit{utyp}} & \sqsubseteq & \mathtt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\mathtt{<=}\,\mathit{ucls}}\ \mathtt{where}\ \overline{x\ \mathtt{::}\ \mathit{utyp}} \\
\mathtt{instance}\ \mathit{kenv}\ \overline{\mathit{ucls}\,\mathtt{=>}}\ \mathit{ucls}\ \mathtt{where}\ \overline{x\ \mathtt{=}\ \mathit{uexp}} & \sqsubseteq & \mathtt{instance}\ \mathit{kenv}\ \overline{\mathit{ucls}\,\mathtt{=>}}\ \mathit{ucls}
\end{array}$$

$$\boxed{\mathit{decl} \sqsubseteq \mathit{dspc}}$$

$$\begin{array}{rcl}
\mathtt{data}\ \mathsf{T}\ \mathit{kenv} & \sqsubseteq & \mathtt{data}\ \mathsf{T}\ \mathit{kenv} \\
\mathtt{data}\ \mathsf{T}\ \mathit{kenv}\ \mathtt{=}\ \overline{\mathsf{K}\ \overline{\mathit{utyp}}} & \sqsubseteq & \mathtt{data}\ \mathsf{T}\ \mathit{kenv} = \overline{\mathsf{K}\ \overline{\mathit{typ}}} \\
x\ \mathtt{::}\ \mathit{utyp} & \sqsubseteq & x\ \mathtt{::}\ \mathit{typ} \\
\mathtt{class}\ \mathsf{C}\ \mathit{kenv}\ \overline{\mathtt{<=}\,\mathit{ucls}}\ \mathtt{where}\ \overline{x\ \mathtt{::}\ \mathit{utyp}} & \sqsubseteq & \mathtt{class}\ \mathsf{C}\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \overline{x\ \mathtt{::}\ \mathit{typ}} \\
\mathtt{instance}\ \mathit{kenv}\ \overline{\mathit{ucls}\,\mathtt{=>}}\ \mathit{ucls} & \sqsubseteq & \mathtt{instance}\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \mathit{cls}
\end{array}$$

$$\boxed{\mathit{dspc} \sqsubseteq \mathit{espc}}$$

$$\begin{array}{rcl}
\mathtt{data}\ \mathsf{T}\ \mathit{kenv} & \sqsubseteq & [\nu]\mathsf{T}() \\
\mathtt{data}\ \mathsf{T}\ \mathit{kenv} = \overline{\mathsf{K}\ \overline{\mathit{typ}}} & \sqsubseteq & [\nu]\mathsf{T}(\overline{\mathsf{K}}) \\
x\ \mathtt{::}\ \mathit{typ} & \sqsubseteq & [\nu]x \\
\mathtt{class}\ \mathsf{C}\ \mathit{kenv}\ \{\overline{\mathit{cls}}\}\ \overline{x\ \mathtt{::}\ \mathit{typ}} & \sqsubseteq & [\nu]\mathsf{C}(\overline{x})
\end{array}$$

$$\boxed{\mathit{espc} \sqsubseteq \mathit{phnm}}$$

$$\begin{array}{rcl}
[\nu]\chi(\overline{\chi}') & \sqsubseteq & [\nu]\chi \\
[\nu]\chi & \sqsubseteq & [\nu]\chi
\end{array}$$

$$\begin{array}{rcl}
\mathit{def} \sqsubseteq \mathit{espc} & \overset{\text{def}}{\Leftrightarrow} & \exists \mathit{decl}, \mathit{dspc} :\ \mathit{def} \sqsubseteq \mathit{decl} \sqsubseteq \mathit{dspc} \sqsubseteq \mathit{espc} \\
\mathit{decl} \sqsubseteq \mathit{espc} & \overset{\text{def}}{\Leftrightarrow} & \exists \mathit{dspc} :\ \mathit{decl} \sqsubseteq \mathit{dspc} \sqsubseteq \mathit{espc}
\end{array}$$

- $\boxed{\mathsf{validspc}(\mathit{dspc}; m)}$ States whether this specification is valid in a module type with the given polarity. This is required because not all kinds of specs are valid in both modules and signatures.

$$\mathsf{validspc}(\mathit{dspc}; m) \overset{\text{def}}{=} \begin{cases} \mathsf{false} & \text{if } m = + \text{ and } \mathit{dspc} = (\mathtt{data}\ \mathsf{T}\ \mathit{kenv}) \\ \mathsf{true} & \text{otherwise} \end{cases}$$

A.1.3 *Algebras for semantic objects*

A.1.3.1 *Partial merge operations and their definedness*

$$\Xi_1 \oplus \Xi_2 \overset{\text{def}}{=} (\Phi_1 \oplus \Phi_2; \mathcal{L}_1 \oplus \mathcal{L}_2)$$

$$\text{where} \begin{cases} \Xi_1 = (\Phi_1; \mathcal{L}_1) \\ \Xi_2 = (\Phi_2; \mathcal{L}_2) \end{cases}$$

$$(\Phi_1; \mathcal{L}_1) \oplus_? (\Phi_2; \mathcal{L}_2) \overset{\text{def}}{\Leftrightarrow} \Phi_1 \oplus_? \Phi_2 \wedge \mathcal{L}_1 \oplus_? \mathcal{L}_2$$

$$\mathcal{L}_1 \oplus \mathcal{L}_2 \overset{\mathsf{def}}{=} \overline{\ell \mapsto v}, \mathcal{L}'_1, \mathcal{L}'_2$$
$$\text{where} \begin{cases} \mathcal{L}_1 = \overline{\ell \mapsto v}, \mathcal{L}'_1 \\ \mathcal{L}_2 = \overline{\ell \mapsto v}, \mathcal{L}'_2 \\ \mathsf{dom}(\mathcal{L}'_1) \mathbin{\#} \mathsf{dom}(\mathcal{L}'_2) \end{cases}$$

$$\mathcal{L}_1 \oplus_? \mathcal{L}_2 \overset{\mathsf{def}}{\Leftrightarrow} \forall \ell, v_1, v_2 : \begin{pmatrix} \ell \mapsto v_1 \in \mathcal{L}_1 \\ \ell \mapsto v_2 \in \mathcal{L}_2 \end{pmatrix} \Rightarrow v_1 \oplus_? v_2$$

$$\Phi_1 \oplus \Phi_2 \overset{\mathsf{def}}{=} \overline{v{:}\tau^{m} @ \omega}, \Phi'_1, \Phi'_2$$
$$\text{where} \begin{cases} \Phi_1 = \overline{v{:}\tau_1^{m_1} @ \omega_1}, \Phi'_1 \\ \Phi_2 = \overline{v{:}\tau_2^{m_2} @ \omega_2}, \Phi'_2 \\ \overline{\tau^{m} @ \omega} = \overline{\tau_1^{m_1} @ \omega_1 \oplus \tau_2^{m_2} @ \omega_2} \\ \mathsf{dom}(\Phi'_1) \mathbin{\#} \mathsf{dom}(\Phi'_2) \end{cases}$$

$$\Phi_1 \oplus_? \Phi_2 \overset{\mathsf{def}}{\Leftrightarrow} \forall v, \tau_1, \tau_2, m_1, m_2, \omega_1, \omega_2 : \begin{pmatrix} v{:}\tau_1{}^{m_1} @ @ \omega_1 \in \Phi_1 \\ v{:}\tau_2{}^{m_2} @ @ \omega_2 \in \Phi_2 \end{pmatrix} \Rightarrow \tau_1^{m_1} @ \omega_1 \oplus_? \tau_2^{m_2} @ \omega_2$$

$$\begin{aligned}
\tau_1{}^{-} @ \omega_1 \oplus \tau_2{}^{-} @ \omega_2 &\overset{\mathsf{def}}{=} (\tau_1 \oplus \tau_2)^{-} @ (\omega_1 \oplus \omega_2) \\
\tau_1{}^{-} @ \omega_1 \oplus \tau_2{}^{+} @ \omega_2 &\overset{\mathsf{def}}{=} \tau_2{}^{+} @ \omega_2 \quad \text{if} \begin{cases} \tau_1 \geqslant \tau_2 \\ \omega_1 \sqsubseteq \omega_2 \end{cases} \\
\tau_1{}^{+} @ \omega_1 \oplus \tau_2{}^{-} @ \omega_2 &\overset{\mathsf{def}}{=} \tau_1{}^{+} @ \omega_1 \quad \text{if} \begin{cases} \tau_1 \leqslant \tau_2 \\ \omega_1 \sqsupseteq \omega_2 \end{cases} \\
\tau_1{}^{+} @ \omega_1 \oplus \tau_2{}^{+} @ \omega_2 &\overset{\mathsf{def}}{=} \tau_1{}^{+} @ \omega_1 \quad \text{if} \begin{cases} \tau_1 = \tau_2 \\ \omega_1 = \omega_2 \end{cases}
\end{aligned}$$

$$\tau_1{}^{m_1} @ \omega_1 \oplus_? \tau_2{}^{m_2} @ \omega_2 \overset{\mathsf{def}}{\Leftrightarrow} \begin{pmatrix} m_1, m_2 = -,- & \Rightarrow & \tau_1 \oplus_? \tau_2 \wedge \omega_1 \oplus_? \omega_2 \\ m_1, m_2 = -,+ & \Rightarrow & \tau_1 \geqslant \tau_2 \wedge \omega_1 \sqsubseteq \omega_2 \\ m_1, m_2 = +,- & \Rightarrow & \tau_1 \leqslant \tau_2 \wedge \omega_1 \sqsupseteq \omega_2 \\ m_1, m_2 = +,+ & \Rightarrow & \tau_1 = \tau_2 \wedge \omega_1 = \omega_2 \end{pmatrix}$$

$$\llparenthesis\, \overline{dspc_1} ; \overline{espc_1} ; \overline{v_1} \,\rrparenthesis \oplus \llparenthesis\, \overline{dspc_2} ; \overline{espc_2} ; \overline{v_2} \,\rrparenthesis \overset{\mathsf{def}}{=} \llparenthesis\, \overline{dspc} ; \overline{espc} ; \overline{v} \,\rrparenthesis$$
$$\text{where} \begin{cases} \overline{dspc} = \overline{dspc_1} \oplus \overline{dspc_2} \\ \overline{espc} = \overline{espc_1} \oplus \overline{espc_2} \\ \mathsf{nooverlap}(\overline{dspc}) \wedge \mathsf{nooverlap}(\overline{espc}) \\ \{\overline{v}\} = \{\overline{v_1}\} \cup \{\overline{v_2}\} = \{\overline{v_1}\} \text{ or } \{\overline{v_2}\} \end{cases}$$

$$\llparenthesis\, \overline{dspc_1}\ ;\ \overline{espc_1}\ ;\ \overline{\nu_1}\, \rrparenthesis \ \oplus_?\ \llparenthesis\, \overline{dspc_2}\ ;\ \overline{espc_2}\ ;\ \overline{\nu_2}\, \rrparenthesis \ \overset{\text{def}}{\Leftrightarrow} \ \begin{cases} \overline{dspc_1}\ \oplus_?\ \overline{dspc_2} \\ \overline{espc_1}\ \oplus_?\ \overline{espc_2} \\ \mathsf{nooverlap}(\overline{dspc_1}\ \oplus\ \overline{dspc_2}) \\ \mathsf{nooverlap}(\overline{espc_1}\ \oplus\ \overline{espc_2}) \\ \{\overline{\nu_1}\} \cup \{\overline{\nu_2}\} = \{\overline{\nu_1}\} \text{ or } \{\overline{\nu_2}\} \end{cases}$$

$$\overline{dspc_1}\ \oplus\ \overline{dspc_2} \ \overset{\text{def}}{=}\ \overline{dspc}, \overline{dspc''_1}, \overline{dspc''_2} \qquad \overline{espc_1}\ \oplus\ \overline{espc_2} \ \overset{\text{def}}{=}\ \overline{espc}, \overline{espc''_1}, \overline{espc''_2}$$

$$\text{where} \begin{cases} \overline{dspc_1} & = & \overline{dspc'_1}, \overline{dspc''_1} \\ \overline{dspc_2} & = & \overline{dspc'_2}, \overline{dspc''_2} \\ \overline{dspc} & = & \overline{dspc'_1\ \oplus\ dspc'_2} \\ \multicolumn{3}{l}{\overline{\mathsf{name}(dspc''_1)}\ \#\ \overline{\mathsf{name}(dspc''_2)}} \end{cases} \qquad \text{where} \begin{cases} \overline{espc_1} & = & \overline{espc'_1}, \overline{espc''_1} \\ \overline{espc_2} & = & \overline{espc'_2}, \overline{espc''_2} \\ \overline{espc} & = & \overline{espc'_1\ \oplus\ espc'_2} \\ \multicolumn{3}{l}{\overline{\mathsf{mkphnm}(espc''_1)}\ \#\ \overline{\mathsf{mkphnm}(espc''_2)}} \end{cases}$$

$$\overline{dspc_1}\ \oplus_?\ \overline{dspc_2} \ \overset{\text{def}}{\Leftrightarrow}\ \forall dspc_1, dspc_2 : \begin{pmatrix} \mathsf{name}(dspc_1) = \mathsf{name}(dspc_2) \\ dspc_1 \in \overline{dspc_1}\ \wedge\ dspc_2 \in \overline{dspc_2} \end{pmatrix} \Rightarrow dspc_1\ \oplus_?\ dspc_2$$

$$\overline{espc_1}\ \oplus_?\ \overline{espc_2} \ \overset{\text{def}}{\Leftrightarrow}\ \forall espc_1, espc_2 : \begin{pmatrix} \mathsf{mkphnm}(espc_1) = \mathsf{mkphnm}(espc_2) \\ espc_1 \in \overline{espc_1}\ \wedge\ espc_2 \in \overline{espc_2} \end{pmatrix} \Rightarrow espc_1\ \oplus_?\ espc_2$$

$$\begin{array}{rcl} (\mathtt{data}\ \mathsf{T}\ kenv)\ \oplus\ (\mathtt{data}\ \mathsf{T}\ kenv = \overline{\mathsf{K}\ \overline{typ}}) & \overset{\text{def}}{=} & \mathtt{data}\ \mathsf{T}\ kenv = \overline{\mathsf{K}\ \overline{typ}} \\ (\mathtt{data}\ \mathsf{T}\ kenv = \overline{\mathsf{K}\ \overline{typ}})\ \oplus\ (\mathtt{data}\ \mathsf{T}\ kenv) & \overset{\text{def}}{=} & \mathtt{data}\ \mathsf{T}\ kenv = \overline{\mathsf{K}\ \overline{typ}} \\ dspc\ \oplus\ dspc & \overset{\text{def}}{=} & dspc \end{array}$$

$$dspc_1\ \oplus_?\ dspc_2 \ \overset{\text{def}}{\Leftrightarrow}\ dspc_1 = dspc_2 \vee \begin{pmatrix} dspc_1, dspc_2 = (\mathtt{data}\ \mathsf{T}\ kenv = \overline{\mathsf{K}\ \overline{typ}}), (\mathtt{data}\ \mathsf{T}\ kenv) \ \vee \\ dspc_1, dspc_2 = (\mathtt{data}\ \mathsf{T}\ kenv), (\mathtt{data}\ \mathsf{T}\ kenv = \overline{\mathsf{K}\ \overline{typ}}) \end{pmatrix}$$

$$\begin{array}{rcl} [\nu]\chi(\overline{\chi_1})\ \oplus\ [\nu]\chi(\overline{\chi_2}) & \overset{\text{def}}{=} & [\nu]\chi(\overline{\chi'}) \quad \text{where } \{\overline{\chi'}\} = \{\overline{\chi_1}\} \cup \{\overline{\chi_2}\} \\ [\nu]\chi\ \oplus\ [\nu]\chi & \overset{\text{def}}{=} & [\nu]\chi \end{array}$$

$$espc_1\ \oplus_?\ espc_2 \ \overset{\text{def}}{\Leftrightarrow} \begin{pmatrix} \mathsf{ident}(espc_1) = \mathsf{ident}(espc_2) \\ \mathsf{name}(espc_1) = \mathsf{name}(espc_2) \\ \mathsf{hasSubs}(espc_1) = \mathsf{hasSubs}(espc_2) \end{pmatrix}$$

A.1.3.2 *Partial order (induced from merge)*

$$\llparenthesis\, \overline{dspc_1}, \overline{dspc'_1}\ ;\ \overline{espc_1}, \overline{espc'_1}\ ;\ \overline{\nu_1}\, \rrparenthesis \leqslant \llparenthesis\, \overline{dspc_2}\ ;\ \overline{espc_2}\ ;\ \overline{\nu_2}\, \rrparenthesis$$
$$\overset{\text{def}}{\Leftrightarrow}$$
$$\overline{dspc_1 \leqslant dspc_2}\ \wedge\ \overline{espc_1 \leqslant espc_2}\ \wedge\ \{\overline{\nu_1}\} \supseteq \{\overline{\nu_2}\}$$

$$\begin{array}{rcl} (\mathtt{data}\ \mathsf{T}\ kenv = \overline{\mathsf{K}\ \overline{typ}}) \leqslant (\mathtt{data}\ \mathsf{T}\ kenv) & \overset{\text{def}}{\Leftrightarrow} & \text{always} \\ dspc \leqslant dspc & \overset{\text{def}}{\Leftrightarrow} & \text{always} \end{array}$$

$$\begin{array}{rcl} [\nu]\chi(\overline{\chi_1}) \leqslant [\nu]\chi(\overline{\chi_2}) & \overset{\text{def}}{\Leftrightarrow} & \{\overline{\chi_1}\} \supseteq \{\overline{\chi_2}\} \\ [\nu]\chi \leqslant [\nu]\chi & \overset{\text{def}}{\Leftrightarrow} & \text{always} \end{array}$$

A.2 INTERNAL LANGUAGE

• EL physical module contexts are isomorphic to file environments whose file names all lie in the range of the translation function $(-)^{\star}$. Similarly with EL module types and source types, and EL identity substitutions and file name replacements. We implicitly rely on this isomorphism by not duplicating definitions which are "parametric" in the identity/file names, like all semantic object well-formedness judgments and algebraic definitions.

• File name replacement in various syntactic forms. Replacement on file names, source modules, file types, and file expressions is straightforward as there are no binding sites and thus no concern for capture. Replacement on directory expressions is trickier since we must recursively merge each substituted file into the rest. (Consider what happens when a *dexp* containing two files/file names get substituted with the same name.)

$$\begin{aligned}
\{\mathsf{f} := \mathsf{f}', \overline{\mathsf{g} := \mathsf{g}'}\}\mathsf{f} &\overset{\text{def}}{=} \mathsf{f}' \quad \text{where } \mathsf{f} \notin \overline{\mathsf{g}} \\
\{\overline{\mathsf{g} := \mathsf{g}'}\}\mathsf{f} &\overset{\text{def}}{=} \mathsf{f} \quad \text{where } \mathsf{f} \notin \overline{\mathsf{g}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{apply}(\mathring{\theta};\{\}) &\overset{\text{def}}{=} \{\} \\
\mathsf{apply}(\mathring{\theta};\{\mathsf{f} \mapsto \mathit{tfexp} \,@\, \mathring{\omega}, \overline{\mathsf{f}' \mapsto \mathit{tfexp}' \,@\, \mathring{\omega}'}\}) &\overset{\text{def}}{=} \{\mathring{\theta}\mathsf{f} \mapsto \mathring{\theta}\mathit{tfexp} \,@\, \mathring{\theta}\mathring{\omega}\} \oplus \mathsf{apply}(\mathring{\theta};\{\overline{\mathsf{f}' \mapsto \mathit{tfexp}' \,@\, \mathring{\omega}'}\})
\end{aligned}$$

• Note that we require this operation to respect translation:

$$\phi^{\star}(\nu^{\star}) \overset{\text{def}}{=} (\phi(\nu))^{\star}$$

• Merging of (typed) file expressions, file types, entity specs, export specs, directory expressions, directory types, and file environments.

$$\begin{aligned}
(\mathit{hsmod}_1 : \mathit{ftyp}_1 \,@\, \mathring{\omega}_1) \oplus (\mathit{hsmod} : \mathit{ftyp}_2 \,@\, \mathring{\omega}_2) &\overset{\text{def}}{=} \mathit{hsmod}_1 : \mathit{ftyp}_1 \,@\, \mathring{\omega}_1 \\
&\quad \text{where} \begin{cases} \mathit{ftyp}_1 = \mathit{ftyp}_2 \\ \mathring{\omega}_1 = \mathring{\omega}_2 \end{cases} \\
(\mathit{hsmod}_1 : \mathit{ftyp}_1 \,@\, \mathring{\omega}_1) \oplus (- : \mathit{ftyp}_2 \,@\, \mathring{\omega}_2) &\overset{\text{def}}{=} \mathit{hsmod}_1 : \mathit{ftyp}_1 \,@\, \mathring{\omega}_1 \\
&\quad \text{where} \begin{cases} \mathit{ftyp}_1 \leqslant \mathit{ftyp}_2 \\ \mathring{\omega}_1 \sqsupseteq \mathring{\omega}_2 \end{cases} \\
(- : \mathit{ftyp}_1 \,@\, \mathring{\omega}_1) \oplus (\mathit{hsmod}_2 : \mathit{ftyp}_2 \,@\, \mathring{\omega}_2) &\overset{\text{def}}{=} \mathit{hsmod}_2 : \mathit{ftyp}_2 \,@\, \mathring{\omega}_2 \\
&\quad \text{where} \begin{cases} \mathit{ftyp}_2 \leqslant \mathit{ftyp}_1 \\ \mathring{\omega}_2 \sqsupseteq \mathring{\omega}_1 \end{cases} \\
(- : \mathit{ftyp}_1 \,@\, \mathring{\omega}_1) \oplus (- : \mathit{ftyp}_2 \,@\, \mathring{\omega}_2) &\overset{\text{def}}{=} - : (\mathit{ftyp}_1 \oplus \mathit{ftyp}_2) \,@\, (\mathring{\omega}_1 \oplus \mathring{\omega}_2) \\
&\quad \text{where} \begin{cases} \mathit{ftyp}_1 \oplus_? \mathit{ftyp}_2 \\ \mathring{\omega}_1 \oplus_? \mathring{\omega}_2 \end{cases}
\end{aligned}$$

$$\mathit{dexp}_1 \oplus \mathit{dexp}_2 \overset{\text{def}}{=} \left\{ \begin{array}{l} \overline{\mathsf{f} \mapsto (\mathit{tfexp}_1 \,@\, \mathring{\omega}_1 \oplus \mathit{tfexp}_2 \,@\, \mathring{\omega}_2)}, \\ \overline{\mathsf{f}_1 \mapsto \mathit{tfexp}'_1 \,@\, \mathring{\omega}'_1}, \\ \overline{\mathsf{f}_2 \mapsto \mathit{tfexp}'_2 \,@\, \mathring{\omega}'_2} \end{array} \right\}$$

$$\text{where} \begin{cases} \mathit{dexp}_1 = \overline{\{\mathsf{f} \mapsto \mathit{tfexp}_1 \,@\, \mathring{\omega}_1\}}, \overline{\{\mathsf{f}_1 \mapsto \mathit{tfexp}'_1 \,@\, \mathring{\omega}'_1\}} \\ \mathit{dexp}_2 = \overline{\{\mathsf{f} \mapsto \mathit{tfexp}_2 \,@\, \mathring{\omega}_2\}}, \overline{\{\mathsf{f}_2 \mapsto \mathit{tfexp}'_2 \,@\, \mathring{\omega}'_2\}} \\ \overline{\mathsf{f}_1} \,\#\, \overline{\mathsf{f}_2} \end{cases}$$

- $\boxed{\mathit{dexp}|_{\mathsf{F}}}$ Filtering a directory expression.

$$\mathit{dexp}|_{\mathsf{F}} \overset{\mathsf{def}}{=} \{\mathsf{f} \mapsto \mathit{tfexp} @ \hat{\omega} \in \mathit{dexp} \mid \mathsf{f} \in \mathsf{F}\}$$

- $\boxed{\mathsf{alias}(\mathring{\mathit{impdecl}})}$ $\boxed{\mathsf{aliases}(\mathring{\mathit{impdecls}})}$ $\boxed{\mathsf{aliases}(\mathit{hsmod})}$ The name or names of module name aliases on imports.

$$\begin{array}{lcl}
\mathsf{alias}(\mathtt{import}\ [\mathtt{qualified}]\ \mathsf{f}\ \mathtt{as}\ \mathsf{f}') & \overset{\mathsf{def}}{=} & \mathsf{f}' \\
\mathsf{aliases}(\mathring{\mathit{impdecls}}) & \overset{\mathsf{def}}{=} & \{\mathsf{alias}(\mathring{\mathit{impdecl}}) \mid \mathring{\mathit{impdecl}} \in \mathring{\mathit{impdecls}}\} \\
\mathsf{aliases}(\mathtt{module}\ \mathsf{f}_0\ \mathring{\mathit{expdecl}}\ \mathtt{where}\ \mathring{\mathit{impdecls}}\mathtt{;}\ \mathit{defs}) & \overset{\mathsf{def}}{=} & \mathsf{aliases}(\mathring{\mathit{impdecls}})
\end{array}$$

- $\boxed{\mathsf{imp}(\mathring{\mathit{impdecl}})}$ $\boxed{\mathsf{imps}(\mathring{\mathit{impdecls}})}$ $\boxed{\mathsf{imps}(\mathit{hsmod})}$ The name or names of imported modules.

$$\begin{array}{lcl}
\mathsf{imp}(\mathtt{import}\ [\mathtt{qualified}]\ \mathsf{f}\ \mathtt{as}\ \mathsf{f}') & \overset{\mathsf{def}}{=} & \mathsf{f} \\
\mathsf{imps}(\mathring{\mathit{impdecls}}) & \overset{\mathsf{def}}{=} & \{\mathsf{imp}(\mathring{\mathit{impdecl}}) \mid \mathring{\mathit{impdecl}} \in \mathring{\mathit{impdecls}}\} \\
\mathsf{imps}(\mathtt{module}\ \mathsf{f}_0\ \mathring{\mathit{expdecl}}\ \mathtt{where}\ \mathring{\mathit{impdecls}}\mathtt{;}\ \mathit{defs}) & \overset{\mathsf{def}}{=} & \mathsf{imps}(\mathring{\mathit{impdecls}})
\end{array}$$

A.2.1 *Augmented environments in IL*

With the augmented entity environments there is not a direct translation from EL to IL definitions because the IL $\mathring{\mathit{aenv}}$ contains a full *fenv*, rather than something corresponding to $\hat{\Phi}$. The semantics are copied below, but the only other change lies in the $\mathring{\mathit{eenv}}$ mapping judgment, which no longer requires that "self" entity references have a local identity.

- $\boxed{\mathring{\mathit{aenv}} \vdash \mathring{\mathit{eenv}}\ \mathsf{loc\text{-}wf}}$ Determine whether the physical names in the range of the entity environment are indeed exported from their defining modules, or from a local definition. Also check that the locally available export specs all make sense.

$$\mathring{\mathit{aenv}} \vdash \mathring{\mathit{eenv}}\ \mathsf{loc\text{-}wf} \overset{\mathsf{def}}{\Leftrightarrow} \begin{array}{l}
\forall \mathring{\mathit{espc}} \in \mathring{\mathit{eenv}} : \begin{cases} \mathring{\mathit{aenv}} \vdash \mathring{\mathit{espc}}\ \mathsf{loc\text{-}wf} \\ \mathsf{allphnms}(\mathring{\mathit{espc}}) \subseteq \mathsf{rng}(\mathring{\mathit{eenv}}) \end{cases} \\
\forall \mathring{\mathit{eref}} \mapsto \mathring{\mathit{phnm}} \in \mathring{\mathit{eenv}} : \begin{cases} \mathring{\mathit{aenv}} \vdash \mathring{\mathit{eref}} \mapsto \mathring{\mathit{phnm}}\ \mathsf{loc\text{-}wf} \\ \mathring{\mathit{phnm}} \in \mathsf{allphnms}(\mathsf{locals}(\mathring{\mathit{eenv}})) \end{cases}
\end{array}$$

- $\boxed{\mathring{\mathit{aenv}} \vdash \mathring{\mathit{eref}} \mapsto \mathring{\mathit{phnm}}\ \mathsf{loc\text{-}wf}}$ Determine whether the physical name in the range of the mapping is indeed exported from its defining module, or from a local definition.

$$\frac{\mathring{\mathit{eref}} = \chi \text{ or } \mathsf{f}.\chi}{\mathring{\mathit{aenv}} \vdash \mathring{\mathit{eref}} \mapsto [\mathsf{f}]\chi\ \mathsf{loc\text{-}wf}}$$

- $\boxed{\mathring{\mathit{aenv}} \vdash \mathring{\mathit{espc}}\ \mathsf{loc\text{-}wf}}$ $\boxed{\mathring{\mathit{aenv}} \vdash \mathring{\mathit{espcs}}\ \mathsf{loc\text{-}wf}}$ Check that a locally available export spec is well formed with respect to a local definition and/or a spec in the context.

$$\frac{\mathring{\mathit{espc}}_0 = \mathsf{locmatch}(\mathring{\mathit{aenv}}; \mathring{\mathit{espc}}) \quad \mathring{\mathit{espc}}_1 = \mathsf{ctxmatch}(\mathring{\mathit{aenv}}; \mathring{\mathit{espc}}) \quad \mathring{\mathit{espc}}_0 \oplus \mathring{\mathit{espc}}_1 \leqslant \mathring{\mathit{espc}}}{\mathring{\mathit{aenv}} \vdash \mathring{\mathit{espc}}\ \mathsf{loc\text{-}wf}}$$

$$\frac{\mathring{\mathit{espc}}_0 = \mathsf{locmatch}(\mathring{\mathit{aenv}}; \mathring{\mathit{espc}}) \quad \mathsf{noctxmatch}(\mathring{\mathit{aenv}}; \mathring{\mathit{espc}}) \quad \mathring{\mathit{espc}}_0 \leqslant \mathring{\mathit{espc}}}{\mathring{\mathit{aenv}} \vdash \mathring{\mathit{espc}}\ \mathsf{loc\text{-}wf}}$$

$$\frac{\mathsf{nolocmatch}(\mathring{aenv};\mathring{espc}) \qquad \mathring{espc}_1 = \mathsf{ctxmatch}(\mathring{aenv};\mathring{espc}) \qquad \mathring{espc}_1 \leqslant \mathring{espc}}{\mathring{aenv} \vdash \mathring{espc}\ \mathsf{loc\text{-}wf}}$$

$$\mathring{aenv} \vdash \mathring{espcs}\ \mathsf{loc\text{-}wf} \quad \overset{\mathsf{def}}{\Leftrightarrow} \quad \forall \mathring{espc} \in \mathring{espcs}:\ \mathring{aenv} \vdash \mathring{espc}\ \mathsf{loc\text{-}wf}$$

- $\boxed{\mathsf{islocal}(\mathring{aenv};\mathring{espc})}$ $\boxed{\mathsf{islocal}(\mathring{aenv};\mathring{phnm})}$ Is the given entity's identity the same as the local module? Or, in the case of signatures, is this entity one of the locally specified ones? (This does *not* check for a matching definition/declaration; it merely looks at the identity.)

$$\begin{aligned}
&\mathsf{islocal}((fenv;\mathsf{f}_0;defs);\mathring{espc}) && \overset{\mathsf{def}}{\Leftrightarrow} && \mathsf{ident}(\mathring{espc}) = \mathsf{f}_0 \\
&\mathsf{islocal}((fenv;\mathsf{f}_0;defs);[\mathsf{f}]\chi) && \overset{\mathsf{def}}{\Leftrightarrow} && \mathsf{f} = \mathsf{f}_0
\end{aligned}$$

- $\boxed{\mathsf{locmatch}(\mathring{aenv};\mathring{espc})}$ $\boxed{\mathsf{ctxmatch}(\mathring{aenv};\mathring{espc})}$ $\boxed{\mathsf{nolocmatch}(\mathring{aenv};\mathring{espc})}$ $\boxed{\mathsf{noctxmatch}(\mathring{aenv};\mathring{espc})}$ Find a matching export spec in the context or the local environment.

$$\begin{aligned}
&\mathsf{locmatch}(\mathring{aenv};\mathring{espc}) && \overset{\mathsf{def}}{=} && \mathring{espc}' \quad \text{if} \begin{cases} \mathsf{islocal}(\mathring{aenv};\mathring{espc}), \\ \exists def \in \mathring{aenv}.defs:\ def \sqsubseteq \mathring{espc}', \\ \mathring{espc}' \ \oplus_? \ \mathring{espc} \end{cases} \\
&\mathsf{ctxmatch}(\mathring{aenv};\mathring{espc}) && \overset{\mathsf{def}}{=} && \mathring{espc}' \quad \text{if} \begin{cases} \mathring{espc}' \in \mathring{aenv}.fenv(\mathsf{ident}(\mathring{espc})), \\ \mathring{espc}' \ \oplus_? \ \mathring{espc} \end{cases} \\
&\mathsf{nolocmatch}(\mathring{aenv};\mathring{espc}) && \overset{\mathsf{def}}{\Leftrightarrow} && \mathsf{islocal}(\mathring{aenv};\mathring{espc}) \Rightarrow \\
&&&&& \quad \forall \mathring{espc}', def \in \mathring{aenv}.defs:\ def \sqsubseteq \mathring{espc}' \Rightarrow \mathring{espc}' \not\oplus_? \mathring{espc} \\
&\mathsf{noctxmatch}(\mathring{aenv};\mathring{espc}) && \overset{\mathsf{def}}{\Leftrightarrow} && \forall \mathring{espc}' \in \mathring{aenv}.fenv(\mathsf{ident}(\mathring{espc})) : \mathring{espc}' \not\oplus_? \mathring{espc}
\end{aligned}$$

A.2.2 *Elaboration definitions*

- $\boxed{\mathsf{mkstubs}(\Phi)}$ Converts a module context of all signatures into a set of IL stubs, *i.e.*, the elaboration of signatures in rule (ElabPkgSig).

$$\mathsf{mkstubs}(\overline{\nu{:}\tau^{-} @\, \omega}) \quad \overset{\mathsf{def}}{=} \quad \{\overline{\nu^{\star} \mapsto - : \tau^{\star} @\, \omega^{\star}}\}$$

B EXTERNAL LANGUAGE METATHEORY

B.1 HAUPTSÄTZE

Theorem A.1 (Regularity of typing). Assume $\Gamma = (\!| \Phi ; \mathcal{L} |\!)$, where applicable.

(1) If $\cdot \vdash \Gamma$ wf and $\Gamma; \nu_0 \vdash M : \tau @ \omega$ and $\Phi \oplus_? \nu_0{:}\tau^+ @ \omega$, then $\Phi \vdash \nu_0{:}\tau^+ @ \omega$ wf. (Proof discussed in §7.7.3.)
(2) If $\cdot \vdash \Gamma$ wf and $\Gamma; \rho \vdash S : \sigma @ \omega \mid \Phi_{sig}$ and $\Phi \oplus_? \Phi_{sig}$ and $\Phi \oplus_? \nu_0{:}\sigma^- @ \omega$, then $\Phi \vdash \Phi_{sig}$ wf and $\Phi \oplus \Phi_{sig} \vdash \nu_0{:}\sigma^- @ \omega$ wf. (Proof discussed in §7.7.3.)
(3) If $\vdash \Delta$ wf and $\cdot \vdash \Gamma$ wf and $\Delta; \Gamma; \hat{\Xi}_{pkg} \vdash B : \Xi$ and $\Gamma \oplus_? \Xi$, then $\Phi \vdash \Xi$ wf. (Proof in §8.7.2.)
(4) If $\vdash \Delta$ wf and $\Delta; \hat{\Xi}_{pkg} \vdash \overline{B} : \Xi$, then $\cdot \vdash \Xi$ wf. (Proof in §8.7.2.)
(5) If $\vdash \Delta$ wf and $\Delta \vdash D : \forall \overline{\alpha}.\Xi$, then $\cdot \vdash \Xi$ wf. (Proof in §8.7.2.)

Theorem A.2 (Elaboration Soundness). Assume $\Gamma = (\Phi; \mathcal{L})$, where applicable.

(1) If $\cdot \vdash \Gamma$ wf and $\Gamma; \nu_0 \vdash M : \tau @ \omega \rightsquigarrow hsmod$ and $\Phi \oplus_? \nu_0{:}\tau^+ @ \omega$, then $\Phi^\star \vdash hsmod : \tau^\star @ w'$ and $w' \sqsupseteq \omega^\star$.
(2) If $\vdash \Delta$ wf and $\cdot \vdash \Gamma$ wf and $\Delta; \Gamma; \hat{\Xi}_{pkg} \vdash B : \Xi \rightsquigarrow dexp$ and $\Gamma \oplus_? \Xi$, then $\Phi^\star \vdash dexp$ and $\Xi \sim dexp$.
(3) If $\vdash \Delta$ wf and $\Delta; \hat{\Xi}_{pkg} \vdash \overline{B} : \Xi \rightsquigarrow dexp$, then $\cdot \vdash dexp$ and $\Xi \sim dexp$.
(4) If $\vdash \Delta$ wf and $\Delta \vdash D : \forall \overline{\alpha}.\Xi \rightsquigarrow \lambda \overline{\alpha}.dexp$, then $\cdot \vdash dexp$ and $\Xi \sim dexp$.

Property A.3 (Package-level consistency in the EL). For all contexts Φ,

$$\forall \begin{pmatrix} \nu_1{:}\tau_1{}^{m_1} @ \omega_1 \\ \nu_2{:}\tau_2{}^{m_2} @ \omega_2 \end{pmatrix} \in \Phi : \quad \omega_1 \oplus_? \omega_2$$

This property is a direct corollary of the definition of (the partial commutative monoid on) the physical module context semantic object (Φ); see Figure 7.4.

B.2 STRUCTURAL PROPERTIES OF WELL-FORMEDNESS

Lemma A.4 (Weakening physical context preserves well-formedness). Suppose $\Phi \oplus_? \Phi_W$.

(1) If $\Phi; kenv \vdash_c typ :: knd$ then $\Phi \oplus \Phi_W; kenv \vdash_c typ :: knd$.
(2) If $\Phi \vdash dspc$ wf then $\Phi \oplus \Phi_W \vdash dspc$ wf.
(3) If $\Phi \vdash espc$ wf then $\Phi \oplus \Phi_W \vdash espc$ wf.

(4) If $\Phi \vdash \tau$ wf then $\Phi \oplus \Phi_W \vdash \tau$ wf.
(5) If $\Phi \vdash fact$ wf then $\Phi \oplus \Phi_W \vdash fact$ wf.
(6) If $\Phi \vdash \omega$ wf then $\Phi \oplus \Phi_W \vdash \omega$ wf.

(7) If $\Phi \vdash \Phi'$ specs-wf then $\Phi \oplus \Phi_W \vdash \Phi'$ specs-wf.
(8) If $\Phi \vdash \Phi'$ exports-wf then $\Phi \oplus \Phi_W \vdash \Phi'$ exports-wf.
(9) If $\Phi \vdash \Phi'$ imps-wf then $\Phi \oplus \Phi_W \vdash \Phi'$ imps-wf. (Proof uses antitonicity of depends [Property A.51] and Property A.62.)
(10) If $\Phi \vdash \Phi'$ wlds-wf then $\Phi \oplus \Phi_W \vdash \Phi'$ wlds-wf.
(11) If $\Phi \vdash \Phi'$ wf and $\Phi' \oplus_? \Phi_W$, then $\Phi \oplus \Phi_W \vdash \Phi'$ wf.

(12) If $\Phi \vdash \mathcal{L}$ wf then $\Phi \oplus \Phi_W \vdash \mathcal{L}$ wf.
(13) If $\Phi \vdash \Xi$ wf and $\Xi.\Phi \oplus_? \Phi_W$, then $\Phi \oplus \Phi_W \vdash \Xi$ wf.

Lemma A.5 (Merging preserves well-formedness)**.** Suppose $\Phi_1 \oplus_? \Phi_2$.

(1) If validspc($dspc_1; m_1$) and validspc($dspc_2; m_2$) and $dspc_1 \oplus_? dspc_2$, then validspc($dspc_1 \oplus dspc_2; m_1 \oplus m_2$).
(2) If $\Phi_1 \vdash dspc_1$ wf and $\Phi_2 \vdash dspc_2$ wf and $dspc_1 \oplus_? dspc_2$, then $\Phi_1 \oplus \Phi_2 \vdash dspc_1 \oplus dspc_2$ wf.
(3) If $\Phi_1 \vdash espc_1$ wf and $\Phi_2 \vdash espc_2$ wf and $espc_1 \oplus_? espc_2$, then $\Phi_1 \oplus \Phi_2 \vdash espc_1 \oplus espc_2$ wf.

(4) If $\Phi_1 \vdash \tau_1$ wf and $\Phi_2 \vdash \tau_2$ wf and $\tau_1 \oplus_? \tau_2$, then $\Phi_1 \oplus \Phi_2 \vdash \tau_1 \oplus \tau_2$ wf. (Proof uses merge and weakening on *dspc* and *espc*.)
(5) If $\Phi_1 \vdash \omega_1$ wf and $\Phi_2 \vdash \omega_2$ wf and $\omega_1 \oplus_? \omega_2$, then $\Phi_1 \oplus \Phi_2 \vdash \omega_1 \oplus \omega_2$ wf.

(6) If $\Phi_1 \vdash \Phi'_1$ specs-wf and $\Phi_2 \vdash \Phi'_2$ specs-wf and $\Phi'_1 \oplus_? \Phi'_2$, then $\Phi_1 \oplus \Phi_2 \vdash \Phi'_1 \oplus \Phi'_2$ specs-wf. (Proof uses merge and weakening on *dspc*.)
(7) If $\Phi_1 \vdash \Phi'_1$ exports-wf and $\Phi_2 \vdash \Phi'_2$ exports-wf and $\Phi'_1 \oplus_? \Phi'_2$, then $\Phi_1 \oplus \Phi_2 \vdash \Phi'_1 \oplus \Phi'_2$ exports-wf. (Proof uses merge and weakening on *espc*.)
(8) If $\Phi_1 \vdash \Phi'_1$ imps-wf and $\Phi_2 \vdash \Phi'_2$ imps-wf and $\Phi'_1 \oplus_? \Phi'_2$, then $\Phi_1 \oplus \Phi_2 \vdash \Phi'_1 \oplus \Phi'_2$ imps-wf. (Proof uses antitonicity of depends, Property A.51.)
(9) If $\Phi_1 \vdash \Phi'_1$ wlds-wf and$\Phi_2 \vdash \Phi'_2$ wlds-wf and $\Phi'_1 \oplus_? \Phi'_2$, then $\Phi_1 \oplus \Phi_2 \vdash \Phi'_1 \oplus \Phi'_2$ wlds-wf.
(10) If $\Phi_1 \vdash \Phi'_1$ wf and $\Phi_2 \vdash \Phi'_2$ wf and $\Phi'_1 \oplus_? \Phi'_2$ and $\Phi_1 \oplus_? \Phi'_2$ and $\Phi_2 \oplus_? \Phi'_1$, then $\Phi_1 \oplus \Phi_2 \vdash \Phi'_1 \oplus \Phi'_2$ wf. (Proof uses merge and weakening on *espc* and Property A.73.)

(11) If $\Phi_1 \vdash \mathcal{L}_1$ wf and $\Phi_2 \vdash \mathcal{L}_2$ wf and $\mathcal{L}_1 \oplus_? \mathcal{L}_2$, then $\Phi_1 \oplus \Phi_2 \vdash \mathcal{L}_1 \oplus \mathcal{L}_2$ wf. (Proof uses merge and weakening on τ.)
(12) If $\Phi_1 \vdash \Xi_1$ wf and $\Phi_2 \vdash \Xi_2$ wf and $\Xi_1.\Phi \oplus_? \Xi_2.\Phi$ and $\Phi_1 \oplus_? \Xi_2.\Phi$ and $\Phi_2 \oplus_? \Xi_1.\Phi$, then $\Phi_1 \oplus \Phi_2 \vdash \Xi_1 \oplus \Xi_2$ wf. (Proof uses merge on $\mathcal{L}$ and Φ.)

Lemma A.6 (Invariance of well-formedness under substitution)**.** Suppose $\cdot \vdash \Phi$ wf and apply($\theta; \Phi$) is defined.

(1) If $\Phi; kenv \vdash_c typ :: knd$ then $\theta\Phi; kenv \vdash_c \theta typ :: knd$.
(2) If $\Phi \vdash dspc$ wf then $\theta\Phi \vdash \theta dspc$ wf.
(3) If $\Phi \vdash espc$ wf then $\theta\Phi \vdash \theta espc$ wf.

(4) If $\Phi \vdash \tau$ wf then $\theta\Phi \vdash \theta\tau$ wf.
(5) If $\Phi \vdash fact$ wf then $\theta\Phi \vdash \theta fact$ wf.
(6) If $\Phi \vdash \omega$ wf and apply($\theta; \omega$) defined then $\theta\Phi \vdash \theta\omega$ wf.

(7) If $\Phi \vdash \Phi'$ specs-wf and apply($\theta; \Phi'$) defined then $\theta\Phi \vdash \theta\Phi'$ specs-wf. (Proof uses invariace and merge on *dspc*.)
(8) If $\Phi \vdash \Phi'$ exports-wf and apply($\theta; \Phi'$) defined then $\theta\Phi \vdash \theta\Phi'$ exports-wf. (Proof uses invariace and merge on *espc*.)
(9) If $\Phi \vdash \Phi'$ imps-wf and apply($\theta; \Phi'$) defined then $\theta\Phi \vdash \theta\Phi'$ imps-wf. (Proof uses Property A.52.)
(10) If $\Phi \vdash \Phi'$ wlds-wf and apply($\theta; \Phi'$) defined then $\theta\Phi \vdash \theta\Phi'$ wlds-wf.
(11) If $\Phi \vdash \Phi'$ wf and apply($\theta; \Phi'$) is defined and $\theta\Phi \oplus_? \theta\Phi'$, then $\theta\Phi \vdash \theta\Phi'$ wf. (Proof uses Lemma A.8 and invariance on exps/specs-wf.)

(12) If $\Phi \vdash \mathcal{L}$ wf then $\theta\Phi \vdash \theta\mathcal{L}$ wf. (Proof uses Lemma A.8.)

(13) If $\Phi \vdash \Xi$ wf and $\mathsf{apply}(\theta;\Xi.\Phi)$ is defined and $\theta\Phi \oplus_? \theta\Xi.\Phi$, then $\theta\Phi \vdash \theta\Xi$ wf. (Proof uses Lemma A.8 and invariance on $\mathcal{L}$ and Φ.)

Lemma A.7 (Preservation of well-formedness under context strengthening)**.** Suppose $N \subseteq \mathsf{dom}(\Phi)$.

(1) If $\Phi; \mathit{kenv} \vdash_c \mathit{typ} :: \mathit{knd}$ and $\mathsf{provs}(\mathit{typ}) \subseteq N$, then $\Phi|_N; \mathit{kenv} \vdash_c \mathit{typ} :: \mathit{knd}$.
(2) If $\Phi \vdash \mathit{dspc}$ wf and $\mathsf{provs}(\mathit{dspc}) \subseteq N$, then $\Phi|_N \vdash \mathit{dspc}$ wf.
(3) If $\Phi \vdash \mathit{espc}$ wf and $\mathsf{ident}(\mathit{espc}) \in N$, then $\Phi|_N \vdash \mathit{espc}$ wf.

(4) If $\Phi \vdash \tau$ wf and $\mathsf{provs}(\tau) \in N$, then $\Phi|_N \vdash \tau$ wf.
(5) If $\Phi \vdash \mathit{fact}$ wf and $\mathsf{ident}(\mathit{fact}) \in N$ then $\Phi|_N \vdash \mathit{fact}$ wf.
(6) If $\Phi \vdash \omega$ wf and $\mathsf{idents}(\omega) \subseteq N$ then $\Phi|_N \vdash \omega$ wf.

(7) If $\Phi \vdash \Phi$ specs-wf and $\mathsf{depends}_\Phi(N) \subseteq N$, then $\Phi|_N \vdash \Phi|_N$ specs-wf.
(8) If $\Phi \vdash \Phi$ exports-wf and $\mathsf{depends}_\Phi(N) \subseteq N$, then $\Phi|_N \vdash \Phi|_N$ exports-wf.
(9) If $\Phi \vdash \Phi$ imps-wf and $\mathsf{depends}_\Phi(N) \subseteq N$, then $\Phi|_N \vdash \Phi|_N$ imps-wf. (Proof uses Lemma A.53.)
(10) If $\Phi \vdash \Phi$ wlds-wf and $\mathsf{depends}_\Phi(N) \subseteq N$, then $\Phi|_N \vdash \Phi|_N$ wlds-wf.
(11) If $\cdot \vdash \Phi$ wf and $\mathsf{depends}_\Phi(N) \subseteq N$, then $\cdot \vdash \Phi|_N$ wf.

(12) If $\Phi \vdash \mathcal{L}$ wf and $\mathsf{depends}_\Phi(N) \subseteq N$, then $\Phi|_N \vdash \mathcal{L}|_N$ wf.
(13) If $\cdot \vdash (\Phi;\mathcal{L})$ wf and $\mathsf{depends}_\Phi(N) \subseteq N$, then $\cdot \vdash (\Phi;\mathcal{L})|_N$ wf.

Lemma A.8 (Cut on physical contexts and signatures)**.**

(1) If $\Phi \vdash \Phi_1$ wf and $\Phi \oplus \Phi_1 \vdash \Phi_2$ wf, then $\Phi_1 \oplus_? \Phi_2$ and $\Phi \vdash \Phi_1 \oplus \Phi_2$ wf. (Proof uses merge on exps/specs/imps-wf.)
(2) If $\cdot \vdash \Xi_1$ wf and $\Xi_1.\Phi \vdash \Xi_2$ wf and $\Xi_1 \oplus_? \Xi_2$, then $\cdot \vdash \Xi_1 \oplus \Xi_2$ wf. (Proof uses cut on physical contexts and merge on logical contexts.)

Lemma A.9 (Thinning preserves well-formedness)**.** If $\cdot \vdash \Xi$ wf and $\vdash \Xi \xrightarrow{t} \Xi'$ then $\cdot \vdash \Xi'$ wf and $\mathsf{depends}_{\Xi.\Phi}(N) \subseteq N$, where $N = \mathsf{dom}(\Xi'.\Phi)$.

Lemma A.10 (Well-formed contexts contain all modules' dependencies)**.** If $\cdot \vdash \Phi$ wf then $\mathsf{depends}_\Phi(\mathsf{dom}(\Phi)) \subseteq \mathsf{dom}(\Phi)$.

Lemma A.11 (Well-formed contexts contain all modules' dependencies)**.** If $\cdot \vdash \Phi'$ wf and $\nu{:}\tau^{-}@ \in \Phi'$, then $\Phi \vdash \tau$ wf. (Proof straightforward since this judgment checks the same properties as the context judgments.)

Lemma A.12 (Logical module renaming preserves well-formedness)**.** If $\mathsf{rename}(r;\Xi)$ is defined and $\Phi \vdash \Xi$ wf, then $\Phi \vdash \mathsf{rename}(r;\Xi)$ wf. (Proof immediate since $\Xi.\Phi$ and $\mathsf{dom}(\Xi.\mathcal{L}) arepreservedbyrenaming$.)

B.3 INVARIANTS OF AUXILIARY EL MODULE MACHINERY

B.3.1 *Axioms about core typing*

Axiom A.13 (Validity of module definition checking)**.**

If $\Phi; \nu_0; \mathit{eenv}; \omega \vdash \mathit{defs} : \mathit{dspcs}$

then
(a) $\mathit{defs} = \mathit{def}_1, \ldots, \mathit{def}_n \wedge \mathit{dspcs} = \mathit{dspc}_1, \ldots, \mathit{dspc}_n$
(b) $\mathsf{nooverlap}(\mathit{defs})$
(c) $\bigcup_{i \in [1..n]} \mathsf{provs}(\mathit{dspc}_i) \subseteq \mathsf{dom}(\Phi) \cup \{\nu_0\}$
(d) $\forall i \in [1..n] : \mathit{def}_i \sqsubseteq \mathit{dspc}_i$
(e) $\forall i \in [1..n]$ s.t. $\mathit{def}_i = (\texttt{instance} \ldots) : \mathit{eenv}(\mathsf{head}(\mathit{def}_i)) = \mathsf{head}(\mathit{dspc}_i)$

Axiom A.14 (Validity of signature declaration checking).

If $\Phi; \overline{\nu}; \mathit{eenv}; \omega \vdash \mathit{decls} : \mathit{dspcs}$

then
(a) $\mathit{decls} = \mathit{decl}_1, \ldots, \mathit{decl}_n \wedge \mathit{dspcs} = \mathit{dspc}_1, \ldots, \mathit{dspc}_n$
(b) $\mathsf{nooverlap}(\mathit{decls})$
(c) $\bigcup_{i \in [1..n]} \mathsf{provs}(\mathit{dspc}_i) \subseteq \mathsf{dom}(\Phi) \cup \{\overline{\nu}\}$
(d) $\forall i \in [1..n] : \mathit{decl}_i \sqsubseteq \mathit{dspc}_i$
(e) $\forall i \in [1..n]$ s.t. $\mathit{decl}_i = (\texttt{instance} \ldots) : \mathit{eenv}(\mathsf{head}(\mathit{decl}_i)) = \mathsf{head}(\mathit{dspc}_i)$

Axiom A.15 (Regularity of module definition checking).

If
(i) $\cdot \vdash \Phi$ wf
(ii) $\mathit{aenv} \Vdash \mathit{eenv}$ loc-wf
(iii) $\mathit{aenv}; \mathit{eenv} \Vdash \mathsf{shape}(\omega)$ loc-wf
(iv) $\Phi; \nu_0; \mathit{eenv}; \omega \vdash \mathit{defs} : \mathit{dspcs}$
(v) $\tau.\mathit{dspcs} = \mathit{dspcs}$
(vi) $\Phi \oplus_? \Phi_0$

then $\Phi \oplus \Phi_0 \vdash \tau^+$ spcs-wf,
where $\mathit{aenv} = (\mathsf{shape}(\Phi); \nu_0; \mathit{defs})$ and $\Phi_0 = \nu_0{:}\tau^+ @ \omega$.

Axiom A.16 (Regularity of signature declaration checking).

If
(i) $\cdot \vdash \Phi$ wf
(ii) $\mathit{aenv} \Vdash \mathit{eenv}$ loc-wf
(iii) $\mathit{aenv}; \mathit{eenv} \Vdash \mathsf{shape}(\omega)$ loc-wf
(iv) $\rho; \mathit{eenv} \vdash \mathit{decls} \rightsquigarrow \overline{\nu}$
(v) $\Phi; \overline{\nu}; \mathit{eenv}; \omega \vdash \mathit{decls} : \mathit{dspcs}$
(vi) $\Phi_{\mathsf{sig}} = \mathsf{sigenv}(\mathit{dspcs}; \mathit{espcs}; \omega)$
(vii) $\Phi \oplus_? \Phi_{\mathsf{sig}}$

then $\Phi \oplus \Phi_{\mathsf{sig}} \vdash {\Phi_{\mathsf{sig}}}^m$ spcs-wf
where $\mathit{aenv} = (\mathsf{shape}(\Phi); \rho; \mathit{decls})$.

B.3.2 *Core environment construction and import resolution*

Lemma A.17 (Soundness of core environment construction (EL)).
- If $\cdot \vdash \Gamma$ wf and $\Gamma \vdash \mathit{impdecls}; \mathit{rbnds} \rightsquigarrow \mathit{eenv} @ \omega$, then $\mathsf{haslocaleenv}(\mathit{eenv}; \mathit{rbnds})$ and $\mathit{aenv} \Vdash \mathit{eenv}$ loc-wf and $\mathit{aenv}; \mathit{eenv} \Vdash \mathsf{shape}(\omega)$ loc-wf, where $\mathit{aenv} = \mathsf{mkaenv}(\mathsf{shape}(\Gamma.\Phi); \mathit{rbnds})$.
- If $\cdot \Vdash \widehat{\Gamma}$ wf and $\widehat{\Gamma} \Vdash \mathit{impdecls}; \mathit{ibnds} \rightsquigarrow \mathit{eenv} @ \widehat{\omega}$, then $\mathsf{haslocaleenv}(\mathit{eenv}; \mathit{ibnds})$ and $\mathit{aenv} \Vdash \mathit{eenv}$ loc-wf and $\mathit{aenv}; \mathit{eenv} \Vdash \widehat{\omega}$ loc-wf, where $\mathit{aenv} = \mathsf{mkaenv}(\widehat{\Gamma}.\widehat{\Phi}; \mathit{ibnds})$.

Lemma A.18 (Soundness of import declaration resolution (EL)). If $\cdot \Vdash \widehat{\Gamma}$ wf and $\widehat{\Gamma} \Vdash \mathit{impdecl} \rightsquigarrow \mathit{eenv}$ and $\mathit{aenv}.\widehat{\Phi} = \widehat{\Gamma}.\widehat{\Phi}$, then $\mathit{aenv} \Vdash \mathit{eenv}$ loc-wf.

Lemma A.19 (Soundness of imported module specification resolution (EL)). If $\cdot \Vdash \widehat{\Gamma}$ wf and $\widehat{\Gamma}; \ell \Vdash \mathit{impspec} \rightsquigarrow \mathit{espcs}$ and $\mathit{aenv}.\widehat{\Phi} = \widehat{\Gamma}.\widehat{\Phi}$ and $\mathsf{nooverlap}(\mathit{aenv})$, then $\mathit{aenv} \Vdash \mathit{espcs}$ loc-wf.

Lemma A.20 (Soundness of imported module entity resolution (EL)). If $\cdot \Vdash \widehat{\Gamma}$ wf and $\widehat{\Gamma}; \ell \Vdash \mathit{import} \rightsquigarrow \mathit{espc}$ and $\mathit{aenv}.\widehat{\Phi} = \widehat{\Gamma}.\widehat{\Phi}$, then $\mathit{aenv} \Vdash \mathit{espc}$ loc-wf.

B.3.2.1 *Technical lemmas needed*

Property A.21 (Local entity environment is always well-formed).

- $(\widehat{\Phi}; \nu_0; \mathit{defs}) \Vdash \mathsf{mkloceenv}((\nu_0, \ldots, \nu_0 \mid \mathit{defs}))$ loc-wf
- $(\widehat{\Phi}; \rho; \mathit{decls}) \Vdash \mathsf{mkloceenv}((\rho \mid \mathit{decls}))$ loc-wf
- If $\omega = \mathsf{mklocworld}((\nu_0, \ldots, \nu_0 \mid \mathit{defs});\ \mathit{eenv})$ then, for any $\widehat{\Phi}$, $(\widehat{\Phi}; \nu_0; \mathit{defs}); \mathit{eenv} \Vdash \mathsf{shape}(\omega)$ loc-wf.
- If $\omega = \mathsf{mklocworld}((\rho \mid \mathit{decls});\ \mathit{eenv})$ then, for any $\widehat{\Phi}$, $(\widehat{\Phi}; \rho; \mathit{decls}); \mathit{eenv} \Vdash \mathsf{shape}(\omega)$ loc-wf.

Lemma A.22 (Qualification preserves entity env well-formedness)**.** If $\mathit{aenv} \Vdash \mathit{eenv}$ loc-wf then $\mathit{aenv} \Vdash \mathsf{qualify}(\mathit{eenv};\ \ell)$ loc-wf.

Lemma A.23 (Merging locally well-formed objects preserves local well-formedness)**.**

- If $\mathit{aenv} \Vdash \mathit{espc}_1$ loc-wf and $\mathit{aenv} \Vdash \mathit{espc}_2$ loc-wf and $\mathit{espc}_1 \oplus_? \mathit{espc}_2$, then $\mathit{aenv} \Vdash \mathit{espc}_1 \oplus \mathit{espc}_2$ loc-wf.
- If $\mathit{aenv} \Vdash \mathit{espcs}_1$ loc-wf and $\mathit{aenv} \Vdash \mathit{espcs}_2$ loc-wf and $\mathit{espcs}_1 \oplus_? \mathit{espcs}_2$, then $\mathit{aenv} \Vdash \mathit{espcs}_1 \oplus \mathit{espcs}_2$ loc-wf.
- If $\mathit{aenv} \Vdash \mathit{eenv}_1$ loc-wf and $\mathit{aenv} \Vdash \mathit{eenv}_2$ loc-wf and $\mathit{eenv}_1 \oplus_? \mathit{eenv}_2$, then $\mathit{aenv} \Vdash \mathit{eenv}_1 \oplus \mathit{eenv}_2$ loc-wf.
- If $\mathit{aenv}; \mathit{eenv} \Vdash \omega_1$ loc-wf and $\mathit{aenv}; \mathit{eenv} \Vdash \omega_2$ loc-wf and $\omega_1 \oplus_? \omega_2$, then $\mathit{aenv}; \mathit{eenv} \Vdash \omega_1 \oplus \omega_2$ loc-wf.

Lemma A.24.

- If $\mathit{aenv} \Vdash \mathit{espc}$ loc-wf then $\mathit{aenv} \Vdash \mathsf{mkeenv}(\mathit{espc})$ loc-wf.
- If $\mathit{aenv} \Vdash \mathit{espcs}$ loc-wf then $\mathit{aenv} \Vdash \mathsf{mkeenv}(\mathit{espcs})$ loc-wf.

Property A.25 (Domains of certain kinds of entity environments)**.**

- $\forall \mathit{eref} \in \mathsf{dom}(\mathsf{qualify}(m;\ \mathit{eenv})) : \mathit{eref} = m.\chi$ for some χ.
- $\forall \mathit{eref} \in \mathsf{dom}(\mathsf{mkeenv}(\mathit{espc})) : \mathit{eref} = \chi$ for some χ.
- $\forall \mathit{eref} \in \mathsf{dom}(\mathsf{mkeenv}(\mathit{espcs})) : \mathit{eref} = \chi$ for some χ.
- $\forall \mathit{eref} \in \mathsf{dom}(\mathsf{mkloceenv}(\mathit{rbnds})) : \mathit{eref} = \chi$ or $\mathtt{Local}.\chi$ for some χ.

Property A.26 (Distributivity of substitution over entity environment merging)**.** This is a corollary to Property A.67.

(1) If $\mathring{\mathit{eenv}}'_1 = \mathsf{apply}(\theta; \mathring{\mathit{eenv}}_1)$ is defined and if $\mathring{\mathit{eenv}}'_2 = \mathsf{apply}(\theta; \mathring{\mathit{eenv}}_2)$ is defined and $\mathring{\mathit{eenv}}_1 \oplus_? \mathring{\mathit{eenv}}_2$ and $\mathring{\mathit{eenv}}'_1 \oplus_? \mathring{\mathit{eenv}}'_2$, then $\mathsf{apply}(\theta; \mathring{\mathit{eenv}}_1 \oplus \mathring{\mathit{eenv}}_2)$ is defined and equals $\mathring{\mathit{eenv}}'_1 \oplus \mathring{\mathit{eenv}}'_2$.

(2) If $\forall i \in [1..n] : \mathsf{apply}(\theta; \mathring{\mathit{eenv}}_i)$ defined and $\forall i, j \in [1..n] : \mathring{\mathit{eenv}}_i \oplus_? \mathring{\mathit{eenv}}_j \wedge \mathsf{apply}(\theta; \mathring{\mathit{eenv}}_i) \oplus_? \mathsf{apply}(\theta; \mathring{\mathit{eenv}}_j)$ defined, then $\mathsf{apply}(\theta; \bigoplus_{i \in [1..n]} \mathring{\mathit{eenv}}_i)$ is defined and equals $\bigoplus_{i \in [1..n]} \mathsf{apply}(\theta; \mathring{\mathit{eenv}}_i)$.

B.3.3 *Export resolution*

Lemma A.27 (Soundness of module export resolution (EL))**.** If $\mathit{eenv} \Vdash \mathit{export} \rightsquigarrow \mathit{espc}$ and $\cdot \vdash \Phi$ wf and $\mathit{aenv} = (\mathsf{shape}(\Phi); \nu_0; \mathit{defs})$ and $\mathit{aenv} \Vdash \mathit{eenv}$ loc-wf and $\mathit{defs} \sqsubseteq \mathit{dspcs}$ and $\tau = \langle\!\!\langle\, \mathit{dspcs}\ ;\ \mathit{espcs}\ ;\ \mathrm{N}\, \rangle\!\!\rangle$ and $\Phi \oplus_? \nu_0{:}\tau^{+}@\,\omega$, then $\Phi \oplus \nu_0{:}\tau^{+}@\,\omega \vdash \tau$ exps-wf. (Proof by case analysis on export judgment. Requires Lemma A.33. Case (ExpLocal) goes by case analysis on eenv loc-wf and requires many technical lemmas about matching names in aenv to show the espcs are non-overlapping. Case (ExpList) requires Lemma A.32, Lemma A.5 on exps-wf, and Lemma A.35.)

Lemma A.28 (Soundness of signature environment exports in EL)**.** If $(\!|\, \Phi\ ;\ \mathcal{L}\, |\!); \rho \vdash S : \sigma\,@\,\omega \mid \Phi_{\mathrm{sig}}$ and $\cdot \vdash \Gamma.\Phi$ wf and $\Phi \oplus_? \Phi_{\mathrm{sig}}$, then $\Phi \oplus \Phi_{\mathrm{sig}} \vdash \Phi_{\mathrm{sig}}$ exps-wf. (Proof requires Axiom A.14; Lemma A.29; Lemma A.5 on exps-wf among $\Phi_i \in \Phi_{\mathrm{sig}}$; and Lemma A.4 on exps-wf.)

Lemma A.29 (Regularity of declaration exports in EL)**.** If $\rho; \mathit{eenv} \vdash \overline{\mathit{decls}} \rightsquigarrow \overline{\nu}$ then $|\overline{\mathit{decl}}| = |\overline{\mathit{espc}}|$ and $\overline{\mathit{decl} \sqsubseteq \mathit{espc}}$.

Lemma A.30 (Kinding is liftable)**.** If $\Phi^{\star}; \mathit{kenv} \vdash^{\mathrm{IL}}_{\mathsf{c}} \mathit{typ}^{\star} :: \mathit{knd}$ then $\Phi; \mathit{kenv} \vdash_{\mathsf{c}} \mathit{typ} :: \mathit{knd}$.

B.3.3.1 *Technical lemmas needed*

Lemma A.31 (Soundness of module export resolution (EL))**.** If $eenv \Vdash export \rightsquigarrow espc$ and $\Phi \vdash \Phi$ wf and $aenv = (\mathsf{shape}(\Phi); \nu_0; \overline{def})$ and $aenv \Vdash eenv$ loc-wf and $\mathsf{consistent}(aenv)$ and $\overline{def \sqsubseteq dspc}$ and $\tau = \langle\!|\, \overline{dspc}\,;\, \overline{espc}\,;\, \overline{\nu'} \,|\!\rangle$ and $\Phi \oplus_? \nu_0{:}\tau^+ @\, \omega$, then $\Phi \oplus \nu_0{:}\tau^- @\, \omega \vdash \tau$ exps-wf. *(Note the polarity of τ in the context of the conclusion.)*

Lemma A.32 (Lifting well-formedness of $espc$ in $aenv$ to Φ)**.** If $\Phi \vdash \Phi$ wf and $aenv = (\mathsf{shape}(\Phi); \nu_0; \overline{def})$ and $aenv \Vdash espc$ loc-wf and $\mathsf{consistent}(aenv)$ and $\mathsf{ident}(espc) = \nu_0$ and $espc \in \overline{espc}$ and $\overline{def \sqsubseteq dspc}$ and $\tau = \langle\!|\, \overline{dspc}\,;\, \overline{espc}\,;\, \overline{\nu'} \,|\!\rangle$ and $\Phi \oplus_? \nu_0{:}\tau^+ @\, \omega$, then $\Phi \oplus \nu_0{:}\tau^+ @\, \omega \vdash espc$ wf. (Used in proof of module export soundness for the local exports case.)

Lemma A.33 (Consistent augmented environments come from mergeable modules (EL))**.** If $\overline{def \sqsubseteq dspc}$ and $\overline{dspc} \in \tau$ and $\Phi \oplus_? \nu_0{:}\tau^+ @\, \omega$ and $\mathsf{nooverlap}(\overline{def})$, then $\mathsf{consistent}(aenv)$, where $aenv = (\Phi; \nu_0; defs)$. (Proof very straightforward; uses Lemma A.60.)

Lemma A.34 (Signature environment context well-formedness implies singleton well-formedness)**.** If $\cdot \vdash \Phi$ wf and $(\!|\, \Phi\,;\, \mathcal{L} \,|\!); \rho \vdash S : \sigma @\, \omega \mid \Phi_{\mathsf{sig}}$ and $\Phi \vdash \Phi_{\mathsf{sig}}$ wf and $\Phi \oplus_? \Phi_0$, then $\Phi \oplus \Phi_{\mathsf{sig}} \vdash \Phi_0$ wf, where $\Phi_0 = \nu_0{:}\sigma^- @\, \omega$. (Used at end of proof of sig regularity.)

Lemma A.35 (Export well-formedness is invariant to polarity in the context)**.** If $\overline{\nu{:}\tau^m @\, \omega} \vdash \Phi'$ exps-wf then, for any $\overline{m'}$ s.t. $|\overline{m'}| = |\overline{m}|$, $\overline{\nu{:}\tau^{m'} @\, \omega} \vdash \Phi'$ exps-wf.

Lemma A.36 (Well-formedness of signature world)**.** If $\cdot \vdash \Phi$ wf and $\mathsf{consistent}(\rho.\widehat{\omega}_0)$ and $\rho; eenv \vdash decls \rightsquigarrow \overline{\nu}$ and $\Phi_{\mathsf{sig}} = \mathsf{sigenv}(dspcs;\ \overline{\nu};\ \omega)$ and $aenv; eenv \Vdash \mathsf{shape}(\omega)$ loc-wf and $\Phi \oplus_? \Phi_{\mathsf{sig}}$, then $\Phi \oplus \Phi_{\mathsf{sig}} \vdash \omega$ wf, where $\Phi = \Gamma.\Phi$ and $aenv = (\mathsf{shape}(\Phi); \rho; decls)$. (Proof by case analysis of well-formedness of $fact \in \mathsf{shape}(\omega)$. In the case of the fact coming from the context, proof requires Lemma A.40 and weakening on fact well-formedness. In the case of the fact coming from a local declaration, proof requires inversion of the siganture provenance judgment, some algebraic manipulation of the context $\Phi' = \Phi \oplus \Phi_{\mathsf{sig}}$ and, critically, the world shape consistency premise to know that any two facts with the same head in $\rho.\widehat{\omega}_0$ must have the same identity.)

Lemma A.37 (Well-formedness of module world)**.** If $\Phi; \nu_0; eenv; \omega \vdash defs : dspcs$ and $dspcs \in \tau$ and $\Phi \oplus_? \nu_0{:}\tau^+ @\, \omega$ and $(\mathsf{shape}(\Phi); \nu_0; defs); eenv \Vdash \widehat{\omega}$ loc-wf and $\cdot \vdash \Phi$ wf, then $\Phi \oplus \nu_0{:}\tau^+ @\, \omega \vdash \omega$ wf, where $\Phi = \Gamma.\Phi$. (Proof analogous to that of Lemma A.36.)

Lemma A.38 (Well-formedness of singleton signature context)**.** If $\cdot \vdash \Phi$ wf and $\Gamma; \rho \vdash S : \sigma @\, \omega \mid \Phi_{\mathsf{sig}}$ and $\Phi \oplus_? \Phi_{\mathsf{sig}}$ and $\Phi \oplus_? \Phi_0$ and $\Phi \vdash \Phi_{\mathsf{sig}}$ wf, then $\Phi \oplus \Phi_{\mathsf{sig}} \vdash \Phi_0$ wf, where $\Phi_0 = \nu_0{:}\sigma^- @\, \omega$. (Proof resembles the first part of the soundness of typing proof, making critical use of weakening and merging of various well-formedness judgments.)

Property A.39 (Worlds created locally are well-formed in the local environment)**.**

If $\omega = \mathsf{mklocalworld}(\nu_0; defs; eenv)$ then, for any $\widehat{\Phi}$, $(\widehat{\Phi}; \nu_0; defs); eenv \Vdash \widehat{\omega}$ loc-wf.

If $\omega = \mathsf{mklocalworld}(\hat{\tau}_0; decls; eenv)$ then, for any $\widehat{\Phi}$, $(\widehat{\Phi}; \rho; decls); eenv \Vdash \widehat{\omega}$ loc-wf.

Lemma A.40 (Well-formed fact in shapey projection of context is well-formed in that context)**.** If $\Phi \vdash \Phi$ wf and $\mathsf{shape}(\Phi) \Vdash head \mapsto \nu$ wf, then $\Phi \vdash head \mapsto \nu$ wf

Property A.41 (Matching in augmented environments succeeds or it doesn't)**.**

- Either $\exists espc_0 : espc_0 = \mathsf{locmatch}(aenv; espc)$ or $\mathsf{nolocmatch}(aenv; espc)$, but not both.
- Either $\exists espc_1 : espc_1 = \mathsf{ctxmatch}(aenv; espc)$ or $\mathsf{noctxmatch}(aenv; espc)$, but not both.

Property A.42 (Matching in augmented environments is unique)**.**

- If $espc_0 = \mathsf{locmatch}(aenv; espc)$ then, $\forall espc_0' : espc_0' = \mathsf{locmatch}(aenv; espc)$, $espc_0' = espc_0$.
- If $espc_1 = \mathsf{ctxmatch}(aenv; espc)$ then, $\forall espc_1' : espc_1' = \mathsf{ctxmatch}(aenv; espc)$, $espc_1' = espc_1$.

Property A.43 (Matching in augmented environments is preserved under mergeability).

- If $espc_0 = \mathsf{locmatch}(aenv; espc)$ and $espc' \oplus_? espc$ then, $espc_0 = \mathsf{locmatch}(aenv; espc')$.
- If $espc_1 = \mathsf{ctxmatch}(aenv; espc)$ and $espc' \oplus_? espc$ then, $espc_1 = \mathsf{ctxmatch}(aenv; espc')$.
- If $\mathsf{nolocmatch}(aenv; espc)$ and $espc' \oplus_? espc$ then, $\mathsf{nolocmatch}(aenv; espc')$.
- If $\mathsf{noctxmatch}(aenv; espc)$ and $espc' \oplus_? espc$ then, $\mathsf{noctxmatch}(aenv; espc')$.

Lemma A.44 (In consistent definition environments, a context match implies a better local match). If $\cdot \Vdash \hat{\Phi}$ wf and $aenv^+ = (\hat{\Phi}; \nu_0; defs)$ and $\mathsf{consistent}(aenv^+)$ and $\mathsf{ident}(espc) = \nu_0$ and $espc_1 = \mathsf{ctxmatch}(aenv^+; espc)$, then $\exists espc_0 : espc_0 = \mathsf{locmatch}(aenv^+; espc)$ and $espc_0 \leqslant espc_1$.

Corollary A.45. If $\cdot \Vdash \hat{\Phi}$ wf and $aenv^+ = (\hat{\Phi}; \nu_0; defs)$ and $\mathsf{consistent}(aenv^+)$ and $espc_0 = \mathsf{locmatch}(aenv^+; espc)$ and $espc_1 = \mathsf{ctxmatch}(aenv^+; espc)$, then $espc_0 \leqslant espc_1$.

Corollary A.46. If $\cdot \Vdash \hat{\Phi}$ wf and $aenv^+ = (\hat{\Phi}; \nu_0; defs)$ and $\mathsf{consistent}(aenv^+)$ and $\mathsf{nolocmatch}(aenv^+; espc)$ and $espc_1 = \mathsf{ctxmatch}(aenv^+; espc)$, then $\mathsf{ident}(espc) \neq \nu_0$.

Definition A.47 (Consistent augmented local environments in EL). If ν_0 exists within $\hat{\Phi}$ then for each specification therein, there is an implementing definition among *defs*.

$$\mathsf{consistent}((\hat{\Phi}; \nu_0; defs)) \overset{\mathsf{def}}{\Leftrightarrow} \begin{cases} \mathsf{nooverlap}(defs), \\ \forall \hat{dspc} \in \hat{\Phi}(\nu_0) : \exists def \in defs, dspc' : \\ \quad def \sqsubseteq dspc' \wedge \mathsf{shape}(dspc') \leqslant \hat{dspc} \end{cases}$$

B.3.4 *Dependency invariants*

Lemma A.48 (Locally available entities come from dependencies of imports).
Suppose $\Gamma = (\Phi; \mathcal{L})$ and $\cdot \vdash \Gamma$ wf.

- If $\cdot \vdash \Gamma$ wf and $\mathsf{shape}(\Gamma); \ell \Vdash import \rightsquigarrow espc$, then $\mathsf{ident}(espc) \in \mathsf{depends}^+_\Phi(\mathcal{L}(\ell))$.
- If $\cdot \vdash \Gamma$ wf and $\mathsf{shape}(\Gamma); \ell \Vdash impspec \rightsquigarrow espcs$, then $\mathsf{ident}(espcs) \in \mathsf{depends}^+_\Phi(\mathcal{L}(\ell))$.
- If $\mathsf{shape}(\Gamma) \Vdash impdecl \rightsquigarrow eenv$, then
$$(\mathsf{ident}(\mathsf{locals}(eenv)) \cup \mathsf{provs}(\omega)) \subseteq \mathsf{depends}^+_\Phi(\mathcal{L}(\mathsf{imp}(impdecl))).$$
- If $\mathsf{shape}(\Gamma) \Vdash impdecls; ibnds \rightsquigarrow eenv \mathbin{@} \hat{\omega}$, then
$$(\mathsf{ident}(\mathsf{locals}(eenv)) \cup \mathsf{provs}(\omega)) \subseteq \left(\mathsf{provs}(ibnds) \cup \mathsf{depends}^+_\Phi(\mathcal{L}(\mathsf{imps}(impdecls)))\right).$$
- If $\Gamma \vdash impdecls; rbnds \rightsquigarrow eenv \mathbin{@} \omega$, then
$$(\mathsf{ident}(\mathsf{locals}(eenv)) \cup \mathsf{provs}(\omega)) \subseteq \left(\mathsf{provs}(rbnds) \cup \mathsf{depends}^+_\Phi(\mathcal{L}(\mathsf{imps}(impdecls)))\right).$$

Lemma A.49 (Well-typed modules preserve dependency invariants). If $\cdot \vdash \Gamma$ wf and $\Gamma; \nu_0 \vdash M : \tau \mathbin{@} \omega$ and $\Gamma.\Phi \oplus_? \nu_0{:}\tau^+ \mathbin{@} \omega$, then $\Gamma.\Phi \oplus \nu_0{:}\tau^+ \mathbin{@} \omega \vdash \nu_0{:}\tau^+ \mathbin{@} \omega$ imps-wf.

Property A.50. If $\nu \in \mathsf{depends}_\Phi(\nu_0)$ then $\mathsf{depends}_\Phi(\nu) \subseteq \mathsf{depends}_\Phi(\nu_0)$. (Proof relies on graph interpretation.)

Property A.51 (Antitonicity of dependencies in the context).

- If $\Phi \oplus_? \Phi_W$ and $\nu \in \mathsf{dom}(\Phi)$, then $\mathsf{depends}_\Phi(\nu) \subseteq \mathsf{depends}_{\Phi \oplus \Phi_W}(\nu)$.
- If $\Phi \oplus_? \Phi_W$ and $\nu \in \mathsf{dom}(\Phi)$, then $\mathsf{depends}_{\Phi;N}(\nu) \subseteq \mathsf{depends}_{\Phi \oplus \Phi_W;N}(\nu)$.
- If $\Phi \oplus_? \Phi_W$ and $\nu \in \mathsf{dom}(\Phi)$, then $\mathsf{depends}^+_\Phi(\nu) \subseteq \mathsf{depends}^+_{\Phi \oplus \Phi_W}(\nu)$.

Property A.52 (Preservation of dependencies under substitution). If $\nu \in \mathsf{depends}_\Phi(\nu_0)$ and $\Phi_\theta = \mathsf{apply}(\theta; \Phi)$ is defined, then $\theta\nu \in \mathsf{depends}_{\Phi_\theta}(\theta\nu_0)$. (Proof relies on correspondence of $\mathsf{depends}_\Phi(-)$ to Φ-graph reachability; θ as a graph homomorphism; and graph homomorphism's preservation of connectedness.)

Lemma A.53. If $\forall \nu \in \mathsf{dom}(\Phi|_N) : \mathsf{depends}_\Phi(\nu) \subseteq \mathsf{dom}(\Phi|_N)$, then $\forall \nu \in \mathsf{dom}(\Phi|_N) : \mathsf{depends}_\Phi(\nu) = \mathsf{depends}_{\Phi|_N}(\nu)$.

Property A.54 (A module's dependencies contain the provenances of its types)**.** If $\Phi(\nu) = \tau$ then $\mathsf{provs}(\tau) \subseteq \mathsf{depends}_\Phi(\nu)$.

B.4 ALGEBRAIC PROPERTIES OF CLASSIFIERS

B.4.1 *Package signatures*

Property A.55. Package signatures (Ξ) form an idempotent partial commutative monoid.

- *Idempotence*: $\Xi \oplus_? \Xi$ and $\Xi \oplus \Xi = \Xi$.
- *Commutativity*: If $\Xi_1 \oplus_? \Xi_2$, then $\Xi_2 \oplus_? \Xi_1$ and $\Xi_1 \oplus \Xi_2 = \Xi_2 \oplus \Xi_1$.
- *Associativity*: If $\Xi_1 \oplus_? \Xi_2$ and $\Xi_1 \oplus \Xi_2 \oplus_? \Xi_3$, then $\Xi_2 \oplus_? \Xi_3$ and $\Xi_1 \oplus_? \Xi_2 \oplus \Xi_3$ and $(\Xi_1 \oplus \Xi_2) \oplus \Xi_3 = \Xi_1 \oplus (\Xi_2 \oplus \Xi_3)$.
- *Identity*: $(\cdot;\cdot) \oplus_? \Xi$ and $(\cdot;\cdot) \oplus \Xi = \Xi$.

B.4.2 *Logical module contexts*

Property A.56. Logical module contexts ($\mathcal{L}$) form an idempotent partial commutative monoid with identity element $(\cdot)$.

B.4.3 *Physical module contexts*

Property A.57. Physical module contexts (Φ) form an idempotent partial commutative monoid with identity element $(\cdot)$.

Property A.58 (Substitution distributes over context merging)**.**

(1) If $\Phi_1' = \mathsf{apply}(\theta;\Phi_1)$ is defined and if $\Phi_2' = \mathsf{apply}(\theta;\Phi_2)$ is defined and $\Phi_1 \oplus_? \Phi_2$ and $\Phi_1' \oplus_? \Phi_2'$, then $\mathsf{apply}(\theta;\Phi_1 \oplus \Phi_2)$ is defined and equals $\Phi_1' \oplus \Phi_2'$.

(2) If $\forall i \in [1..n] : \mathsf{apply}(\theta;\Phi_i)$ defined and $\forall i,j \in [1..n] : \Phi_i \oplus_? \Phi_j \wedge \mathsf{apply}(\theta;\Phi_i) \oplus_? \mathsf{apply}(\theta;\Phi_j)$, then $\mathsf{apply}(\theta;\bigoplus_{i\in[1..n]} \Phi_i)$ is defined and equals $\bigoplus_{i\in[1..n]} \mathsf{apply}(\theta;\Phi_i)$.

B.4.4 *Polarized module types*

Property A.59. Polarized module types (τ^m) form an idempotent partial commutative monoid with identity element $\langle\!| \cdot \,;\, \cdot \,;\, \cdot |\!\rangle^-$.

Lemma A.60.

- If $\tau^m = \tau_1^{m_1} \oplus \tau_2^{m_2}$ then $m = m_1 \oplus m_2$ and $\tau = \tau_1 \oplus \tau_2$.
- If $\tau^m @ \omega = (\tau_1^{m_1} @ \omega_1) \oplus (\tau_2^{m_2} @ \omega_2)$ then $m = m_1 \oplus m_2$ and $\tau = \tau_1 \oplus \tau_2$ and $\omega = \omega_1 \oplus \omega_2$.
- If $\tau_1^{m_1} \oplus_? \tau_2^{m_2}$ and $m_1 = +$ then $\tau_1^{m_1} \oplus \tau_2^{m_2} = \tau_1^{m_1}$.
- If $(\tau_1^{m_1} @ \omega_1) \oplus_? (\tau_2^{m_2} @ \omega_2)$ and $m_1 = +$ then $(\tau_1^{m_1} @ \omega_1) \oplus (\tau_2^{m_2} @ \omega_2) = \tau_1^{m_1} @ \omega_1$.

B.4.5 *Module types*

Property A.61. Module types (τ) form an idempotent partial commutative monoid with identity element $\langle\!| \cdot \,;\, \cdot \,;\, \cdot |\!\rangle$.

Property A.62. If $\tau = \tau_1 \oplus \tau_2$ then $\mathsf{provs}(\tau) = \mathsf{provs}(\tau_1) \cup \mathsf{provs}(\tau_2)$.

B.4.6 *Entity specifications*

Property A.63. Sets of entity specifications (*dspcs*) form an idempotent partial commutative monoid with identity element $(\cdot)$.

Property A.64. Merging of two entity specifications ($dspc_1 \oplus dspc_2$) is idempotent, associative, and commutative.

B.4.7 *Export specifications*

Property A.65. Merging of two export specifications ($espc_1 \oplus espc_2$) is idempotent, associative, and commutative.

Property A.66. Sets of export specifications (*espcs*) form an idempotent partial commutative monoid with identity element $(\cdot)$.

Property A.67 (Substitution distributes over *espc*-set merging)**.**

(1) If $espcs'_1 = \mathsf{apply}(\theta; espcs_1)$ is defined and if $espcs'_2 = \mathsf{apply}(\theta; espcs_2)$ is defined and $espcs_1 \oplus_? espcs_2$ and $espcs'_1 \oplus_? espcs'_2$, then $\mathsf{apply}(\theta; espcs_1 \oplus espcs_2)$ is defined and equals $espcs'_1 \oplus espcs'_2$.

(2) If $\forall i \in [1..n] : \mathsf{apply}(\theta; espcs_i)$ defined and $\forall i, j \in [1..n] : espcs_i \oplus_? espcs_j \wedge \mathsf{apply}(\theta; espcs_i) \oplus_? \mathsf{apply}(\theta; espcs_j)$, then $\mathsf{apply}(\theta; \bigoplus_{i \in [1..n]} espcs_i)$ is defined and equals $\bigoplus_{i \in [1..n]} \mathsf{apply}(\theta; espcs_i)$. (Proof uses previous part, Property A.74.)

Property A.68 (Substitution on non-overlapping *espc*-sets is always defined)**.** If $\mathsf{nooverlap}(espcs)$ then $\mathsf{apply}(\theta; espcs)$ is defined. We often write $\theta espcs$ instead of $\mathsf{apply}(\theta; espcs)$ for non-overlapping *espcs*.

Property A.69 (Membership in a merging of *espc*-sets entails similarity to some *espc*)**.**

(1) If $espc \in espcs_1 \oplus espcs_2$ then $\exists i \in \{1, 2\}, espc_i \in espcs_i : espc \oplus_? espc_i$.

(2) If $espc \in \bigoplus_{i \in [1..n]} espcs_i$ then $\exists i \in [1..n], espc_i \in espcs_i : espc \oplus_? espc_i$.

Property A.70 (Merge produces a greatest lower bound on *espc*)**.**

(1) If $espc \leqslant espc_1$ and $espc \leqslant espc_2$ then $espc \leqslant espc_1 \oplus espc_2$.

(2) If $\forall i \in [1..n] : espc \leqslant espc_i$ then $espc \leqslant \bigoplus_{i \in [1..n]} espc_i$.

B.4.8 *Extra properties of semantic objects*

Lemma A.71 (Name disjointedness of two merged sets of uniquely-named entities)**.**

- If $\overline{dspc_1} \oplus_? \overline{dspc_2}$ and $\mathsf{nooverlap}(\overline{dspc_1})$ and $\mathsf{nooverlap}(\overline{dspc_2})$, then $\mathsf{nooverlap}(\overline{dspc_1} \oplus \overline{dspc_2}) \Leftrightarrow \forall \chi_1{:}dspc_1{\in}\overline{dspc_1}, \chi_2{:}dspc_2{\in}\overline{dspc_2}$ s.t. $\chi_1 \neq \chi_2 : \mathsf{allnames}(dspc_1) \mathbin{\#} \mathsf{allnames}(dspc_2)$.
- Likewise but with *espc* everywhere instead of *dspc*.

Lemma A.72 (Name disjointness of three merged sets of uniquely-named entities)**.**

- If $\mathsf{nooverlap}(\overline{dspc_1} \oplus \overline{dspc_2})$ and $\mathsf{nooverlap}(\overline{dspc_2} \oplus \overline{dspc_3})$ and $\mathsf{nooverlap}(\overline{dspc_2} \oplus \overline{dspc_3})$ and $\forall i \in \{1, 2, 3\} : \mathsf{nooverlap}(\overline{dspc_i})$, then $\mathsf{nooverlap}(\overline{dspc_1} \oplus (\overline{dspc_2} \oplus \overline{dspc_3}))$.
- Likewise but with *espc* everywhere instead of *dspc*.

Property A.73 (Partiality of multiple merges)**.**

- If $dspc_1 \oplus_? dspc_2$ and $dspc_2 \oplus_? dspc_3$ and $dspc_1 \oplus_? dspc_3$, then $dspc_1 \oplus_? (dspc_2 \oplus dspc_3)$.
- If $espc_1 \oplus_? espc_2$ and $espc_2 \oplus_? espc_3$ and $espc_1 \oplus_? espc_3$, then $espc_1 \oplus_? (espc_2 \oplus espc_3)$.
- If $dspcs_1 \oplus_? dspcs_2$ and $dspcs_2 \oplus_? dspcs_3$ and $dspcs_1 \oplus_? dspcs_3$, then $dspcs_1 \oplus_? (dspcs_2 \oplus dspcs_3)$.
- If $espcs_1 \oplus_? espcs_2$ and $espcs_2 \oplus_? espcs_3$ and $espcs_1 \oplus_? espcs_3$, then $espcs_1 \oplus_? (espcs_2 \oplus espcs_3)$.

- If $\tau_1 \;\oplus_?\; \tau_2$ and $\tau_2 \;\oplus_?\; \tau_3$ and $\tau_1 \;\oplus_?\; \tau_3$, then $\tau_1 \;\oplus_?\; (\tau_2 \;\oplus\; \tau_3)$. (Proof also uses Lemma A.72.)
- If $\tau_1^{m_1} \;\oplus_?\; \tau_2^{m_2}$ and $\tau_2^{m_2} \;\oplus_?\; \tau_3^{m_3}$ and $\tau_1^{m_1} \;\oplus_?\; \tau_3^{m_3}$, then $\tau_1^{m_1} \;\oplus_?\; (\tau_2^{m_2} \;\oplus\; \tau_3^{m_3})$.
- If $\Phi_1 \;\oplus_?\; \Phi_2$ and $\Phi_2 \;\oplus_?\; \Phi_3$ and $\Phi_1 \;\oplus_?\; \Phi_3$, then $\Phi_1 \;\oplus_?\; (\Phi_2 \;\oplus\; \Phi_3)$.

Property A.74 (Partiality of multiple merges). If $\bigoplus_{i\in[1..n]} A_i$ defined and $\forall i \in [1..n] : A_i \;\oplus_?\; A'$, then $\bigoplus_{i\in[1..n]} A_i \;\oplus_?\; A'$, where A ranges over all syntactic categories in previous statement.

Property A.75 (Preservation of entity membership by substitution).

- If $\mathsf{apply}(\theta; \Phi)$ and $\mathit{dspc} \in \Phi(\nu)$, then $\exists \mathit{dspc}' \in \mathsf{apply}(\theta; \Phi)(\theta\nu) : \mathit{dspc}' \leqslant \theta\mathit{dspc}$. Likewise for *espc*.
- If $\mathsf{apply}(\theta; \mathit{espcs})$ and $\mathit{espc} \in \mathit{espcs}$, then $\exists \mathit{espc}' \in \mathsf{apply}(\theta; \mathit{espcs}) : \mathit{espc}' \leqslant \theta\mathit{espc}$.

Property A.76 (Structure of sums of environments and types).

- If $\bigoplus_{i\in[1..n]} \Phi_i$ is fully defined and $\nu{:}\tau^m@ \in (\bigoplus_{i\in[1..n]} \Phi_i)$ and $I = \{i \in [1..n] \mid \nu \in \mathsf{dom}(\Phi_i)\}$ and $\{\tau_i^{m_i}\}_{i\in I} = \{(\tau')^{m'} \mid \nu{:}(\tau')^{m'} \in \Phi_i\}_{i\in I}$, then $\tau^m = \bigoplus_{i\in I} (\tau_i^{m_i})$.
- If $\bigoplus_{i\in[1..n]} \tau_i$ is fully defined and $\chi{:}\mathit{dspc} \in (\bigoplus_{i\in[1..n]} \tau_i)$ and $I = \{i \in [1..n] \mid \exists \mathit{dspc}' : \chi{:}\mathit{dspc}' \in \tau_i\}$ and $\{\mathit{dspc}_i\}_{i\in I} = \{\mathit{dspc}' \mid \chi{:}\mathit{dspc}' \in \tau_i\}_{i\in I}$, then $\mathit{dspc} = \bigoplus_{i\in I} \mathit{dspc}_i$.
- Likewise but with *espc* everywhere instead of *dspc*.

Property A.77.

- If $\mathit{espc}_1 \;\oplus_?\; \mathit{espc}_2$ then $\mathsf{allphnms}(\mathit{espc}_1 \;\oplus\; \mathit{espc}_2) = \mathsf{allphnms}(\mathit{espc}_1) \cup \mathsf{allphnms}(\mathit{espc}_2)$.
- If $\overline{\mathit{espc}_1} \;\oplus_?\; \overline{\mathit{espc}_2}$ then $\mathsf{allphnms}(\overline{\mathit{espc}_1} \;\oplus\; \overline{\mathit{espc}_2}) = \mathsf{allphnms}(\overline{\mathit{espc}_1}) \cup \mathsf{allphnms}(\overline{\mathit{espc}_2})$.

Lemma A.78. If $\hat{\mathit{dspc}} \sqsubseteq \mathit{espc}$ and $\mathsf{shape}(\mathit{dspc}) \leqslant \hat{\mathit{dspc}}$ then $\exists \mathit{espc}' :\; \mathit{dspc} \sqsubseteq \mathit{espc}' \leqslant \mathit{espc}$.

Lemma A.79.

- If $\mathit{dspc}_1 \in \Phi_1(\nu)$ and $\Phi_1 \;\oplus_?\; \Phi_2$ then $\exists \mathit{dspc}_2 : \mathit{dspc}_1 \;\oplus\; \mathit{dspc}_2 \in (\Phi_1 \;\oplus\; \Phi_2)(\nu)$.
- Likewise for *espc*.

Lemma A.80. If $\mathit{dspc}_1 \sqsubseteq \mathit{espc}_2$ and $\mathit{dspc}_1 \;\oplus_?\; \mathit{dspc}_2$ then $\exists \mathit{espc}_2 : \mathit{dspc}_2 \sqsubseteq \mathit{espc}_2$.

Property A.81 (Monotonicity of spec matching).

(1) If $\mathit{dspc}_1 \sqsubseteq \mathit{espc}_1$ and $\mathit{dspc}_2 \sqsubseteq \mathit{espc}_2$ and $\mathit{dspc}_1 \;\oplus_?\; \mathit{dspc}_2$, then $\mathit{espc}_1 \;\oplus_?\; \mathit{espc}_2$ and $\mathit{dspc}_1 \;\oplus\; \mathit{dspc}_2 \sqsubseteq \mathit{espc}_1 \;\oplus\; \mathit{espc}_2$.
(2) If $\mathit{dspc} = \bigoplus_{i\in I} \mathit{dspc}_i$ and $\mathit{espc} = \bigoplus_{i\in I} \mathit{espc}_i$ and $\forall i \in I :\; \mathit{dspc}_i \sqsubseteq \mathit{espc}_i$, then $\mathit{dspc} \sqsubseteq \mathit{espc}$.

Property A.82 (Monotonicity of export specs).

(1) If $\mathit{espc}_1 \leqslant \mathit{espc}'_1$ and $\mathit{espc}_2 \leqslant \mathit{espc}'_2$ and $\mathit{espc}_1 \;\oplus_?\; \mathit{espc}_2$, then $\mathit{espc}'_1 \;\oplus_?\; \mathit{espc}'_2$ and $\mathit{espc}_1 \;\oplus\; \mathit{espc}_2 \leqslant \mathit{espc}'_1 \;\oplus\; \mathit{espc}'_2$.
(2) If $\mathit{espc} = \bigoplus_{i\in I} \mathit{espc}_i$ and $\mathit{espc}' = \bigoplus_{i\in I} \mathit{espc}'_i$ and $\forall i \in I :\; \mathit{espc}_i \leqslant \mathit{espc}'_i$, then $\mathit{espc} \leqslant \mathit{espc}'$.

Lemma A.83. If $\Phi \;\oplus_?\; \nu{:}\tau^+$ then $\Phi \;\oplus_?\; \nu{:}\tau^m$ and $(\Phi \;\oplus\; \nu{:}\tau^m)(\nu) = \tau$.

Property A.84 (Substitution on singleton maps indexed by identities).

- If $\Phi = \nu{:}\tau^m@$ then $\mathsf{apply}(\theta; \Phi)$ is defined and equals $(\theta\nu){:}(\theta\tau)^m@$.
- If $\mathit{espcs} = (\mathit{espc})$ then $\mathsf{apply}(\theta; \mathit{espcs})$ is defined and equals $(\theta\mathit{espc})$.

(All proofs straightforward by the definition of substitution in terms of merging (over a singleton index set).)

Property A.85 (Definedness of substitution over smaller worlds).
If $\omega_1 \sqsupseteq \omega_2$ and $\mathsf{apply}(\theta; \omega_1)$ is defined then $\mathsf{apply}(\theta; \omega_2)$ is defined and $\theta\omega_1 \sqsupseteq \theta\omega_2$. And likewise for IL.

Property A.86 (Worlds from substituted contexts). Suppose $\mathsf{apply}(\theta; \Phi)$ is defined.

(1) If $\mathit{fact} \in \mathsf{world}_\Phi(\nu)$ then $\theta\mathit{fact} \in \mathsf{world}_{\theta\Phi}(\theta\nu)$.

(2) If $\omega = \mathsf{world}_{\Phi}(\nu)$ defined then $\mathsf{apply}(\theta;\omega)$ defined and $\omega' = \mathsf{world}_{\theta\Phi}(\theta\nu)$ defined and $\theta\omega \sqsubseteq \omega'$.

(3) If $\omega = \mathsf{world}_{\Phi}(N)$ defined and $\omega' = \mathsf{world}_{\theta\Phi}(\theta N)$ defined, then $\mathsf{apply}(\theta;\omega)$ defined and $\theta\omega \sqsubseteq \omega'$.

(4) If *fact* $\in \mathsf{world}^{+}_{\Phi}(\nu)$ then θ*fact* $\in \mathsf{world}^{+}_{\theta\Phi}(\theta\nu)$.

(5) If $\omega = \mathsf{world}^{+}_{\Phi}(N)$ defined and $\omega' = \mathsf{world}^{+}_{\theta\Phi}(\theta N)$ defined, then $\mathsf{apply}(\theta;\omega)$ defined and $\theta\omega \sqsubseteq \omega'$. (Only used in the IL.)

And likewise for IL.

C INTERNAL LANGUAGE METATHEORY

C.1 AXIOMS ABOUT CORE-LEVEL TYPING

Axiom A.87 (Validity of definition checking in IL)**.** Just like Axiom A.13 but for the IL judgment.

If $\quad \mathit{fenv};\, \mathsf{f}_0;\, \mathring{\mathit{eenv}};\, \mathring{\omega} \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$

then

(a) $\mathit{defs} = \mathit{def}_1, \ldots, \mathit{def}_n \wedge \mathring{\mathit{dspcs}} = \mathring{\mathit{dspc}}_1, \ldots, \mathring{\mathit{dspc}}_n$
(b) $\mathsf{nooverlap}(\mathit{defs})$
(c) $\bigcup_{i \in [1..n]} \mathsf{provs}(\mathring{\mathit{dspc}}_i) \subseteq \mathsf{dom}(\mathit{fenv}) \cup \{\mathsf{f}_0\}$
(d) $\forall i \in [1..n] : \mathit{def}_i \sqsubseteq \mathring{\mathit{dspc}}_i$
(e) $\forall i \in [1..n]$ s.t. $\mathit{def}_i = (\texttt{instance} \ldots) : \mathring{\mathit{eenv}}(\mathsf{head}(\mathit{def}_i)) = \mathsf{head}(\mathring{\mathit{dspc}}_i)$

Axiom A.88 (Regularity of module definition checking in IL)**.** Just like Axiom A.15 but for the IL judgment.

If

(i) $\cdot \vdash \mathit{fenv}$ wf
(ii) $\mathring{\mathit{aenv}} \vdash \mathring{\mathit{eenv}}$ loc-wf
(iii) $\mathring{\mathit{aenv}}; \mathring{\mathit{eenv}} \vdash \mathring{\omega}$ loc-wf
(iv) $\mathit{fenv};\, \mathsf{f}_0;\, \mathring{\mathit{eenv}};\, \mathring{\omega} \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$
(v) $\mathit{ftyp}.\mathring{\mathit{dspcs}} = \mathring{\mathit{dspcs}}$
(vi) $\mathit{fenv} \oplus_? \mathit{fenv}_0$

then $\quad \mathit{fenv} \oplus \mathit{fenv}_0 \vdash \mathit{ftyp}^+$ spcs-wf,
where $\mathring{\mathit{aenv}} = (\mathit{fenv}; \mathsf{f}_0; \mathit{defs})$ and $\mathit{fenv}_0 = \mathsf{f}_0{:}\mathit{ftyp}^+ @ \mathring{\omega}$.

Axiom A.89 (Weakening of core definition checking in IL)**.** Extending the file environment does not change the semantics of Haskell typechecking. This is used in the proof of weakening for module typing.

If
- $\mathit{fenv};\, \mathsf{f}_0;\, \mathring{\mathit{eenv}};\, \mathring{\omega} \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$
- $\mathit{fenv} \oplus_? \mathit{fenv}_W$

then $\quad \mathit{fenv} \oplus \mathit{fenv}_W;\, \mathsf{f}_0;\, \mathring{\mathit{eenv}};\, \mathring{\omega} \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$

Axiom A.90 (Monotonicity of core definition checking in IL)**.** Adding additional, non-overlapping instances to a Haskell program does not change the semantics of Haskell typechecking. This is used in the proof of weakening for module typing.

If
- $\mathit{fenv};\, \mathsf{f}_0;\, \mathring{\mathit{eenv}};\, \mathring{\omega} \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$
- $\mathring{\omega}' \sqsupseteq \mathring{\omega}$

then $\quad \mathit{fenv};\, \mathsf{f}_0;\, \mathring{\mathit{eenv}};\, \mathring{\omega}' \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$

Axiom A.91 (Module context strengthening for typing of core bindings in IL).

If
- $\cdot \vdash \mathit{fenv}\ \mathsf{wf}$
- $\mathit{fenv};\ \mathsf{f}_0;\ \mathring{\mathit{eenv}};\ \mathring{\omega} \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$
- $\mathsf{F} \supseteq \bigcup_{\mathring{\mathit{espc}} \in \mathring{\mathit{eenv}}} \mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{ident}(\mathring{\mathit{espc}}))$
- $\mathsf{F} \supseteq \mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{provs}(\mathring{\omega}))$

then $(\mathit{fenv}|_{\mathsf{F}});\ \mathsf{f}_0;\ \mathring{\mathit{eenv}};\ \mathring{\omega} \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$

Axiom A.92 (Invariance under substitution of core definition checking in IL).

If
- $\mathit{fenv} \vdash \mathit{fenv}\ \mathsf{wf}$
- $\mathsf{apply}(\theta;\mathit{fenv})$ defined
- $\mathit{fenv};\ \mathsf{f}_0;\ \mathring{\mathit{eenv}};\ \mathring{\omega} \vdash \mathring{\mathit{defs}} : \mathring{\mathit{dspcs}}$
- $\mathsf{avoidaliases}(\theta;\mathring{\mathit{eenv}})$

then $\mathsf{apply}(\theta;\mathit{fenv});\ \theta\mathsf{f}_0;\ \mathsf{apply}(\theta;\mathring{\mathit{eenv}});\ \theta\mathring{\mathit{defs}} \vdash \theta\mathring{\mathit{dspcs}} : \mathring{\mathit{dspcs}}$

C.2 JUDGMENTAL PROPERTIES OF IL TERMS

C.2.1 *Weakening*

Lemma A.93 (Weakening of IL typing judgments). Suppose $\mathit{fenv} \oplus_? \mathit{fenv}'$.

(1) *If*
- $\mathit{fenv} \vdash \mathit{hsmod} : \mathit{ftyp} \,@\, \mathring{\omega}$
- $\omega' = \mathsf{modworld}_{\mathit{fenv} \oplus \mathit{fenv}'}(\mathit{hsmod};\mathit{ftyp})$ defined

then
- $\mathit{fenv} \oplus \mathit{fenv}' \vdash \mathit{hsmod} : \mathit{ftyp} \,@\, \omega'$
- $\omega \sqsubseteq \omega'$

(2) *If*
- $\mathit{fenv};\ \mathsf{f}_0 \vdash \mathit{tfexp} \,@\, \mathring{\omega}$
- $\mathsf{modworld}_{\mathit{fenv} \oplus \mathit{fenv}'}(\mathsf{f}_0;\mathit{tfexp})$ defined

then $\mathit{fenv} \oplus \mathit{fenv}';\ \mathsf{f}_0 \vdash \mathit{tfexp} \,@\, \mathring{\omega}$

(3) *If*
- $\mathit{fenv} \vdash \mathit{dexp}$
- $\mathit{fenv}' \oplus_? \lfloor \mathit{dexp} \rfloor$
- $\forall \mathsf{f} \mapsto (\mathit{hsmod} : \mathit{ftyp}) \,@\, \omega \in \mathit{dexp} : \mathsf{modworld}_{\mathit{fenv} \oplus \mathit{fenv}' \oplus \lfloor \mathit{dexp} \rfloor}(\mathit{hsmod};\mathit{ftyp})$ defined

then $\mathit{fenv} \oplus \mathit{fenv}' \vdash \mathit{dexp}$

Weakening the file environment does not change the types of terms; rather, it leads to potentially extended worlds in which those terms are judged well-typed. Of particular interest is part (2): because the world of a typed file expression is "annotated," we can be sure that weakening the file environment does not actually change the world of the expression. Used in the proof of cut on directory expression typing.

Lemma A.94 (Weakening of core construction in IL). Suppose $\mathit{fenv} \oplus_? \mathit{fenv}'$.

(1) *If* $\mathit{fenv};\ \mathsf{f} \vdash \mathring{\mathit{import}} \rightsquigarrow \mathring{\mathit{espc}}$
then $\mathit{fenv} \oplus \mathit{fenv}';\ \mathsf{f} \vdash \mathring{\mathit{import}} \rightsquigarrow \mathring{\mathit{espc}}$

(2) *If* $\mathit{fenv};\ \mathsf{f} \vdash \mathring{\mathit{impspec}} \rightsquigarrow \mathring{\mathit{espcs}}$
then $\mathit{fenv} \oplus \mathit{fenv}';\ \mathsf{f} \vdash \mathring{\mathit{impspec}} \rightsquigarrow \mathring{\mathit{espcs}}$

(3) *If* $\mathit{fenv} \vdash \mathring{\mathit{impdecl}} \rightsquigarrow \mathring{\mathit{eenv}}$
then $\mathit{fenv} \oplus \mathit{fenv}' \vdash \mathring{\mathit{impdecl}} \rightsquigarrow \mathring{\mathit{eenv}}$

(4) $\begin{cases} \textit{If} & \cdot\ \textit{fenv};\, \mathsf{f}_0 \vdash \mathring{\textit{impdecls}}; \mathring{\textit{defs}} \rightsquigarrow \mathring{\textit{eenv}} \mathbin{@} \mathring{\omega}_{\mathsf{imp}} | \mathring{\omega}_{\mathsf{loc}} \\ & \cdot\ \mathring{\omega}'_{\mathsf{imp}} = \mathsf{world}^{+}_{\textit{fenv} \oplus \textit{fenv}'}(\mathsf{imps}(\mathring{\textit{impdecls}}))\ \text{defined} \\ & \cdot\ \mathring{\omega}'_{\mathsf{imp}} \oplus_{?} \mathring{\omega}_{\mathsf{loc}} \\ \textit{then} & \\ & \text{(a)}\ \textit{fenv} \oplus \textit{fenv}';\, \mathsf{f}_0 \vdash \mathring{\textit{impdecls}}; \mathring{\textit{defs}} \rightsquigarrow \mathring{\textit{eenv}} \mathbin{@} \mathring{\omega}'_{\mathsf{imp}} | \mathring{\omega}_{\mathsf{loc}} \\ & \text{(b)}\ \mathring{\omega}_{\mathsf{imp}} \sqsubseteq \mathring{\omega}'_{\mathsf{imp}} \end{cases}$

Weakening the file environment does not change the entity environment ($\mathring{\textit{eenv}}$) resulting from import resolution. It does, however, result in an extended world. Used in the proof of weakening for module typing.

Lemma A.95 (Soundness of reinterpreted module world).

(1) $\begin{cases} \textit{If} & \textit{fenv} \vdash \textit{hsmod} : \textit{ftyp} \mathbin{@} \mathring{\omega} \\ \textit{then} & \\ & \cdot\ \mathsf{modworld}_{\textit{fenv}}(\textit{hsmod}; \textit{ftyp})\ \text{defined} \\ & \cdot\ \mathsf{modworld}_{\textit{fenv}}(\textit{hsmod}; \textit{ftyp}) = \mathring{\omega} \end{cases}$

(2) $\begin{cases} \textit{If} & \textit{fenv};\, \mathsf{f}_0 \vdash \textit{tfexp} \mathbin{@} \mathring{\omega} \\ \textit{then} & \\ & \cdot\ \mathsf{modworld}_{\textit{fenv}}(\mathsf{f}_0; \textit{tfexp})\ \text{defined} \\ & \cdot\ \textit{tfexp} = \textit{hsmod} : \textit{ftyp} \;\Rightarrow\; \mathsf{modworld}_{\textit{fenv}}(\mathsf{f}_0; \textit{tfexp}) \sqsupseteq \mathring{\omega} \end{cases}$

(3) $\begin{cases} \textit{If} & \textit{fenv} \vdash \textit{dexp} \\ \textit{then} & \forall \mathsf{f} \mapsto \textit{tfexp} \mathbin{@} \mathring{\omega} \in \textit{dexp} :\ \mathsf{modworld}_{\textit{fenv} \oplus \lfloor \textit{dexp} \rfloor}(\mathsf{f}; \textit{tfexp})\ \text{defined} \end{cases}$

Used in proof of Lemma A.93 and elsewhere.

Corollary A.96 (Well-typing implies world consistency in the IL).
If $\cdot \vdash \textit{dexp}$ and $\textit{dexp}$ is entirely modules, then $\forall \textit{hsmod} : \textit{ftyp} \in \textit{dexp} : \mathsf{modworld}_{\lfloor \textit{dexp} \rfloor}(\textit{hsmod}; \textit{ftyp})$ is defined. (Straightforward corollary of Lemma A.95(3).)

Property A.97 (Properties on world merging and extension).

(1) if $\mathring{\omega}_1 \oplus_{?} \mathring{\omega}_2 \oplus \mathring{\omega}_3$ then $\mathring{\omega}_1 \oplus_{?} \mathring{\omega}_2$ and $\mathring{\omega}_1 \oplus_{?} \mathring{\omega}_3$
(2) if $\mathring{\omega}_1 \oplus \mathring{\omega}_2 = \mathring{\omega}_3 \oplus \mathring{\omega}_4$ then $\forall i, j \in \{1,2,3,4\} :\ \mathring{\omega}_i \oplus_{?} \mathring{\omega}_j$
(3) if $\mathring{\omega}_1 \oplus_{?} \mathring{\omega}_2$ and $\mathring{\omega}_1 \sqsubseteq \mathring{\omega}'_1$ and $\mathring{\omega}_2 \sqsubseteq \mathring{\omega}'_2$ and $\mathring{\omega}'_1 \oplus_{?} \mathring{\omega}'_2$, then $\mathring{\omega}_1 \oplus \mathring{\omega}_2 \sqsubseteq \mathring{\omega}'_1 \oplus \mathring{\omega}'_2$

Used in proof of Lemma A.93(1) and in proof of Lemma A.94(4).

Property A.98 (File worlds extend along contexts).

(1) If $\textit{fenv} \oplus_{?} \textit{fenv}'$ and $\mathsf{f} \in \mathsf{dom}(\textit{fenv})$, then $\mathsf{world}_{\textit{fenv}}(\mathsf{f}) \sqsubseteq \mathsf{world}_{\textit{fenv} \oplus \textit{fenv}'}(\mathsf{f})$.
(2) If $\textit{fenv} \oplus_{?} \textit{fenv}'$ and $\mathsf{f} \in \mathsf{dom}(\textit{fenv})$, then $\mathsf{world}^{+}_{\textit{fenv}}(\mathsf{f}) \sqsubseteq \mathsf{world}^{+}_{\textit{fenv} \oplus \textit{fenv}'}(\mathsf{f})$.

Used in proof of Lemma A.94 (4).

C.2.2 *Merging*

Lemma A.99 (Merging on IL term judgments). Suppose $\textit{fenv}_1 \oplus_{?} \textit{fenv}_2$.

If $\cdot\ (\textit{tfexp}_1 \mathbin{@} \mathring{\omega}_1) \oplus_{?} (\textit{tfexp}_2 \mathbin{@} \mathring{\omega}_2)$
$\cdot\ \textit{fenv}_1;\, \textit{tfexp}_1 \vdash \mathring{\omega}_1 \mathbin{@} \mathring{\omega}$
$\cdot\ \textit{fenv}_2;\, \textit{tfexp}_2 \vdash \mathring{\omega}_2 \mathbin{@} \mathring{\omega}$
$\cdot\ \mathsf{modworld}_{\textit{fenv}_1 \oplus \textit{fenv}_2}(\mathsf{f}_0; \textit{tfexp}_1 \oplus \textit{tfexp}_2)$ defined

then $\textit{fenv}_1 \oplus \textit{fenv}_2;\, \mathsf{f}_0 \vdash \textit{tfexp} \mathbin{@} \mathring{\omega}\textit{tfexp}_1 \oplus \textit{tfexp}_2$
(Proof uses weakening on *tfexp* and merging on *ftyp*-wf.)

C.2.3 *Strengthening*

Lemma A.100 (Preservation of well-formedness and typing under context strengthening)**.** Suppose $\mathsf{F} \subseteq \mathsf{dom}(\mathit{fenv})$ and $\mathsf{depends}_{\mathit{fenv}}(\mathsf{F}) \subseteq \mathsf{F}$.

(1) If $\mathit{fenv} \vdash \mathit{hsmod} : \mathit{ftyp} @ \mathring{\omega}$ and $\mathit{fenv}(\mathsf{f}_0) = \mathit{ftyp}$ and $\mathsf{f}_0 \in \mathsf{F}$, then $\mathit{fenv}|_{\mathsf{F}} \vdash \mathit{hsmod} : \mathit{ftyp} @ \mathring{\omega}$.
(Proof uses Corollary A.102 to show F is suitable for Axiom A.91; defn of $\mathsf{depends}_{\mathit{fenv}}(\mathsf{f}_0)$ to show F suitably large for imports and Lemma A.103 for strengthening of imports.)

(2) If $\mathit{fenv};\, \mathsf{f}_0 \vdash \mathit{tfexp} @ \mathring{\omega}$ and $\mathit{fenv}(\mathsf{f}_0) = \mathsf{typ}(\mathit{tfexp})$ and $\mathsf{f}_0 \in \mathsf{F}$, then $\mathit{fenv}|_{\mathsf{F}};\, \mathsf{f}_0 \vdash \mathit{tfexp} @ \mathring{\omega}$.
(Proof uses Property A.54 and τ part of Lemma A.7.)

(3) If $\cdot \vdash \mathit{dexp}$ and $\mathit{fenv} = \mathsf{mkfenv}(\mathit{dexp})$, then $\cdot \vdash \mathit{dexp}|_{\mathsf{F}}$.
(Proof uses strengthening on *fenv*-wf (Lemma A.7 on Φ) and previous part to show *fenv* wf; Property A.123 and *tfexp* part to show $\mathit{tfexp} @ \mathring{\omega}$ well-typed.)

Lemma A.101 (Locally available entities come from dependencies of imports (IL))**.** Directly corresponds to the analogous lemma in the EL, Lemma A.48. Suppose $\cdot \vdash \mathit{fenv}$ wf.

(1) If $\cdot \vdash \mathit{fenv}$ wf and $\mathit{fenv};\, \mathsf{f} \vdash \mathring{\mathit{import}} \rightsquigarrow \mathring{\mathit{espc}}$, then $\mathsf{ident}(\mathring{\mathit{espc}}) \in \mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{f})$.

(2) If $\cdot \vdash \mathit{fenv}$ wf and $\mathit{fenv};\, \mathsf{f} \vdash \mathring{\mathit{impspec}} \rightsquigarrow \mathring{\mathit{espcs}}$, then $\mathsf{ident}(\mathring{\mathit{espcs}}) \subseteq \mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{f})$.

(3) If $\mathit{fenv} \vdash \mathring{\mathit{impdecl}} \rightsquigarrow \mathring{\mathit{eenv}}$, then

$$(\mathsf{ident}(\mathsf{locals}(\mathring{\mathit{eenv}})) \cup \mathsf{provs}(\mathring{\omega})) \subseteq \mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{imp}(\mathring{\mathit{impdecl}})).$$

(4) If $\mathit{fenv};\, \mathsf{f}_0 \vdash \mathring{\mathit{impdecls}}; \mathring{\mathit{defs}} \rightsquigarrow \mathring{\mathit{eenv}} @ \mathring{\omega}$, then

$$(\mathsf{ident}(\mathsf{locals}(\mathring{\mathit{eenv}})) \cup \mathsf{provs}(\mathring{\omega})) \subseteq \Big(\{\mathsf{f}_0\} \cup \mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{imps}(\mathring{\mathit{impdecls}}))\Big).$$

(Proof of part (4) relies on the fact that if $\mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{f})$ is defined then $\mathsf{provs}(\mathsf{world}^{+}_{\mathit{fenv}}(\mathsf{f})) \subseteq \mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{f})$.)

Corollary A.102 (Dependencies of this mod contains those of locally available entities)**.** If $\mathit{fenv} \vdash \mathit{fenv}$ wf and $\mathit{fenv};\, \mathsf{f}_0 \vdash \mathring{\mathit{impdecls}}; \mathring{\mathit{defs}} \rightsquigarrow \mathring{\mathit{eenv}} @ \mathring{\omega}$ and $\mathsf{imps}(\mathring{\mathit{impdecls}}) = \mathsf{imps}(\mathit{fenv}(\mathsf{f}_0))$, then $\mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{ident}(\mathsf{locals}(\mathring{\mathit{eenv}}))) \subseteq \mathsf{depends}^{+}_{\mathit{fenv}}(\mathsf{f}_0)$.
(Proof splits into cases whether $\mathsf{ident}(\mathit{espc}) = \mathsf{f}_0$, using Lemma A.101 and Property A.50 in the case that it's not f_0.)

Lemma A.103 (Strengthening of import resolution in IL)**.**

(1) If $\mathsf{f} \in \mathsf{F}$ and $\mathit{fenv};\, \mathsf{f} \vdash \mathring{\mathit{import}} \rightsquigarrow \mathring{\mathit{espc}}$, then $\mathit{fenv}|_{\mathsf{F}};\, \mathsf{f} \vdash \mathring{\mathit{import}} \rightsquigarrow \mathring{\mathit{espc}}$.

(2) If $\mathsf{f} \in \mathsf{F}$ and $\mathit{fenv};\, \mathsf{f} \vdash \mathring{\mathit{impspec}} \rightsquigarrow \mathring{\mathit{espcs}}$, then $\mathit{fenv}|_{\mathsf{F}};\, \mathsf{f} \vdash \mathring{\mathit{impspec}} \rightsquigarrow \mathring{\mathit{espcs}}$.

(3) If $\mathsf{imp}(\mathring{\mathit{impdecl}}) \in \mathsf{F}$ and $\mathit{fenv} \vdash \mathring{\mathit{impdecl}} \rightsquigarrow \mathring{\mathit{eenv}}$, then $\mathit{fenv}|_{\mathsf{F}} \vdash \mathring{\mathit{impdecl}} \rightsquigarrow \mathring{\mathit{eenv}}$.

(4) If $\mathsf{imps}(\mathring{\mathit{impdecls}}) \subseteq \mathsf{F}$ and $\mathsf{provs}(\mathring{\omega}) \subseteq \mathsf{F}$ and $\mathit{fenv};\, \mathsf{f}_0 \vdash \mathring{\mathit{impdecls}}; \mathring{\mathit{defs}} \rightsquigarrow \mathring{\mathit{eenv}} @ \mathring{\omega}$, then $\mathit{fenv}|_{\mathsf{F}};\, \mathsf{f}_0 \vdash \mathring{\mathit{impdecls}}; \mathring{\mathit{defs}} \rightsquigarrow \mathring{\mathit{eenv}} @ \mathring{\omega}$.
(Proof uses Lemma A.101 and Corollary A.102.)

C.2.4 *Invariance under Substitution*

Lemma A.104 (Invariance under substitution of IL terms)**.**
Suppose $\cdot \vdash \mathit{fenv}$ wf and $\mathsf{apply}(\theta; \mathit{fenv})$.

(1) *If*
- $\mathit{fenv} \vdash \mathit{hsmod} : \mathit{ftyp} \,@\, \mathring{\omega}$
- $\mathsf{avoidaliases}(\theta; \mathit{hsmod})$
- $\mathsf{apply}(\theta; \mathring{\omega})$ defined
- $\exists\, \mathring{\omega}_0 : \theta\mathit{fenv} \;\oplus_?\; (\theta\mathsf{name}(\mathit{hsmod})){:}(\theta\mathit{ftyp})^{+} @\, \mathring{\omega}_0$
- $\mathsf{modworld}_{\theta\mathit{fenv}}(\theta\mathit{hsmod}; \theta\mathit{ftyp})$ defined

then
- $\theta\mathit{fenv} \vdash \theta\mathit{hsmod} : \theta\mathit{ftyp} \,@\, \mathring{\omega}'$
- $\mathring{\omega}' \sqsupseteq \theta\mathring{\omega}$

(Proof uses Lemma A.105 and Lemma A.110 for import resolution; Lemma A.117 for export resolution; Axiom A.92 for definition checking.)

(2) *If*
- $\mathit{fenv}; \mathsf{f}_0 \vdash \mathit{tfexp} \,@\, \mathring{\omega}$
- $\mathsf{avoidaliases}(\theta; \mathit{tfexp})$
- $\mathsf{apply}(\theta; \mathring{\omega})$ defined
- $\exists\, \mathring{\omega}_0 : \theta\mathit{fenv} \;\oplus_?\; (\theta\mathsf{f}_0){:}\lfloor\theta\mathit{tfexp}\rfloor @\, \mathring{\omega}_0$
- $\mathsf{modworld}_{\theta\mathit{fenv}}(\theta\mathsf{f}_0; \theta\mathit{tfexp})$ defined

then $\theta\mathit{fenv}; \theta\mathsf{f}_0 \vdash \theta\mathit{tfexp} \,@\, \theta\mathring{\omega}$

(Proof uses above part and invariance for τ well-formedness Lemma A.6.)

(3) *If*
- $\mathit{fenv} \vdash \mathit{dexp}$
- $\mathsf{avoidaliases}(\theta; \mathit{dexp})$
- $\theta\mathit{fenv} \;\oplus_?\; \lfloor\theta\mathit{dexp}\rfloor$
- $\forall \mathsf{f} \mapsto \mathit{tfexp} \,@\, \mathring{\omega} \in \mathit{dexp} :\; \mathsf{modworld}_{\theta\mathit{fenv} \,\oplus\, \lfloor\theta\mathit{dexp}\rfloor}(\theta\mathsf{f}; \theta\mathit{tfexp})$ defined

then $\theta\mathit{fenv} \vdash \theta\mathit{dexp}$

(Proof uses Lemma A.126 and invariance on *fenv*-wf and Lemma A.126 for wf-ness of the *dexp*'s file env; Property A.122 and Lemma A.106 and invariance on *tfexp* and merging on *tfexp* for typing of files.)

Lemma A.105 (Consistent augmented environments come from mergeable modules (IL))**.** If $\overline{\mathit{def}} \sqsubseteq \overline{\mathit{d\mathring{s}pc}}$ and $\overline{\mathit{d\mathring{s}pc}} \in \mathit{ftyp}$ and $\mathit{fenv} \;\oplus_?\; \mathsf{f}_0{:}\mathit{ftyp}^{+} @$ [and $\mathsf{nooverlap}(\overline{\mathit{def}})$], then $\mathsf{consistent}(\mathit{a\mathring{e}nv})$, where $\mathit{a\mathring{e}nv} = (\mathit{fenv}; \mathsf{f}_0; \mathit{defs})$. (Proof very straightforward; uses Lemma A.60.)

Lemma A.106. If $\mathsf{apply}(\theta;\mathit{fenv}_1)$ is defined and $\mathsf{apply}(\theta;\mathit{fenv}_2)$ is defined and $\mathsf{apply}(\theta;\mathit{fenv}_1) \;\oplus_?\; \mathsf{apply}(\theta;\mathit{fenv}_2)$, then $\mathsf{apply}(\theta;\mathit{fenv}_1 \oplus \mathit{fenv}_2)$ is defined and equals $\mathsf{apply}(\theta;\mathit{fenv}_1) \oplus \mathsf{apply}(\theta;\mathit{fenv}_2)$ and $\forall \mathsf{f}{:}\mathit{ftyp}^{\mathsf{m}} @ \in \mathit{fenv}_2 : \mathsf{apply}(\theta;\mathit{fenv}_1) \;\oplus_?\; (\theta\mathsf{f}){:}(\theta\mathit{ftyp})^{\mathsf{m}} @$. (First part is just Property A.58; second part uses the fact that substitution on contexts is just the merge of all substituted singletons, and that merge is commutative and associative.)

Definition A.107 (Consistent augmented local environment in IL)**.** Analogous to EL definition.

$$\mathsf{consistent}((\mathit{fenv}; \mathsf{f}_0; \mathit{defs})) \;\overset{\text{def}}{\Leftrightarrow}\; \begin{cases} \mathsf{nooverlap}(\mathit{defs}), \\ \forall \mathit{d\mathring{s}pc} \in \mathit{fenv}(\mathsf{f}_0) :\; \exists \mathit{def} \in \mathit{defs}, \mathit{d\mathring{s}pc}' : \\ \quad \mathit{def} \sqsubseteq \mathit{d\mathring{s}pc}' \;\wedge\; \mathit{d\mathring{s}pc}' \leqslant \mathit{d\mathring{s}pc} \end{cases}$$

Property A.108 (Definability and distributivity of substitution over world merging)**.**

If
- $\forall i \in I : \mathsf{apply}(\theta; \omega_i)$
- $\forall i, j \in I : \theta\omega_i \;\oplus_?\; \theta\omega_j$

then
- $\theta\left(\bigoplus_{i\in I} \omega_i\right)$ defined
- $\theta\left(\bigoplus_{i\in I} \omega_i\right) = \bigoplus_{i\in I} (\theta\omega_i)$

Definition A.109 (Valid substitutions that avoid import aliases in IL)**.** Analogous to EL definition. Make sure that the substitution entirely avoids the logical import aliases in all files.

$$
\begin{array}{lcl}
\mathsf{avoidaliases}(\theta; \mathit{dexp}) & \overset{\text{def}}{\Leftrightarrow} & \begin{cases} \mathsf{apply}(\theta; \mathit{dexp}) \text{ defined} \\ \forall (\mathsf{f} \mapsto \mathit{tfexp}) \in \mathit{dexp} : \ \mathsf{avoidaliases}(\theta; \mathit{tfexp}) \end{cases} \\
\mathsf{avoidaliases}(\theta; - : \mathit{ftyp}) & \overset{\text{def}}{\Leftrightarrow} & \text{always} \\
\mathsf{avoidaliases}(\theta; \mathit{hsmod} : \mathit{ftyp}) & \overset{\text{def}}{\Leftrightarrow} & \mathsf{avoidaliases}(\theta; \mathit{hsmod}) \\
\mathsf{avoidaliases}(\theta; \mathit{hsmod}) & \overset{\text{def}}{\Leftrightarrow} & \begin{cases} \forall \mathsf{f} \in \mathsf{aliases}(\mathit{hsmod}) : \ \theta\mathsf{f} = \mathsf{f} \\ \forall \mathsf{f} \in \mathit{hsmod} : \ \theta\mathsf{f} \neq \mathsf{f} \ \Rightarrow \ \theta\mathsf{f} \notin \mathsf{aliases}(\mathit{hsmod}) \end{cases}
\end{array}
$$

Lemma A.110 (Invariance under substitution of import resolution in IL)**.**
Suppose $\mathsf{apply}(\theta; \mathit{fenv})$ is defined.

$$
(1) \quad \begin{cases} \textit{If} & \mathit{fenv};\, \mathsf{f} \ \vdash \ \overset{\circ}{\mathit{import}} \ \leadsto \ \overset{\circ}{\mathit{espc}} \\ \textit{then} & \theta\mathit{fenv};\, \theta\mathsf{f}_0 \ \vdash \ \theta\overset{\circ}{\mathit{import}} \ \leadsto \ \theta\overset{\circ}{\mathit{espc}} \end{cases}
$$

$$
(2) \quad \begin{cases} \textit{If} & \mathit{fenv};\, \mathsf{f} \ \vdash \ \overset{\circ}{\mathit{impspec}} \ \leadsto \ \overset{\circ}{\mathit{espcs}} \\ \textit{then} & \\ & \text{(a)} \ \mathsf{apply}(\theta; \overset{\circ}{\mathit{espcs}}) \text{ defined} \\ & \text{(b)} \ \theta\mathit{fenv};\, \theta\mathsf{f}_0 \ \vdash \ \theta\overset{\circ}{\mathit{impspec}} \ \leadsto \ \theta\overset{\circ}{\mathit{espcs}} \end{cases}
$$

Proof uses Lemma A.113 and Property A.68 for definedness.

$$
(3) \quad \begin{cases} \textit{If} & \mathit{fenv} \ \vdash \ \overset{\circ}{\mathit{impdecl}} \ \leadsto \ \overset{\circ}{\mathit{eenv}} \\ \textit{then} & \\ & \bullet \ \mathsf{apply}(\theta; \overset{\circ}{\mathit{eenv}}) \text{ defined} \\ & \bullet \ \mathsf{apply}(\theta; \mathring{\omega}) \text{ defined} \\ & \bullet \ \mathring{\omega}' = \mathsf{world}^{+}_{\theta\mathit{fenv}}(\theta\mathsf{imp}(\overset{\circ}{\mathit{impdecl}})) \text{ defined} \\ & \bullet \ \theta\mathit{fenv} \ \vdash \ \theta\overset{\circ}{\mathit{impdecl}} \ \leadsto \ \theta\overset{\circ}{\mathit{eenv}}[\mathring{\omega}'] \end{cases}
$$

Proof uses Property A.68 and Property A.118 and Property A.26 for definedness/distributivity of substituted, merged envs.

$$
(4) \quad \begin{cases} \textit{If} & \bullet \ \mathit{fenv};\, \mathsf{f}_0 \ \vdash \ \overset{\circ}{\mathit{impdecls}}; \overset{\circ}{\mathit{defs}} \ \leadsto \ \overset{\circ}{\mathit{eenv}} \ @ \ \mathring{\omega}_{\mathsf{imp}} | \mathring{\omega}_{\mathsf{loc}} \\ & \bullet \ \mathsf{consistent}(\theta\mathit{fenv}; \theta\mathsf{f}_0; \theta\overset{\circ}{\mathit{defs}}) \\ & \bullet \ \mathring{\omega}'_{\mathsf{imp}} = \mathsf{world}^{+}_{\theta\mathit{fenv}}(\theta\mathsf{imps}(\overset{\circ}{\mathit{impdecls}})]) \text{ defined} \\ & \bullet \ \mathring{\omega}'_{\mathsf{loc}} = \mathsf{apply}(\theta; \mathring{\omega}_{\mathsf{loc}}) \text{ defined} \\ & \bullet \ \mathring{\omega}'_{\mathsf{imp}} \ \oplus ? \ \mathring{\omega}'_{\mathsf{loc}} \\ \textit{then} & \\ & \bullet \ \mathsf{apply}(\theta; \overset{\circ}{\mathit{eenv}}) \text{ defined} \\ & \bullet \ \mathsf{apply}(\theta; \mathring{\omega}_{\mathsf{imp}} \ \oplus \ \mathring{\omega}_{\mathsf{loc}}) \text{ defined} \\ & \bullet \ \theta\mathit{fenv};\, \theta\mathsf{f}_0 \ \vdash \ \theta\overset{\circ}{\mathit{impdecls}}; \theta\overset{\circ}{\mathit{defs}} \ \leadsto \ \theta\overset{\circ}{\mathit{eenv}} \ @ \ \mathring{\omega}'_{\mathsf{imp}} | \mathring{\omega}'_{\mathsf{loc}} \\ & \bullet \ \theta(\mathring{\omega}_{\mathsf{imp}} \ \oplus \ \mathring{\omega}_{\mathsf{loc}}) \ \sqsubseteq \ \mathring{\omega}'_{\mathsf{imp}} \ \oplus \ \mathring{\omega}'_{\mathsf{loc}} \end{cases}
$$

(The judgment above is a modified form of the core environment construction judgment that makes explicit the distinction between the *imported* and *local* worlds, rather than merging them together.) Proof uses Lemma A.113 and Lemma A.116(1) and Property A.67 for definedness/distributivity of substituted, merged import envs; Property A.118 for definedness/distributivity of substituted local env; Lemma A.114 and Lemma A.116(2) and Property A.67 for definedness/distributivity of substituted, merged import and local env; Property A.108 throughout.

The awkward side conditions in part (4) restrict the substitution θ by prohibiting certain kinds of world inconsistency. Premise (iii) ensures mutual consistency of the (substituted) worlds among modules in the context that have been unified. Premise (iv) ensures consistency of the (substitued) local world. And premise (v) ensures mutual consistency of the whole substituted world for this module.

C.2.5 *Cut*

Lemma A.111 (Cut on typing of IL terms).

If
- $\cdot \vdash \mathit{dexp}_1$
- $\lfloor \mathit{dexp}_1 \rfloor \vdash \mathit{dexp}_2$
- $\mathit{dexp}_1 \oplus_? \mathit{dexp}_2$
- $\forall f_1 \mapsto \mathit{tfexp}_1 \,@\, \mathring{\omega}_1 \in \mathit{dexp}_1 :$ $\mathsf{modworld}_{\lfloor \mathit{dexp}_1 \oplus \mathit{dexp}_2 \rfloor}(f_1; \mathit{tfexp}_1)$ defined
- $\forall f_2 \mapsto \mathit{tfexp}_2 \,@\, \mathring{\omega}_2 \in \mathit{dexp}_2 :$ $\mathsf{modworld}_{\lfloor \mathit{dexp}_1 \oplus \mathit{dexp}_2 \rfloor}(f_2; \mathit{tfexp}_2)$ defined

then $\cdot \vdash \mathit{dexp}_1 \oplus \mathit{dexp}_2$

(Proof uses Property A.125 and *Cut* on *fenv*-wf to show wf-ness of $\lfloor \mathit{dexp}_1 \oplus \mathit{dexp}_2 \rfloor$, and

Lemma A.93 and Lemma A.99 to show that each *tfexp* is well-typed.)

C.2.6 *Misc*

Lemma A.112 (Well-formedness of a directory of signatures). If $\mathit{fenv} \vdash \mathit{fenv}'$ wf and $\mathsf{pol}(\mathit{fenv}') = -$ and $\mathit{fenv}' = \mathsf{mkfenv}(\mathit{dexp})$, then $\mathit{fenv} \vdash \mathit{dexp}$. (Proof straightforward application of Lemma A.11.)

C.3 JUDGMENTAL PROPERTIES OF AUXILIARY IL MODULE MACHINERY

C.3.1 *Substitutability*

Lemma A.113 (Imports produce non-overlapping *espcs* that match the context). Suppose $\mathit{fenv} \vdash \mathit{fenv}$ wf and $\mathring{\mathit{aenv}} = (\mathit{fenv}; f_0; \mathit{defs})$.

(1) If $\mathit{fenv}; f \vdash^{\mathrm{IL}}_{\mathrm{c}} \mathring{\mathit{import}} \rightsquigarrow \mathring{\mathit{espc}}$, then $\mathsf{ctxmatch}(\mathring{\mathit{aenv}}; \mathring{\mathit{espc}})$ and $\exists \mathring{\mathit{espc}}' \in \mathit{fenv}(f) : \mathring{\mathit{espc}}' \leqslant \mathring{\mathit{espc}}$.

(2) If $\mathit{fenv}; f \vdash^{\mathrm{IL}}_{\mathrm{c}} \mathring{\mathit{impspec}} \rightsquigarrow \mathring{\mathit{espcs}}$, then $\mathsf{nooverlap}(\mathring{\mathit{espcs}})$ and $\forall \mathit{espc} \in \mathring{\mathit{espcs}} : \mathsf{ctxmatch}(\mathit{aenv}; \mathit{espc})$.

(3) If $\mathit{fenv} \vdash^{\mathrm{IL}}_{\mathrm{c}} \mathring{\mathit{impdecl}} \rightsquigarrow \mathring{\mathit{eenv}}$, then $\mathsf{nooverlap}(\mathsf{locals}(\mathring{\mathit{eenv}}))$ and $\forall \mathit{espc} \in \mathring{\mathit{eenv}} : \mathsf{ctxmatch}(\mathit{aenv}; \mathit{espc})$ and $\forall \mathring{\mathit{eref}} \in \mathsf{dom}(\mathring{\mathit{eenv}}) : \mathring{\mathit{eref}} = \chi$ or $\mathsf{alias}(\mathring{\mathit{impdecl}}).\chi$. (Proof uses Property A.25 for last result.)

Lemma A.114 (Entities in local entity environment produce local matches). $\forall \mathring{\mathit{espc}} \in \mathsf{mklocaleenv}(f_0; \mathit{defs})$, $\mathsf{locmatch}((\mathit{fenv}; f_0; \mathit{defs}); \mathring{\mathit{espc}})$.

Property A.115 (Membership of an *espc* in a merging of entity envs). Straightforward corollary of Property A.69, by definition of $\oplus$ on entity envs.

(1) If $\mathring{\mathit{espc}} \in \mathring{\mathit{eenv}}_1 \oplus \mathring{\mathit{eenv}}_2$ then $\exists i \in \{1, 2\}, \mathring{\mathit{espc}}_i \in \mathring{\mathit{eenv}}_i : \mathring{\mathit{espc}} \oplus_? \mathring{\mathit{espc}}_i$.

(2) If $\mathring{\mathit{espc}} \in \bigoplus_{i \in [1..n]} \mathring{\mathit{eenv}}_i$ then $\exists i \in [1..n], \mathring{\mathit{espc}}_i \in \mathring{\mathit{eenv}}_i : \mathring{\mathit{espc}} \oplus_? \mathring{\mathit{espc}}_i$.

Corollary A.116 (Definedness of substitution on merges of entity envs). Suppose $\mathit{fenv} \vdash \mathit{fenv}$ wf and $\mathsf{apply}(\theta; \mathit{fenv})$ defined and $\mathit{aenv} = (\mathit{fenv}; f_0; \mathit{defs})$.

(1) If $\forall \mathit{espc}_1 \in \mathring{\mathit{eenv}}_1 : \mathsf{ctxmatch}(\mathit{aenv}; \mathit{espc}_1)$ and $\forall \mathit{espc}_2 \in \mathring{\mathit{eenv}}_2 : \mathsf{ctxmatch}(\mathit{aenv}; \mathit{espc}_2)$, then $\mathsf{apply}(\theta; \mathring{\mathit{eenv}}_1)$ defined and $\mathsf{apply}(\theta; \mathring{\mathit{eenv}}_2)$ defined and $\mathsf{apply}(\theta; \mathring{\mathit{eenv}}_1) \oplus_? \mathsf{apply}(\theta; \mathring{\mathit{eenv}}_2)$.

(2) If $\mathsf{consistent}(\mathsf{apply}(\theta; \mathit{aenv}))$ and $\forall \mathit{espc}_1 \in \mathring{\mathit{eenv}}_1 : \mathsf{ctxmatch}(\mathit{aenv}; \mathit{espc}_1)$ and $\forall \mathit{espc}_2 \in \mathring{\mathit{eenv}}_2 : \mathsf{locmatch}(\mathit{aenv}; \mathit{espc}_2)$, then $\mathsf{apply}(\theta; \mathring{\mathit{eenv}}_1)$ defined and $\mathsf{apply}(\theta; \mathring{\mathit{eenv}}_2)$ defined and $\mathsf{apply}(\theta; \mathring{\mathit{eenv}}_1) \oplus_? \mathsf{apply}(\theta; \mathring{\mathit{eenv}}_2)$.

Lemma A.117 (Invariance under substitution of IL export resolution). Suppose $\mathsf{avoidaliases}(\theta; \mathring{\mathit{eenv}})$.

(1) If $\mathsf{f}_0; \mathring{\mathit{eenv}} \vdash^{\mathrm{IL}}_{\mathrm{c}} \mathring{\mathit{export}} \rightsquigarrow \mathring{\mathit{espc}}$, then $\theta\mathsf{f}_0; \mathsf{apply}(\theta; \mathring{\mathit{eenv}}) \vdash^{\mathrm{IL}}_{\mathrm{c}} \theta\mathring{\mathit{export}} \rightsquigarrow \theta\mathring{\mathit{espc}}$. (Proof uses Property A.75.)

(2) If $\mathsf{f}_0; \mathring{\mathit{eenv}} \vdash^{\mathrm{IL}}_{\mathrm{c}} \mathring{\mathit{expdecl}} \rightsquigarrow \mathring{\mathit{espcs}}$, then $\mathsf{apply}(\theta; \mathring{\mathit{espcs}})$ defined and $\theta\mathsf{f}_0; \mathsf{apply}(\theta; \mathring{\mathit{eenv}}) \vdash^{\mathrm{IL}}_{\mathrm{c}} \theta\mathring{\mathit{expdecl}} \rightsquigarrow \mathsf{apply}(\theta; \mathring{\mathit{espcs}})$. (Proof uses Property A.68 for definedness of specs; Property A.67 for definedness/distributivity of substituted specs.)

Property A.118 (Distributivity of substitution over entity environment operations).

- If $\mathsf{apply}(\theta; \mathring{\mathit{espcs}})$ is defined then $\mathsf{apply}(\theta; \mathsf{mkeenv}(\mathring{\mathit{espcs}}))$ is defined and equals $\mathsf{mkeenv}(\mathsf{apply}(\theta; \mathring{\mathit{espcs}}))$.
- If $\mathsf{apply}(\theta; \mathring{\mathit{espcs}})$ is defined then $\mathsf{apply}(\theta; \mathsf{mklocaleenv}(\mathsf{f}_0; \mathit{defs}))$ is defined and equals $\mathsf{mklocaleenv}(\theta\mathsf{f}_0; \theta\mathit{defs})$.
- If $\mathsf{apply}(\theta; \mathring{\mathit{eenv}})$ is defined then $\mathsf{apply}(\theta; \mathsf{qualify}(\mathsf{f}; \mathring{\mathit{eenv}}))$ is defined and equals $\mathsf{qualify}(\theta\mathsf{f}; \mathsf{apply}(\theta; \mathring{\mathit{eenv}}))$.

Lemma A.119 (Definedness of substitution on merges of contextual and local *espc*-sets). Suppose $\mathit{fenv} \vdash \mathit{fenv}$ wf and $\mathsf{apply}(\theta; \mathit{fenv})$ defined and $\mathit{aenv} = (\mathit{fenv}; \mathsf{f}_0; \mathit{defs})$.

(1) If $\forall \mathit{espc}_1 \in \mathit{espcs}_1 : \mathsf{ctxmatch}(\mathit{aenv}; \mathit{espc}_1)$ and $\forall \mathit{espc}_2 \in \mathit{espcs}_2 : \mathsf{ctxmatch}(\mathit{aenv}; \mathit{espc}_2)$, then $\mathsf{apply}(\theta; \mathit{espcs}_1)$ defined and $\mathsf{apply}(\theta; \mathit{espcs}_2)$ defined and $\mathsf{apply}(\theta; \mathit{espcs}_1) \oplus_? \mathsf{apply}(\theta; \mathit{espcs}_2)$.

(2) If $\mathsf{consistent}(\mathsf{apply}(\theta; \mathit{aenv}))$ and $\forall \mathit{espc}_1 \in \mathit{espcs}_1 : \mathsf{ctxmatch}(\mathit{aenv}; \mathit{espc}_1)$ and $\forall \mathit{espc}_2 \in \mathit{espcs}_2 : \mathsf{locmatch}(\mathit{aenv}; \mathit{espc}_2)$, then $\mathsf{apply}(\theta; \mathit{espcs}_1)$ defined and $\mathsf{apply}(\theta; \mathit{espcs}_2)$ defined and $\mathsf{apply}(\theta; \mathit{espcs}_1) \oplus_? \mathsf{apply}(\theta; \mathit{espcs}_2)$.

Definition A.120 (Valid substitutions for an entity environment). The substitution preserves lookup in the entity environment.

$$\mathsf{avoidaliases}(\theta; \mathring{\mathit{eenv}}) \overset{\mathrm{def}}{\Leftrightarrow} \begin{cases} \mathsf{apply}(\theta; \mathring{\mathit{eenv}}) \text{ defined} \\ \forall \mathring{\mathit{eref}}, \mathring{\mathit{phnm}} \text{ s.t. } \mathring{\mathit{eenv}}(\mathring{\mathit{eref}}) = \mathring{\mathit{phnm}} : \\ \quad \mathsf{apply}(\theta; \mathring{\mathit{eenv}})(\theta\mathring{\mathit{eref}}) = \theta\mathring{\mathit{phnm}} \end{cases}$$

Lemma A.121 (Definedness of substituted local world creation).

$$(1) \begin{cases} \textit{If} & \cdot\ \mathsf{avoidaliases}(\mathring{\theta}; \mathring{\mathit{eenv}}) \\ & \cdot\ \omega = \mathsf{mkloc\mathring{a}lworld}(\mathsf{f}_0; \mathring{\mathit{defs}}; \mathring{\mathit{eenv}}) \\ & \cdot\ \omega' = \mathsf{mkloc\mathring{a}lworld}(\mathring{\theta}\mathsf{f}_0; \mathring{\theta}\mathring{\mathit{defs}}; \mathring{\theta}\mathring{\mathit{eenv}}) \\ \textit{then} & \\ & \cdot\ \mathsf{apply}(\mathring{\theta}; \mathring{\omega}) \\ & \cdot\ \mathring{\theta}\mathring{\omega} = \mathring{\omega}' \end{cases}$$

$$(2) \begin{cases} \textit{If} & \cdot\ \mathring{\omega} = \mathsf{extworld}(\mathsf{f}_0; \mathit{ftyp}) \\ & \cdot\ \mathring{\omega}' = \mathsf{extworld}(\mathring{\theta}\mathsf{f}_0; \mathring{\theta}\mathit{ftyp}) \\ \textit{then} & \\ & \cdot\ \mathsf{apply}(\mathring{\theta}; \mathring{\omega}) \\ & \cdot\ \mathring{\theta}\mathring{\omega} = \mathring{\omega}' \end{cases}$$

C.4 ALGEBRAIC PROPERTIES OF IL TERMS

Property A.122 (Structure of sums of directory expressions).

- If $\bigoplus_{i \in [1..n]} \mathit{dexp}_i$ is fully defined and $(\mathsf{f} \mapsto \mathit{tfexp}) \in (\bigoplus_{i \in [1..n]} \mathit{dexp}_i)$ and $I = \{i \in [1..n] \mid \mathsf{f} \in \mathsf{dom}(\mathit{dexp}_i)\}$ and $\{\mathit{tfexp}_i\}_{i \in I} = \{\mathit{tfexp}' \mid \mathsf{f} \mapsto \mathit{tfexp}' \in \Phi_i\}_{i \in I}$, then $\mathit{tfexp} = \bigoplus_{i \in I} \mathit{tfexp}_i$.

Property A.123 (Projections from directory environments). If $\mathit{fenv} = \mathsf{mkfenv}(\mathit{dexp})$ and $\mathit{dexp}(\mathsf{f}) = \mathit{tfexp}$, then $\mathit{fenv}(\mathsf{f}) = \mathsf{typ}(\mathit{tfexp})$.

Property A.124 (Typed file expressions are mergeable when their polarized types are mergeable).

(1) $\mathit{tfexp}_1 \oplus_? \mathit{tfexp}_2$ if and only if $\mathsf{typ}(\mathit{tfexp}_1)^{\mathsf{pol}(\mathit{tfexp}_1)} \oplus_? \mathsf{typ}(\mathit{tfexp}_2)^{\mathsf{pol}(\mathit{tfexp}_2)}$.
(2) $\bigoplus_{i\in[1..n]} \mathit{tfexp}_i$ is defined if and only if $\bigoplus_{i\in[1..n]} \mathsf{typ}(\mathit{tfexp}_i)^{\mathsf{pol}(\mathit{tfexp}_i)}$ is defined.

(Proofs straightforward from definition of *tfexp* merging.)

Property A.125 (Merging of directories commutes with file environment creation)**.**
(1) $\mathit{dexp}_1 \oplus_? \mathit{dexp}_2$ if and only if $\mathsf{mkfenv}(\mathit{dexp}_1) \oplus_? \mathsf{mkfenv}(\mathit{dexp}_2)$, and if so, then $\mathsf{mkfenv}(\mathit{dexp}_1) \oplus \mathsf{mkfenv}(\mathit{dexp}_2) = \mathsf{mkfenv}(\mathit{dexp}_1 \oplus \mathit{dexp}_2)$.
(2) $\bigoplus_{i\in[1..n]} \mathit{dexp}_i$ is defined if and only if $\bigoplus_{i\in[1..n]} \mathsf{mkfenv}(\mathit{dexp}_i)$ is defined, and if so, then $(\bigoplus_{i\in[1..n]} \mathsf{mkfenv}(\mathit{dexp}_i)) = \mathsf{mkfenv}(\bigoplus_{i\in[1..n]} \mathit{dexp}_i)$.

(Proofs straightforward from Property A.124.)

Property A.126 (Substitution on *dexp* commutes with $\mathsf{mkfenv}(-)$)**.** $\mathsf{apply}(\theta; \mathsf{mkfenv}(\mathit{dexp}))$ is defined if and only if $\mathsf{apply}(\theta; \mathit{dexp})$ is defined, and if so, then $\mathsf{apply}(\theta; \mathsf{mkfenv}(\mathit{dexp})) = \mathsf{mkfenv}(\mathsf{apply}(\theta; \mathit{dexp}))$. (Proof straightforward from Property A.125.)

Property A.127 (Distributivity of merging and substitution over *tfexp* attributes)**.**
- If $\mathit{tfexp}_1 \oplus_? \mathit{tfexp}_2$ then $\mathsf{typ}(\mathit{tfexp}_1 \oplus \mathit{tfexp}_2) = \mathsf{typ}(\mathit{tfexp}_1) \oplus \mathsf{typ}(\mathit{tfexp}_2)$.
- If $\mathit{tfexp}_1 \oplus_? \mathit{tfexp}_2$ then $\mathsf{pol}(\mathit{tfexp}_1 \oplus \mathit{tfexp}_2) = \mathsf{pol}(\mathit{tfexp}_1) \oplus \mathsf{pol}(\mathit{tfexp}_2)$.
- Likewise for n-ary merges over a non-empty index set I.
- $\mathsf{typ}(\theta \mathit{tfexp}) = \theta \mathsf{typ}(\mathit{tfexp})$

Property A.128 (Substitution on singleton directory expressions)**.** If $\mathit{dexp} = \{f \mapsto \mathit{tfexp}\}$ then $\mathsf{apply}(\theta; \mathit{dexp})$ is defined and equals $\{(\theta f) \mapsto (\theta \mathit{tfexp})\}$. (Proofs straightforward by the definition of substitution in terms of merging (over a singleton index set).)

D

ELABORATION METATHEORY

D.1 INVARIANTS NEEDED FOR ELABORATION SOUNDNESS

D.1.1 *Import resolution*

Lemma A.129 (Elaboration of imports produces translations of original entity envs)**.** Suppose $\cdot \vdash \Gamma$ wf and $\Gamma = (\Phi; \mathcal{L})$.

(1) $\begin{cases} \textit{If} & \mathsf{shape}(\Gamma); \ell \Vdash \textit{import} \rightsquigarrow \textit{espc} \\ \textit{then} & \Phi^\star; \mathcal{L}(\ell)^\star \vdash \mathsf{mkentimp}(\textit{espc}) \rightsquigarrow \textit{espc}^\star \end{cases}$

(2) $\begin{cases} \textit{If} & \mathsf{shape}(\Gamma); \ell \Vdash \textit{impspec} \rightsquigarrow \textit{espcs} \\ \textit{then} & \Phi^\star; \mathcal{L}(\ell)^\star \vdash \mathsf{mkimpspec}(\mathcal{L}; \ell; \textit{impspec}) \rightsquigarrow \textit{espcs}^\star \end{cases}$

(3) $\begin{cases} \textit{If} & \mathsf{shape}(\Gamma) \Vdash \textit{impdecl} \rightsquigarrow \textit{eenv} \\ \textit{then} & \\ & \cdot\ \textit{eenv}^\star \text{ defined} \\ & \cdot\ \Phi^\star \vdash \mathsf{mkimpdecl}(\Gamma; \textit{impdecl}) \rightsquigarrow \textit{eenv}^\star \end{cases}$

(Proof uses Property A.130. In general, $\textit{eenv}^\star$ could be undefined if any *eref* in it were of the form $\texttt{Local.}\chi$, as there's no way to translate `Local` into the IL via $(-)^\star$; that's what $\mathsf{refs}^\star_{\nu_0}(\textit{eenv})$ does. In this case, however, because the *eenv* results from import resolution, then by the definition of import resolution there will be no *eref* with `Local`.)

(4) $\begin{cases} \textit{If} & \cdot\ \Gamma \vdash \textit{impdecls}; (\nu_0, \ldots, \nu_0 \mid \textit{defs}) \rightsquigarrow \textit{eenv} \mathbin{@} \omega \\ & \cdot\ \mathsf{world}^+_\Phi(\mathcal{L}(\mathsf{imps}(\textit{impdecls}))) \text{ defined} \\ & \cdot\ \mathsf{world}^+_\Phi(\mathcal{L}(\mathsf{imps}(\textit{impdecls}))) \oplus_? \omega \\ \textit{then} & \\ & \cdot\ \Phi^\star; \nu_0^\star \vdash \mathsf{mkimpdecls}(\mathcal{L}; \textit{impdecls}); \mathsf{refs}^\star_{\nu_0}(\textit{defs}) \rightsquigarrow \\ & \quad \mathsf{refs}^\star_{\nu_0}(\textit{eenv}) \mathbin{@} w' \\ & \cdot\ w' \sqsupseteq \omega^\star \end{cases}$

(Proof uses previous part and Property A.130, Property A.144, and Property A.131. The latter two premises are needed in order to construct the $\textit{hsmod} : \tau^\star$ IL derivation, and this is the point where we go from the EL's semantics of "use the worlds of my *direct* imports" to the IL's semantics of "use the worlds of all my *transitive* imports.")

Property A.130 (Distributivity of translation over entity env operations)**.**

- $\mathsf{mkeenv}(\textit{espcs})^\star$ is defined and equals $\mathsf{mkeenv}(\textit{espcs}^\star)$.
- If $\mathsf{qualify}(p; \textit{eenv})$ is defined, then $\textit{eenv}^\star$ is defined, and $\mathsf{qualify}(p; \textit{eenv})^\star$ is defined and equals $\mathsf{qualify}(p; \textit{eenv}^\star)$.
- $\mathsf{refs}^\star_{\nu_0}(\mathsf{mkloceenv}((\nu_0, \ldots, \nu_0 \mid \textit{defs}))) = \mathring{\mathsf{mklocaleenv}}(\nu_0^\star; \mathsf{refs}^\star_{\nu_0}(\textit{defs}))$.
- $\mathsf{locals}(\textit{eenv})^\star = \mathsf{locals}(\textit{eenv}^\star)$.
- If $\textit{eenv}_1 \oplus_? \textit{eenv}_2$, then $\mathsf{refs}^\star_{\nu_0}(\textit{eenv}_1) \oplus \mathsf{refs}^\star_{\nu_0}(\textit{eenv}_2)$ is defined and equals $\mathsf{refs}^\star_{\nu_0}(\textit{eenv}_1 \oplus \textit{eenv}_2)$.
- If $\forall i, j \in [1..n] : \textit{eenv}_i \oplus_? \textit{eenv}_j$, then $\bigoplus_{i \in [1..n]} \mathsf{refs}^\star_{\nu_0}(\textit{eenv}_i)$ is defined and equals $\mathsf{refs}^\star_{\nu_0}(\bigoplus_{i \in [1..n]} \textit{eenv}_i)$.

Property A.131 (Distributivity of translation over world operations)**.**

- If $\nu \in \mathsf{dom}(\Phi)$ then $\mathsf{world}_\Phi(\nu)^\star = \mathsf{world}_{\Phi^\star}(\nu^\star)$.

- If $\omega = \mathsf{mklocworld}((\nu_0, \ldots, \nu_0 \mid \mathit{defs});\ \mathit{eenv})$ is defined, then

  $$\mathsf{mklo\mathring{c}alworld}(\nu_0^\star; \mathsf{refs}^\star_{\nu_0}(\mathit{defs}); \mathsf{refs}^\star_{\nu_0}(\mathit{eenv}))$$

  is defined and equals $\omega^\star$. (Proof uses Property A.137.)

D.1.2 *Core definition checking*

Axiom A.132 (Translation preserves well-typedness of core definitions).
If $\Phi;\ \nu_0;\ \mathit{eenv};\ \omega\ \vdash\ \mathit{defs}\ :\ \mathit{dspcs}$ then $\Phi^\star;\ \nu_0^\star;\ \mathsf{refs}^\star_{\nu_0}(\mathit{eenv});\ \omega^\star\ \vdash\ \mathsf{refs}^\star_{\nu_0}(\mathit{defs})\ :\ \mathit{dspcs}^\star$.

Lemma A.133 (Translated substitution is always valid on elaborations).
If $\mathit{hsmod} = \mathsf{mkmod}(\Gamma; \nu_0; \mathit{eenv}; \mathit{impdecls}; \mathit{defs}; \mathit{espcs})$ is defined, then $\mathsf{avoidaliases}(\theta^\star; \mathit{hsmod})$. ($\theta^\star$ maps file names in the image of $(-)^\star$ to other file names in the image of $(-)^\star$. In *hsmod* each alias is in *ModNames* (*i.e.*, EL logical module names), rather than the image of $(-)^\star$, so $\theta^\star$ is the identity on these and never maps anything into them.)

D.1.3 *Export resolution*

Lemma A.134 (Elaboration of exports produces translations of exports).
(1) If $\mathit{eenv}(\mathit{eref}) = \mathit{phnm} : \mathit{espc}'$ and $\mathit{espc}' \leqslant \mathit{espc} \sqsubseteq \mathit{phnm}$, then $\mathsf{refs}^\star_{\nu_0}(\mathit{eenv})\ \vdash\ \mathsf{mkexp}(\mathit{eenv}; \mathit{espc})\ \rightsquigarrow\ \mathit{espc}^\star$. (Proof uses Property A.137 to show that the translated export reference is well-formed.)
(2) If $\mathit{eenv}\ \Vdash\ \mathit{expdecl}\ \rightsquigarrow\ \mathit{espcs}$ then $\mathsf{refs}^\star_{\nu_0}(\mathit{eenv})\ \vdash\ \mathsf{mkexpdecl}(\mathit{espcs}; \mathit{eenv})\ \rightsquigarrow\ \mathit{espcs}^\star$. (Proof uses Lemma A.136 to prove existence of a syntactic reference that identifies each export (for $\mathsf{mkexpdecl}(-;-)$); and previous part.)

Lemma A.135 (Identifiability of local entities in EL). If $\texttt{Local}.\chi \in \mathsf{dom}(\mathit{eenv})$ and $\mathsf{haslocaleenv}(\mathit{eenv}; \nu_0; \mathit{defs})$ then $\mathit{eenv}(\texttt{Local}.\chi) = [\nu_0]\chi$. (Proof from definition of $\mathsf{mkloceenv}((\nu_0, \ldots, \nu_0 \mid \mathit{defs}))$.)

Lemma A.136 (Identifiability of exported entities).
(1) If $\mathit{eenv}\ \Vdash\ \mathit{export}\ \rightsquigarrow\ \mathit{espc}$ then $\forall \mathit{espc} \in \mathit{espcs} :\ \exists \mathit{eref} \in \mathsf{dom}(\mathit{eenv}), \mathit{espc}' \in \mathit{eenv} : (\mathit{eenv}(\mathit{eref}) = \ell : \mathit{espc}') \wedge \mathit{espc}' \leqslant \mathit{espc} \sqsubseteq \ell$. (Proof clear in all cases but ExpModAll; in that case, follows by defn of $\mathsf{filterespcs}(-;-)$ and $\mathsf{filterespc}(-;-)$.)
(2) If $\mathring{\mathit{eenv}}\ \vdash\ \mathring{\mathit{expdecl}}\ \rightsquigarrow\ \mathring{\mathit{espcs}}$ then $\forall \mathit{espc} \in \mathit{espcs} :\ \exists \mathit{eref} \in \mathsf{dom}(\mathit{eenv}), \mathit{espc}' \in \mathit{eenv} : (\mathit{eenv}(\mathit{eref}) = \ell : \mathit{espc}') \wedge \mathit{espc}' \leqslant \mathit{espc} \sqsubseteq \ell$. (Proof uses Lemma A.135 in ExpLocal case; in ExpList, need to show that all individual lookup results combine to a single one; use previous part and Property A.69 to get some initial *eref* to use; then Property A.138 and Property A.70 to combine those results and relate them to the initial *eref*.)

Property A.137 (Preservation of entity environment lookup by translation).
(1) If $\mathit{eenv}(\mathit{eref}) = \mathit{phnm}$ then $\mathsf{refs}^\star_{\nu_0}(\mathit{eenv})(\mathsf{refs}^\star_{\nu_0}(\mathit{eref})) = \ell^\star$. (Proof uses injectivity of $\mathsf{refs}^\star_{\nu_0}(-)$ (Property A.140).)
(2) If $\mathit{eenv}(\mathit{eref}) = \mathit{phnm} : \mathit{espc}$ then $\mathsf{refs}^\star_{\nu_0}(\mathit{eenv})(\mathsf{refs}^\star_{\nu_0}(\mathit{eref})) = \ell^\star : \mathit{espc}^\star$. (Proof uses Property A.144 and Property A.130.)

Property A.138 (Uniqueness of entities looked up in environment). If $\mathit{eenv}(\mathit{eref}) = \mathit{phnm} : \mathit{espc}$ and $\mathit{eenv}(\mathit{eref}') = \mathit{phnm} : \mathit{espc}'$ then $\mathit{espc} = \mathit{espc}'$. (Proof by name uniqueness in an *espc*-set, $\mathsf{locals}(\mathit{eenv})$.)

D.1.4 *Misc*

Corollary A.139 (Package-level consistency in the elaboration). Suppose $F \subseteq \mathrm{ModIdent}^\star$.
(1) $\mathsf{world}_{\Phi^\star}(F)$ defined for any context Φ.

(2) If $\Phi_1\ \oplus_?\ \Phi_2$ then $\mathsf{world}_{\Phi_1^\star}(F)\ \oplus\ \mathsf{world}_{\Phi_2^\star}(F)$ is defined and equals $\mathsf{world}_{(\Phi_1 \oplus \Phi_2)^\star}(F)$.

(3) $\mathsf{world}^{+}_{\Phi^\star}(F)$ defined for any context Φ.

(4) If $\Phi_1 \oplus_? \Phi_2$ then $\mathsf{world}^{+}_{\Phi_1^\star}(F) \oplus \mathsf{world}^{+}_{\Phi_2^\star}(F)$ is defined and equals $\mathsf{world}^{+}_{(\Phi_1 \oplus \Phi_2)^\star}(F)$.
(This is a corollary to Property A.3. Uses Property A.145.)

D.2 ALGEBRAIC PROPERTIES OF TRANSLATION

Property A.140 (Injectivity of entity translation).

- If $\mathsf{refs}^\star_{\nu_0}(\mathit{eref}_1) = \mathsf{refs}^\star_{\nu_0}(\mathit{eref}_2)$ then $\mathit{eref}_1 = \mathit{eref}_2$.
- If $\mathsf{refs}^\star_{\nu_0}(\mathit{eenv}_1) = \mathsf{refs}^\star_{\nu_0}(\mathit{eenv}_2)$ then $\mathit{eenv}_1 = \mathit{eenv}_2$.

Property A.141 (Translation distributes over (total) substitutions). $(\theta\nu)^\star = \theta^\star\nu^\star$. Likewise for *typ*, *dspc*, *dspcs*, *phnm*, *espc*, *espcs*, τ.

Property A.142 (Physical context well-formed implies translation well-formed). If $\Phi_1 \vdash \Phi_2$ wf then $\Phi_1^\star \vdash \Phi_2^\star$ wf.
(Obvious from structure of judgments and objects. The only change in the definition of well-formedness from EL to IL lies in that of Φ: in the EL there's an additional stipulation on Φ objects in that they must satisfy Package-Level Consistency. That doesn't complicate the proof of this property, however, since the proof simply drops that stipulation when proving $\Phi^\star$ well-formed.)

Property A.143 (Translation distributes over merging).

- If $\Phi_1 \oplus \Phi_2$ defined, then $(\Phi_1 \oplus \Phi_2)^\star = \Phi_1^\star \oplus \Phi_2^\star$, which is itself defined.
- If $\tau_1^{m_1} \oplus \tau_2^{m_2}$ defined, then $(\tau_1^{m_1} \oplus \tau_2^{m_2})^\star = (\tau_1{}^\star)^{m_1} \oplus (\tau_2{}^\star)^{m_2}$, which is itself defined.
- If $\tau_1 \oplus \tau_2$ defined, then $(\tau_1 \oplus \tau_2)^\star = \tau_1^\star \oplus \tau_2^\star$, which is itself defined.
- If $\mathit{dspcs}_1 \oplus \mathit{dspcs}_2$ defined, then $(\mathit{dspcs}_1 \oplus \mathit{dspcs}_2)^\star = \mathit{dspcs}_1^\star \oplus \mathit{dspcs}_2^\star$, which is itself defined.
- If $\mathit{espcs}_1 \oplus \mathit{espcs}_2$ defined, then $(\mathit{espcs}_1 \oplus \mathit{espcs}_2)^\star = \mathit{espcs}_1^\star \oplus \mathit{espcs}_2^\star$, which is itself defined.
- If $\mathit{dspc}_1 \oplus \mathit{dspc}_2$ defined, then $(\mathit{dspc}_1 \oplus \mathit{dspc}_2)^\star = \mathit{dspc}_1^\star \oplus \mathit{dspc}_2^\star$, which is itself defined.
- If $\mathit{espc}_1 \oplus \mathit{espc}_2$ defined, then $(\mathit{espc}_1 \oplus \mathit{espc}_2)^\star = \mathit{espc}_1^\star \oplus \mathit{espc}_2^\star$, which is itself defined.
- If $\omega_1 \oplus \omega_2$ defined, then $(\omega_1 \oplus \omega_2)^\star = \omega_1^\star \oplus \omega_2^\star$ is defined.
- Likewise for all n-ary merges over a non-empty index set I.

Property A.144 (Distributivity of translation over semantic object operations).

- $(\nu_1{:}\tau_1{}^{m_1}@, \ldots, \nu_n{:}\tau_n{}^{m_n}@)^\star = (\nu_1^\star{:}\tau_1^\star{}^{m_1}@, \ldots, \nu_n^\star{:}\tau_n^\star{}^{m_n}@)$.
- If $\tau = \Phi(\nu)$ then $\tau^\star = \Phi^\star(\nu^\star)$.
- If $\mathit{espc} \in \Phi(\nu)$ then $\mathit{espc}^\star \in \Phi^\star(\nu^\star)$.
- If $\mathit{dspc} \in \Phi(\nu)$ then $\mathit{dspc}^\star \in \Phi^\star(\nu^\star)$.
- $\mathsf{allphnms}(\mathit{espc})^\star = \mathsf{allphnms}(\mathit{espc}^\star)$.
- $\mathsf{allphnms}(\mathit{espcs})^\star = \mathsf{allphnms}(\mathit{espcs}^\star)$.

Property A.145 (Properties of translation on worlds).

(1) If $\omega_1 \oplus_? \omega_2$ then $\omega_1^\star \oplus \omega_2^\star$ is defined and equals $(\omega_1 \oplus \omega_2)^\star$.

(2) If $\nu \in \mathsf{dom}(\Phi)$ then $\mathsf{world}_\Phi(\nu)^\star = \mathsf{world}_{\Phi^\star}(\nu^\star)$.

(3) If $\nu \in \mathsf{dom}(\Phi)$ then $\mathsf{world}^{+}_\Phi(\nu)^\star = \mathsf{world}^{+}_{\Phi^\star}(\nu^\star)$.

Lemma A.146 (Kind extraction ignores translation). $\mathsf{mkknd}(\mathit{dspc}^\star)$ is defined if and only if $\mathsf{mkknd}(\mathit{dspc})$ is defined, and if so, $\mathsf{mkknd}(\mathit{dspc}^\star) = \mathsf{mkknd}(\mathit{dspc})$.

Property A.147 (Substitution commutes with translation on contexts). If $\mathsf{apply}(\theta;\Phi)$ is defined then $\mathsf{apply}(\theta^\star;\Phi^\star)$ is defined and equals $\mathsf{apply}(\theta;\Phi)^\star$. (Proof relies on injectivity of $(-)^\star$ and Property A.143.)

D.3 PROPERTIES OF THE SOUNDNESS RELATION

Lemma A.148 (Definition of the members of substitutions of corresponding contexts)**.** Suppose $\Phi = (\nu_i{:}\tau_i{}^{m_i}@\,\omega_i \mid i \in [1..n])$ and *dexp* $= (f_i \mapsto \mathit{tfexp}_i\,@\,\omega_i \mid i \in [1..n])$ and $\forall i \in [1..n] : \nu_i{}^\star = f_i$ and $\mathsf{apply}(\theta;\Phi)$ and $\mathsf{apply}(\theta;\mathit{dexp})$. Then

(1) $\mathsf{dom}(\mathsf{apply}(\theta;\Phi))^\star = \mathsf{dom}(\mathsf{apply}(\theta^\star;\mathit{dexp}))$, and

(2) $\forall \nu{:}\tau^m@\,\omega \in \mathsf{apply}(\theta;\Phi)$:

a) $I = \{i \in [1..n] \mid \nu = \theta\nu_i\}$ and $\tau^m = \bigoplus_{i\in I}(\theta\tau_i)^{m_i}$ and $\omega = \bigoplus_{i\in I}\theta\omega_i$, and

b) $(f \mapsto \mathit{tfexp}\,@\,\dot\omega) \in \mathsf{apply}(\theta^\star;\mathit{dexp})$, where $f = \nu^\star$ and $\mathit{tfexp} = \bigoplus_{i\in I}\theta^\star\mathit{tfexp}_i$ and $\dot\omega = \bigoplus_{i\in I}\theta^\star\dot\omega_i$.

Proof of (1) uses Property A.69 and injectivity of $(-)^\star$ and Property A.141; proof of (2) follows from definitions of substitution and the same properties.)

Lemma A.149 (Related packages form a bijection)**.** If $\Xi \sim \mathit{dexp}$ then

- $\Xi.\Phi = (\nu_i{:}\tau_i{}^{m_i}@\,\omega_i \mid i \in [1..n])$,
- $\mathit{dexp} = \{f_i \mapsto \mathit{tfexp}_i\,@\,\dot\omega_i \mid i \in [1..n]\}$, and
- $\forall i \in [1..n] :\ \nu_i^\star = f_i\ \wedge\ \tau_i^\star = \mathsf{typ}(\mathit{tfexp}_i)\ \wedge\ m_i = \mathsf{pol}(\mathit{tfexp}_i)\ \wedge\ \omega_i^\star = \dot\omega_i$.

(Clear by definition of soundness relation since $(-)^\star$ is injective and the two mappings have the same size.)

Lemma A.150 (A negative EL context is related to its IL stub file translation)**.** If $\cdot \vdash \Phi$ wf and $\mathsf{pol}(\Phi) = -$, then $\Xi \sim \mathsf{mkstubs}(\Phi)$. (Proof uses Property A.142 and Lemma A.112.)

Lemma A.151 (EL contexts translate to the environment of related terms)**.** If $\Phi \sim \mathit{dexp}$ then $\mathsf{mkfenv}(\mathit{dexp}) = \Phi^\star$. (Proof straightforward by Lemma A.149 and Property A.144.)

D.3.1 *Strengthening*

Lemma A.152 (Relatedness of packages is preserved under filtering)**.** If $\Phi \sim \mathit{dexp}$ and $\mathsf{depends}_\Phi(N) \subseteq N$, then $\Phi|_N \sim (\mathit{dexp}|_{N^\star})$. (Proof uses Property A.153; Lemma A.7 and Lemma A.100 for well-formedness of filtered context and directory.)

Property A.153 (Dependencies are preserved by translation)**.** If $\nu \in \mathsf{dom}(\Phi)$ then $\mathsf{depends}_\Phi(\nu)^\star = \mathsf{depends}_{\Phi^\star}(\nu^\star)$.

D.3.2 *Substitutability*

Lemma A.154 (Relatedness of packages is preserved under substitution)**.** If $\Phi \sim \mathit{dexp}$ and $\mathsf{apply}(\theta;\Phi)$ defined, then $\mathsf{apply}(\theta^\star;\mathit{dexp})$ is defined and $\mathsf{apply}(\theta;\Phi) \sim \mathsf{apply}(\theta^\star;\mathit{dexp})$. (Proof uses Lemma A.151 and Lemma A.126 for definability of substitution on *dexp*; Lemma A.155 and Lemma A.104 for well-typedness of $\theta^\star\mathit{dexp}$; Lemma A.6 for well-formedness of $\theta\Phi$; Lemma A.148 for gathering substituted pieces; and Lemma A.156 and Lemma A.157 for preserving the relations.)

Lemma A.155 (Translated substitution is always valid on related *dexp*s)**.** If $\Xi \sim \mathit{dexp}$ then $\mathsf{avoidaliases}(\theta^\star;\mathit{dexp})$. (Every *hsmod* $\in$ *dexp* is the result of elaboration, by definition of the soundness relation.)

Lemma A.156 (Relatedness of single module types is preserved under substitution)**.** If $(\nu{:}\tau^m@\,\omega) \sim (f \mapsto \mathit{tfexp}\,@\,\dot\omega)$ then $((\theta\nu){:}(\theta\tau)^m@\,(\theta\omega)) \sim ((\theta^\star f) \mapsto (\theta^\star\mathit{tfexp})\,@\,\theta^\star\dot\omega)$. (Proof straightforward by Property A.127.)

D.3.3 *Merging*

Lemma A.157 (Relatedness of single module types is preserved under merging)**.**

(1) If $(\nu{:}\tau_1{}^{m_1} @ \omega_1) \sim (f \mapsto \mathit{tfexp}_1 @ \mathring{\omega}_1)$ and $(\nu{:}\tau_2{}^{m_2} @ \omega_2) \sim (f \mapsto \mathit{tfexp}_2 @ \mathring{\omega}_2)$ and $\tau_1^{m_1} \oplus_? \tau_2^{m_2}$ and $\mathit{tfexp}_1 \oplus_? \mathit{tfexp}_2$ and $\omega_1 \oplus_? \omega_2$ and $\mathring{\omega}_1 \oplus_? \mathring{\omega}_2$, then $(\nu{:}(\tau_1 \oplus \tau_2)^{(m_1 \oplus m_2)} @ (\omega_1 \oplus \omega_2)) \sim (f \mapsto (\mathit{tfexp}_1 \oplus \mathit{tfexp}_2) @ (\mathring{\omega}_1 \oplus \mathring{\omega}_2))$. (Proof straightforward by Property A.127.)

(2) Likewise for n-ary merges over a non-empty index set I.

BIBLIOGRAPHY

Ancona, Davide, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca (2005a). "Polymorphic bytecode: compositional compilation for Java-like languages." In: *POPL '05* (cit. on p. 169).

Ancona, Davide, Giovanni Lagorio, and Elena Zucca (2005b). "Smart modules for Java-like languages." In: *FTfJP '05* (cit. on p. 169).

Ancona, Davide, Giovanni Lagorio, and Elena Zucca (2006). "Flexible type-safe linking of components for Java-like languages." In: *JMLC '06* (cit. on p. 169).

Appel, Andrew W. and David B. MacQueen (1994). "Separate Compilation for Standard ML." In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI '94. Orlando, Florida, USA: ACM, pp. 13–23. ISBN: 0-89791-662-X. DOI: 10.1145/178243.178245. URL: http://doi.acm.org/10.1145/178243.178245 (cit. on p. 13).

Birkedal, Lars, Nick Rothwell, Mads Tofte, and David N. Turner (1993). *The ML Kit, Version 1*. Tech. rep. 93/14. DIKU, University of Copenhagen (cit. on p. 12).

Biswas, Sandip K. (1995). "Higher-order Functors with Transparent Signatures." In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, pp. 154–163. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199478. URL: http://doi.acm.org/10.1145/199448.199478 (cit. on p. 10).

Bracha, Gilad and William Cook (1990). "Mixin-based Inheritance." In: *OOPSLA '90* (cit. on p. 21).

Bracha, Gilad and Gary Lindstrom (1992). "Modularity Meets Inheritance." In: *ICCL '92* (cit. on p. 21).

Brady, Edwin (Sept. 2013). "Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation." In: *Journal of Functional Programming* 23.5. URL: https://eb.host.cs.st-andrews.ac.uk/drafts/impldtp.pdf (cit. on p. 13).

Breitner, Joachim, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich (2014). "Safe zero-cost coercions for Haskell." In: *ICFP '14* (cit. on p. 57).

Cardelli, Luca (1997). "Program fragments, linking, and modularization." In: *POPL '97* (cit. on pp. 5, 107, 174).

Considine, Jeffrey (2000). *Efficient Hash-Consing of Recursive Types*. Tech. rep. Boston University. URL: http://www.cs.bu.edu/techreports/pdf/2000-006-hashconsing-recursive-types.pdf (cit. on p. 190).

Corradi, Andrea, Marco Servetto, and Elena Zucca (2011). "DeepFJig: Modular composition of nested classes." In: *PPPJ '11* (cit. on p. 169).

Coutts, Duncan, Isaac Potoczny-Jones, and Don Stewart (2008). "Haskell: Batteries Included." In: *Haskell '08* (cit. on pp. 5, 23).

Crary, Karl (2017). "Modules, Abstraction, and Parametric Polymorphism." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, pp. 100–113. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009892. URL: http://doi.acm.org/10.1145/3009837.3009892 (cit. on p. 9).

Crary, Karl, Robert Harper, and Sidd Puri (1999). "What is a Recursive Module?" In: *PLDI '99* (cit. on pp. 16, 19, 40).

Diatchki, Iavor S., Mark P. Jones, and Thomas Hallgren (2002). "A formal specification of the Haskell 98 module system." In: *Haskell '02* (cit. on pp. 9, 13, 16, 29, 60).

Dreyer, Derek (May 2005). "Understanding and Evolving the ML Module System." PhD thesis. Pittsburgh, PA: Carnegie Mellon University. URL: https://people.mpi-sws.org/~dreyer/thesis/main.pdf (cit. on pp. 11, 18).

Dreyer, Derek (2007a). “A Type System for Recursive Modules.” In: *ICFP ’07* (cit. on pp. 17, 18, 40, 113, 172).

Dreyer, Derek (2007b). “Recursive Type Generativity.” In: *Journal of Functional Programming* 17.4&5. URL: `https://people.mpi-sws.org/~dreyer/papers/dps/jfp.pdf` (cit. on pp. 18, 40, 171).

Dreyer, Derek (2014). *Progress and Preservation Considered Boring: A Paean to Parametricity*. Talk at PLMW ’14. Slides: `http://www.mpi-sws.org/~dreyer/talks/plmw2014-talk.pdf` (cit. on p. 56).

Dreyer, Derek and Andreas Rossberg (2008). “Mixin’ Up the ML Module System.” In: *ICFP ’08* (cit. on pp. 21, 132).

Dreyer, Derek, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller (2007). “Modular Type Classes.” In: *POPL ’07* (cit. on pp. 53, 175, 185).

Duggan, Dominic (2002). “Type-safe linking with recursive DLLs and shared libraries.” In: *ACM Transactions on Programming Languages and Systems* 24.6, pp. 711–804 (cit. on p. 169).

Faxén, Karl-Filip (2002). “A static semantics for Haskell.” In: *Journal of Functional Programming* 12.5 (cit. on pp. 7, 9, 29, 90, 110).

Flatt, Matthew and Matthias Felleisen (1998). “Units: Cool Modules for HOT Languages.” In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI ’98. New York, NY, USA: ACM, pp. 236–248. ISBN: 0-89791-987-4. DOI: `10.1145/277650.277730`. URL: `http://doi.acm.org/10.1145/277650.277730` (cit. on p. 21).

Gauthier, Nadji and Francois Pottier (2004). “Numbering Matters: First-Order Canonical Forms for Second-Order Recursive Types.” In: *ICFP ’04* (cit. on pp. 43, 136, 172, 189, 190).

Gazagnaire, Thomas, Louis Gesbert, and Others (2018). *The OCaml Package Manager 2.0: The opam manual*. URL: `https://opam.ocaml.org/doc/Manual.html` (cit. on p. 5).

Goldfarb, Warren D. (1981). “The Undecidability of the Second-Order Unification Problem.” In: *Theoretical Computer Science* 13, pp. 225–230 (cit. on p. 171).

Harper, Robert and Mark Lillibridge (1994). “A Type-Theoretic Approach to Higher-Order Modules with Sharing.” In: *POPL ’94* (cit. on pp. 12, 60).

Harper, Robert and Benjamin C. Pierce (2005). “Design Considerations for ML-Style Module Systems.” In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. MIT Press (cit. on pp. 13, 174).

Harper, Robert and Chris Stone (2000). “A Type-Theoretic Interpretation of Standard ML.” In: *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press (cit. on pp. 170, 172).

Hudak, Paul, John Hughes, Simon Peyton Jones, and Philip Wadler (2007). “A history of Haskell: Being lazy with class.” In: *HOPL III* (cit. on pp. 1, 2, 6, 53).

Huet, Gérard (Sept. 1976). “Résolution d’équations dans des langages d’ordre $1, 2, \ldots, \omega$.” PhD thesis. Université Paris 7 (cit. on pp. 43, 136, 172).

Im, Hyeonseung, Keiko Nakata, Jacques Garrigue, and Sungwoo Park (2011). “A Syntactic Type System for Recursive Modules.” In: *OOPSLA ’11* (cit. on pp. 172, 182).

Jones, Mark P (1993). *Coherence for qualified types*. Tech. rep. Research Report YALEU/DCS/RR-989, Yale University, Department of Computer Science (cit. on p. 56).

Jones, Mark P. (1994). *Qualified Types: Theory and Practice*. Cambridge University Press (cit. on p. 90).

Kahl, Wolfram and Jan Scheffczyk (2001). “Named Instances for Haskell Type Classes.” In: *Haskell Workshop* (cit. on p. 53).

Kilpatrick, Scott, Derek Dreyer, Simon Peyton Jones, and Simon Marlow (2013). *Backpack: Retrofitting Haskell with Interfaces (Technical Appendix)*. Tech. rep. MPI-SWS. URL: `http://plv.mpi-sws.org/backpack/backpack-appendix.pdf` (cit. on pp. 101, 107).

Kilpatrick, Scott, Derek Dreyer, Simon Peyton Jones, and Simon Marlow (2014). “Backpack: Retrofitting Haskell with Interfaces.” In: *POPL ’14* (cit. on pp. 1, 9, 13, 53, 60, 131, 185).

Knight, Kevin (Mar. 1989). “Unification: A Multidisciplinary Survey.” In: *ACM Computing Surveys* 21.1. DOI: 10.1145/62029.62030. URL: http://doi.acm.org/10.1145/62029.62030 (cit. on p. 189).

Lagorio, Giovanni, Marco Servetto, and Elena Zucca (2012). “Featherweight Jigsaw: Replacing inheritance by composition in Java-like languages.” In: *Information and Computation* 214 (cit. on p. 169).

Leroy, Xavier (1994). “Manifest types, modules, and separate compilation.” In: *POPL '94* (cit. on pp. 12, 13, 60, 87).

Leroy, Xavier (1995). “Applicative Functors and Fully Transparent Higher-Order Modules.” In: *POPL '95* (cit. on pp. 12, 39, 172).

Leroy, Xavier (1996). “A syntactic theory of type generativity and sharing.” In: *Journal of Functional Programming* 6.5 (cit. on p. 173).

Leroy, Xavier (2000). “A modular module system.” In: *Journal of Functional Programming* 10.3, pp. 269–303 (cit. on p. 87).

Leroy, Xavier (2003). *A proposal for recursive modules in Objective Caml*. URL: http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf (cit. on p. 18).

Leroy, Xavier, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon (2017). *The OCaml System release 4.06: Documentation and user's manual*. URL: http://caml.inria.fr/distrib/ocaml-4.06/ocaml-4.06-refman.pdf (cit. on pp. 2, 15, 175).

MacQueen, David B. (1984). “Modules for Standard ML.” In: *LISP and Functional Programming*, pp. 198–207 (cit. on pp. 12, 14).

MacQueen, David B. and Mads Tofte (1994). “A semantics for higher-order functors.” In: *ESOP '94* (cit. on p. 172).

MacQueen, David (1986). “Using Dependent Types to Express Modular Structure.” In: *POPL '86* (cit. on pp. 4, 12).

Marlow, Simon, ed. (2010). *Haskell 2010 Language Report* (cit. on pp. 7, 29, 53, 56, 57, 60, 113).

Milner, Robin, Mads Tofte, and Robert Harper (1990). *The Definition of Standard ML*. MIT Press. URL: http://sml-family.org/sml90-defn.pdf (cit. on pp. 11, 12, 173).

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press. URL: http://sml-family.org/sml97-defn.pdf (cit. on pp. 2, 15, 18, 172, 173).

Mitchell, J. C. and R. Harper (1988). “The Essence of ML.” In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, pp. 28–46. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73563. URL: http://doi.acm.org/10.1145/73560.73563 (cit. on p. 9).

Mitchell, John C. (1986). “Representation Independence and Data Abstraction.” In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: ACM, pp. 263–276. DOI: 10.1145/512644.512669. URL: http://doi.acm.org/10.1145/512644.512669 (cit. on p. 9).

Myreen, Magnus O. and Scott Owens (2014). “Proof-producing translation of higher-order logic into pure and stateful ML.” In: *Journal of Functional Programming* 24.2-3 (cit. on p. 3).

Nystrom, Nathaniel, Xin Qi, and Andrew Myers (2006). “J&: Nested Intersection for Scalable Software Composition.” In: *OOPSLA '06* (cit. on p. 169).

Odersky, Martin and Matthias Zenger (2005). “Scalable component abstractions.” In: *OOPSLA '05* (cit. on p. 169).

Owens, Scott and Matthew Flatt (2006). “From Structures and Functors to Modules and Units.” In: *ICFP '06* (cit. on pp. 21, 169).

Peyton Jones, Simon, Mark Jones, and Erik Meijer (1997a). “Type classes: Exploring the design space.” In: *Haskell Workshop* (cit. on p. 56).

Peyton Jones, Simon, Mark Jones, and Erik Meijer (1997b). “Type classes: an exploration of the design space.” In: *Haskell Workshop* (cit. on p. 76).

Peyton Jones, Simon *et al.* (2003). “Haskell 98 Language and Libraries: the Revised Report.” In: *Journal of Functional Programming* 13.1. URL: http://haskell.org/definition/ (cit. on pp. 7, 29, 62, 97, 104).

Reynolds, John C. (1983). “Types, Abstraction and Parametric Polymorphism.” In: *Information Processing* (cit. on p. 2).

Rossberg, Andreas (2006). “The missing link – Dynamic components for ML.” In: *ICFP '06* (cit. on p. 175).

Rossberg, Andreas (2015). “1ML - Core and modules united (F-ing first-class modules).” In: *ICFP '15*. URL: https://people.mpi-sws.org/~rossberg/papers/Rossberg%20-%201ML%20--%20Core%20and%20modules%20united%20[Extended].pdf (cit. on p. 174).

Rossberg, Andreas and Derek Dreyer (Apr. 2013). “Mixin' Up the ML Module System.” In: *ACM Trans. Program. Lang. Syst.* 35.1, 2:1–2:84. ISSN: 0164-0925. DOI: 10.1145/2450136.2450137. URL: http://doi.acm.org/10.1145/2450136.2450137 (cit. on pp. 21, 22, 40, 87, 109, 169, 170, 172, 183).

Rossberg, Andreas, Claudio V. Russo, and Derek Dreyer (2010). “F-ing Modules.” In: *TLDI '10*. URL: http://www.mpi-sws.org/~rossberg/f-ing (cit. on pp. 12, 50, 170, 172, 182).

Rossberg, Andreas, Claudio Russo, and Derek Dreyer (2014). “F-ing modules.” In: *Journal of Functional Programming* 24 (05), pp. 529–607. ISSN: 1469-7653. DOI: 10.1017/S0956796814000264. URL: http://journals.cambridge.org/article_S0956796814000264 (cit. on pp. 10, 11, 39, 56, 87, 173, 174).

Russo, Claudio V. (1998). “Types for Modules.” PhD thesis. University of Edinburgh (cit. on pp. 172, 173).

Sewell, Peter, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis (2007). “Acute: High-level programming language design for distributed computation.” In: *Journal of Functional Programming* 17.4–5 (cit. on p. 175).

Sulzmann, Martin (2006). “Extracting Programs from Type Class Proofs.” In: *PPDP '06* (cit. on pp. 76, 99).

Swasey, David, Tom Murphy VII, Karl Crary, and Robert Harper (2006). “A separate compilation extension to Standard ML.” In: *ML '06*. URL: https://dl.acm.org/citation.cfm?id=1159876.1159883 (cit. on pp. 5, 15, 175).

Syme, Don (Oct. 2012). *The F♯ 3.0 Language Specification*. URL: https://www.microsoft.com/en-us/research/publication/f-3-0-language-specification/ (cit. on p. 3).

The Coq Development Team (Oct. 2017). *The Coq Proof Assistant, version 8.7.0*. DOI: 10.5281/zenodo.1028037. URL: https://doi.org/10.5281/zenodo.1028037 (cit. on p. 13).

Yang, Edward Z. (Sept. 2017). “Backpack: Toward Practical Mix-in Linking in Haskell.” PhD thesis. Stanford University. URL: https://github.com/ezyang/thesis/releases (cit. on pp. 1, 176–179, 182).