Compositional Compiler Correctness Via Parametric Simulations

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

by

Georg Neis, M.Sc.

Saarbrücken, 2018

Tag des Kolloquiums	7. Juni 2018
Dekan der Fakultät	Univ.-Prof. Dr. Sebastian Hack
Vorsitzender des Prüfungsausschusses	Univ.-Prof. Dr. Sebastian Hack
Erstgutachter / Doktorvater	Prof. Dr. Derek Dreyer
Zweitgutachter	Dr. Deepak Garg
Drittgutachter	Prof. Lars Birkedal, Ph.D.
Akademischer Beisitzer	Marco Patrignani, Ph.D.

Zusammenfassung

Die Verifikation von Compilern ist für die Konstruktion von vollständig verifizierter Software essenziell. Jedoch beschränkt sich die meiste bisherige Forschung (z.B. CompCert) auf das Szenario der Kompilierung von *ganzen* Programmen. Um separate Kompilierung und das Linken von Erzeugnissen *verschiedener* Compiler zu unterstützen, scheint es erforderlich einen kompositionalen Begriff von Compilerkorrektheit zu entwickeln, der modular (kompatibel mit Linken), transitiv (mehrphasige Kompilierung unterstützend), und flexibel ist (anwendbar auf Compiler mit verschiedenen Zwischensprachen oder unkonventionellen Transformationen).

In dieser Arbeit formalisieren wir einen solchen Korrektheitsbegriff basierend auf *parametrischen Simulationen*, und entwickeln auf diese Art einen neuartigen Ansatz zur kompositionalen Compilerverifikation.

1. Wir führen die grundlegende Idee von parametrischen (Bi-)simulationen ein und stellen eine konkrete Instanz vor. Diese bildet eine kompositionale Technik zum Beweisen von Programmäquivalenzen einer höheren ML-ähnlichen Sprache $\mathcal{S}$.

2. Dann betrachten wir eine Umgebung, in der $\mathcal{S}$-Programme in systemnahen Code einer Maschinensprache $\mathcal{T}$ übersetzt werden. Wir verallgemeinern unsere Formulierung von PBs zu *parametrischen interlingualen Simulationen* (PILS), und demonstrieren dass PILS als Fundament von kompositionaler Compilerverifikation dienen können.

Unsere PB- und PILS-Entwicklungen wurden mittels des Beweissystems Coq durchgeführt und korrekt bewiesen.

Summary

Compiler verification is essential for the construction of fully verified software, but most prior work (such as CompCert) has focused on verifying whole-program compilers. To support separate compilation and to enable linking of results from *different* verified compilers, it is important to develop a compositional notion of compiler correctness that is modular (preserved under linking), transitive (supports multi-pass compilation), and flexible (applicable to compilers that use different intermediate languages or employ non-standard program transformations).

In this thesis, we formalize such a notion of correctness based on *parametric simulations*, thus developing a novel approach to compositional compiler verification.

The thesis is roughly divided into two parts.

1. We introduce the basic idea of parametric (bi-)simulations (PBs) and present a concrete instance of it. This instance constitutes a compositional technique for proving equivalences of programs written in the same higher-order ML-like language $\mathcal{S}$.

2. We then move from such a single-language setting to a setting where $\mathcal{S}$ programs are being compiled into low-level code of some machine language $\mathcal{T}$. We generalize our PB formalization to *parametric inter-language simulations* (PILS) and demonstrate that PILS can serve as a foundation for compositional compiler verification.

Both our PB and PILS developments have been carried out and proven correct using the Coq proof assistant.

Note

This thesis is based primarily on the following publications.

- **The Marriage of Bisimulations and Kripke Logical Relations**.
  Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. In *2012 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*.

- **The Transitive Composability of Relation Transition Systems**.
  Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. *Technical Report MPI-SWS-2012-002, May 2012*.

- **Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language**.
  Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. In *2015 ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*.

Acknowledgements

It's been a long journey. I am immensely grateful to my advisor Derek Dreyer and my mentors Andreas Rossberg and Chung-Kil Hur. Thank you so much for your support, guidance, care, and wisdom!

I also want to thank Lars Birkedal and Viktor Vafeiadis for their support and insights; Neel Krishnaswami for excellent explanations; Claudio Russo and Aleks Nanevski for making me feel welcome at conferences; and everyone at MPI-SWS for the great environment.

Last but not least, I thank my family.

My PhD studies were partially funded by a Google European Doctoral Fellowship in Programming Technology. Thank you, Google!

Contents

Chapter 1

Introduction

Contents

1.1 Compiler Correctness

Compilers constitute some of the most critical infrastructure of software development. And yet they often have bugs. In 2011, for instance, Yang *et al.* [92] uncovered 325 bugs in widely used C compilers. In more recent work, Le *et al.* [46] identified 147 confirmed bugs in the industrial-strength GCC and LLVM compilers.

Let us distinguish between two classes of compiler bugs:

1. Bugs that cause the compiler to crash, to run forever, to reject valid source programs, or to produce inefficient code. Those are bad.

2. Bugs that cause the compiler to not preserve the semantics of its input, *i.e.,* to produce code that *behaves differently* than its source program. Those are terrible.

Compiler correctness, at least in the context of this thesis, is the absence of the second class of bugs. These bugs are so bad because they can go unnoticed until they blow up a system in production, with potentially fatal consequences. Moreover, they essentially nullify any efforts of formally verifying the program (usually a very

expensive endeavor), because typical verification tools operate on the source code and simply trust in the compiler being correct. In a sense, these bugs can make a correct program incorrect. Out of the 147 bugs mentioned above, 95 were violations of semantics preservation.

Establishing compiler correctness means providing a formal, machine-checked guarantee that a compiler preserves the semantics of its source programs, and has been an active research area ever since [50, 55, 20]. The standard way of expressing this preservation formally is roughly as follows. Suppose we are working with a distinguished *source* language $\mathcal{S}$ and a distinguished *target* language $\mathcal{T}$. A compiler c from $\mathcal{S}$ to $\mathcal{T}$, modelled as a partial function from $\mathcal{S}$ programs to $\mathcal{T}$ programs, is correct (semantics-preserving) iff for any given $\mathcal{S}$ program $p_{\mathcal{S}}$, for which the compiler produces a $\mathcal{T}$ program $p_{\mathcal{T}}$, this target program $p_{\mathcal{T}}$ *refines* its source program $p_{\mathcal{S}}$:

$$\forall p_{\mathcal{S}}, p_{\mathcal{T}}.\ c(p_{\mathcal{S}}) = p_{\mathcal{T}} \implies \mathsf{Behav}(p_{\mathcal{T}}) \subseteq \mathsf{Behav}(p_{\mathcal{S}})$$

Refinement means that any observable behavior of $p_{\mathcal{T}}$ is also a valid observable behavior of $p_{\mathcal{S}}$, for some common notion of "observable behavior" that both the $\mathcal{S}$ and $\mathcal{T}$ languages share (*e.g.,* termination, I/O events). In the case of $\mathcal{S}$ being a deterministic language, this actually implies that $p_{\mathcal{S}}$ and $p_{\mathcal{T}}$ have exactly the same observable behavior. In the case of $\mathcal{S}$ being non-deterministic, the compiler is permitted to translate away some of the non-determinism. We will see a concrete definition of Behav later on.

Research on compiler correctness has made great progress in the last years. Undoubtetly the most successful project so far has been CompCert [47], an optimizing compiler for most of the C language, targetting PowerPC, ARM and x86 processors. It was developed by Leroy and collaborators using the Coq proof assistant [19]. Indeed, Le *et al.* [46] report that, despite extensive testing, they were unable to uncover a single bug in CompCert. On the more functional side of languages, CakeML [42, 85] has become a serious optimizing ML compiler, implemented and verified in the HOL4 system. (See Chapter 6 for further references.)

However, the notion of semantics preservation stated earlier suffers from a fundamental limitation, and therefore so does CompCert as well as CakeML: it only applies to *whole program* compilation. The issue is that the formulation of refinement in terms of the observable behavior of a program implicitly assumes that the program is complete and thus can be meaningfully executed.

Consider the simple scenario where the source program $p_{\mathcal{S}}$ consists of two modules m_1 and m_2:

$$p_{\mathcal{S}} = \mathsf{link}(m_1, m_2)$$

The correctness statement then tells us that $c(p_{\mathcal{S}})$ refines $p_{\mathcal{S}}$. But what if we want to compile m_1 and m_2 separately and link them together? In that case the target program $p_{\mathcal{T}}$ is given as

$$p_{\mathcal{T}} = \mathsf{link}(c(m_1), c(m_2))$$

and what we actually need to know is whether $p_{\mathcal{T}}$ refines $p_{\mathcal{S}}$, not whether $c(p_{\mathcal{S}})$ refines $p_{\mathcal{S}}$. While it is possible for $p_{\mathcal{T}}$ and $c(p_{\mathcal{S}})$ to be syntactically identical, this situation is

very rare (the compiler must not do cross-module optimizations and linking must be very simple). In general, they are different programs and thus the above correctness statement about $c(p_{\mathcal{S}})$ is useless.

In practice, of course, most programs are indeed linked together from multiple separately-produced modules. Some of these modules may as well be generated by different compilers or even hand-optimized in assembly. The limitation of compiler correctness to whole-program compilation is therefore a serious handicap.

1.2 Compositionality

The obvious question then is: can we define a more general notion of semantics preservation that says what it means for a single *module* in a program to be compiled correctly? And can this be done while assuming as little as possible about how the other modules in the program are compiled?

In particular, we articulate the following three desiderata concerning different aspects of compositionality:

- **Modularity:** To enable verified separate compilation, semantics preservation should be defined at the level of modules, not just whole programs, and it should be preserved under linking. Specifically, suppose that source ($\mathcal{S}$) module m_S is refined by target ($\mathcal{T}$) module m_T, and that $\mathcal{S}$ module m'_S is refined by $\mathcal{T}$ module m'_T. We should then be able to conclude that the $\mathcal{S}$-level linking of m_S and m'_S is refined by the $\mathcal{T}$-level linking of m_T and m'_T. (This is sometimes referred to as "horizontal compositionality".)

  Of course, in the special case of modules representing whole programs, this new notion of semantics preservation should imply the original one.

- **Transitivity:** Compilers are rarely expressed as direct translations but typically consist of several (if not dozens of) individual transformation passes that form a pipeline. Proofs of semantic preservation should therefore be transitively composable. That is, one should be able to prove a compiler correct by verifying refinement for its constituent passes independently and then chaining the results together by transitivity. (This is sometimes referred to as "vertical compositionality".)

- **Flexibility:** It should be possible to prove semantics preservation for a range of different compilers and program transformations, so that the results of different verified compilers (which might employ different intermediate languages) can be safely linked together, and so that hand-optimized and hand-verified machine code can be safely linked with compiler-generated code.[1]

[1] **Note:** In our model of the semantics preservation problem, every module in a program is represented by both an $\mathcal{S}$ and a $\mathcal{T}$ version, and we aim to prove that the $\mathcal{T}$ version refines its $\mathcal{S}$ counterpart. For modules that are compiled by a verified compiler, the $\mathcal{T}$ version is generated automatically by the compiler. But for any module that is hand-coded in $\mathcal{T}$, one must also manually

1.3 State of the Art

We are not the first to broach this question—it has been an active research topic in recent years. But we argue that all previously proposed solutions are lacking in some dimension of compositionality.

Contextual Refinement The natural starting point is the standard notion of *contextual refinement*: target module $m_{\mathcal{T}}$ contextually refines source module $m_{\mathcal{S}}$ if $C[m_{\mathcal{T}}]$ refines $C[m_{\mathcal{S}}]$ for all closing program contexts C. While contextual refinement is inherently modular and transitive, it only applies if the input and output languages are the same (we call this the *intra-language* setting): $m_{\mathcal{T}}$ and $m_{\mathcal{S}}$ are modules written in the same language, since they are placed into (think: linked with) the same program context C. Contextual refinement is therefore unsuitable for realistic compiler verification (unless combined with multi-language semantics, as discussed below).

Bisimulations *Bisimulations* originate in the study of process calculi [52, 72, 73] and have successfully been employed (and generalized) to characterize contextual equivalence in a variety of languages [29, 28, 41, 80, 79, 74, 43, 45, 7, 78]. Unfortunately, bisimulation methods rely on technical devices that prevent them (it seems) from scaling to the *inter*-language setting of transformations from one language to another. Specifically, in order to deal with higher-order functions, bisimulation methods employ various "syntactic" devices that restrict the applicability of the methods to intra-language reasoning (see Section 3.4 for details).

Logical Relations *Logical Relations* were originally used to prove fundamental properties of typed lambda calculi such as strong normalization [84, 76] and lambda-definability results [66, 67]. Later they were applied to relational reasoning [69, 82, 90, 65, 64, 2, 38] and used as an effective technique for proving correctness of a wide variety of program transformations in a wide variety of languages. For instance, Dreyer *et al.*'s recent Kripke logical relations [4, 22] support proofs with a very clean and intuitive high-level structure, based on the idea of establishing a *state transition system* that expresses directly how the "abstract state" of a module may evolve over time. We review this technique in the background Chapter 2, as it is important to the work in this thesis.

---

supply its $\mathcal{S}$ counterpart, which serves as a "specification" that the hand-coded $\mathcal{T}$ module is then proven to refine. (We will see an example of this in Section 5.13.) This means that we can only account for hand-coded $\mathcal{T}$ modules that have *some* $\mathcal{S}$-level counterpart. This is somewhat of a restriction at present, since we focus here on the setting where $\mathcal{S}$ is a high-level, ML-like language and $\mathcal{T}$ a low-level, assembly-like language, and certainly not all assembly modules have an ML-level counterpart. However, we do not view this as a fundamental restriction: there is nothing in principle preventing us from generalizing our approach to a setting where $\mathcal{S}$ itself supports interoperation between high- and low-level modules. We discuss this point further in Section 6.3.

Besides being highly flexible, logical relations are also inherently modular and, unlike contextual refinement and bisimulations, can be used to relate different source and target languages. For these reasons, Benton and Hur [6] proposed the idea of employing logical relations to define compositional semantics preservation.

Hur and Dreyer [32] developed this idea further by formalizing the compositional correctness of a simple, single-pass compiler from an ML-like source language to an idealized assembly language. They additionally demonstrated the flexibility of their inter-language logical relations by using them to verify a contrived but illustrative example, wherein a higher-order ML function was implemented in a rather baroque way by some tricky hand-written *self-modifying* assembly code. Thanks to the modularity of their logical relations method, this highly non-standard assembly code could nonetheless be safely linked with assembly modules produced by their verified compiler, with the resulting assembly program guaranteed to preserve the semantics of the corresponding linked source modules.

Unfortunately, it is not clear how to scale Hur *et al.*'s approach from single- to multi-pass compilers because, although logical relations are modular and flexible, they are not typically transitive. Chapter 2 provides a short review of logical relations, including the issue of transitivity.

Multi-Language Semantics Motivated by the goal of supporting compiler verification for programs that interoperate between different languages, Perconti and Ahmed [62] propose an approach based on *multi-language semantics* [49]. In particular, they define a "big-tent" language that comprises the source, target, and intermediate languages of a compiler, and provides "wrapping" operations for embedding terms of each language within the others. They then use logical relations to prove that every source module is contextually equivalent to a suitably wrapped version of the target module to which it is compiled. In this way, their method synthesizes the benefits of logical relations (modularity and different source and target languages) and contextual equivalence (transitivity).

One downside of their approach is that the intermediate languages (ILs) used in a compiler show up explicitly in the statement of compiler correctness. This leads to a loss of modularity: the semantics of source-level linking is not preserved when linking the results of compilers that have different ILs. Another downside concerns flexibility: the approach seems to be restricted to compilers that use *typed* intermediate and assembly languages. Lastly, Perconti and Ahmed have so far only applied their technique to a compiler for a purely functional source language.

Compositional Verification for CompCert Motivated by the goal of compositional compiler verification, Beringer *et al.* [9] propose an adaptation of the CompCert framework based on a novel operational semantics that differentiates between internal (intra-module) and external (inter-module) function calls. They introduce a notion of "logical simulation relation" that assumes little about the memory transformations performed by external function calls.

Beringer *et al.*'s approach is transitive, but lacking somewhat in modularity and flexibility. Concerning modularity, it does not yet (according to the authors) support fully separate compilation. Concerning flexibility, it depends on compiler passes only performing a restricted set of memory transformations—permitting additional transformations can potentially break the transitivity property. In addition, their method appears to be geared specifically toward compilers in the style of CompCert, which employ a uniform memory model across source, intermediate, and target languages. It is not clear how to generalize their technique to support richer (*e.g.,* ML-like) source languages, or compilers whose source and target languages have different memory models.

1.4 This Thesis

In this thesis, we develop a novel approach to compositional compiler verification in the form of *parametric inter-language simulations (PILS)*. PILS formalize a compositional notion of semantics preservations that matches the criteria described in Section 1.2.

In order to ease the presentation of PILS, the thesis is divided into two main parts.

1.4.1 First Part

In the first part, consisting of Chapter 3, we introduce the basic idea of parametric inter-language simulations in the *intra*-language setting of an ML-like language $\mathcal{S}$. We call this development *parametric bisimulations* (PBs). Focusing on a single high-level language first lets us develop the core theory without getting lost in the details of lower-level languages and their implementation of higher-level concepts.

PBs marry together some of the most appealing aspects of KLRs and bisimulations. In particular, they synthesize bisimulations' support for reasoning about recursive features via coinduction with KLRs' support for reasoning about local state via state transition systems. Moreover, PBs are designed to avoid the limitations of KLRs and bisimulations that preclude their generalization to inter-language reasoning. To achieve this goal, we have to come up with a novel way of accounting for higher-order functions in the context of a coinductive bisimulation-like proof method, without relying on the "syntactic" devices that previous bisimulation methods use.

We explore PBs here in the setting of $\mathcal{S}$—a call-by-value λ-calculus with general recursive types, products, sums, universals, existentials, general references, and I/O—as this provides a relatively clear point of comparison with recent work on both KLRs [22] and bisimulations [79]. Technically, our result is a compositional technique for proving equivalences of $\mathcal{S}$ programs, proven sound with respect to contextual equivalence. With one notable exception (see Section 3.10), we believe PBs are capable of reasoning effectively about all the challenging $\mathcal{S}$ equivalences studied in the aforementioned papers, and we demonstrate PBs' effectiveness on several such

equivalences.

Crucially, the proof of transitivity does not simply exploit soundness and completeness with respect to contextual equivalence[2]. Instead, it is developed from scratch and requires a number of technical innovations.

1.4.2 Second Part

The second main part of the thesis consists of Chapters 4 and 5. In it, we move from the previous single-language setting to a setting where $\mathcal{S}$ programs are being compiled into low-level code of an idealized machine language $\mathcal{T}$. We generalize our PBs formalization to the full parametric inter-language simulations and demonstrate that PILS can serve as a foundation for compositional compiler verification.

Essentially, we are doing for PBs what Hur and Dreyer [32] did for logical relations: namely, taking a proof technique for single-language reasoning and showing how to scale it to inter-language reasoning between high- and low-level languages. The key difference is that, due to its reliance on a logical-relations model, Hur and Dreyer's method only supports verification of single-pass compilation, whereas PILS support verification of multi-pass compilation.

Indeed, PILS satisfy all three compositionality criteria from Section 1.1.

- PILS are *modular*: they enable compiler verification in a way that supports separate compilation and is preserved under linking.
- PILS are *transitive*: we use them to verify **Pilsner**, a simple (but non-trivial) *multi-pass* optimizing compiler from $\mathcal{S}$ to $\mathcal{T}$, going through a CPS-based intermediate language $\mathcal{I}$. After CPS conversion, Pilsner performs several optimizations at the $\mathcal{I}$ level prior to code generation. These optimizations include function inlining, contification, dead code elimination, and hoisting (Figure 1.1). Although Pilsner is relatively simple—it is not nearly as realistic as the (whole-program) verified CakeML compiler, for instance [42]—it is **the first multi-pass compiler for a higher-order imperative language to be *compositionally* verified.**
- PILS are *flexible*: we use them to additionally verify (1) **Zwickel**, a direct (one-pass) non-optimizing compiler from $\mathcal{S}$ to $\mathcal{T}$, and (2) Hur and Dreyer's aforementioned self-modifying code example, programmed as a $\mathcal{T}$ module and proven correct w.r.t. an $\mathcal{S}$-level specification. Thanks to PILS' modularity, the output of Zwickel and the self-modifying $\mathcal{T}$ module can then be safely linked together with the output of Pilsner.

1.4.3 Other Chapters

The two main parts are preceded by Chapter 2, which briefly reviews Kripke logical relations, and followed by Chapter 6, in which we discuss related work and conclude.

[2] In fact, our PB model is not complete, but this is not necessarily a "bad" thing.

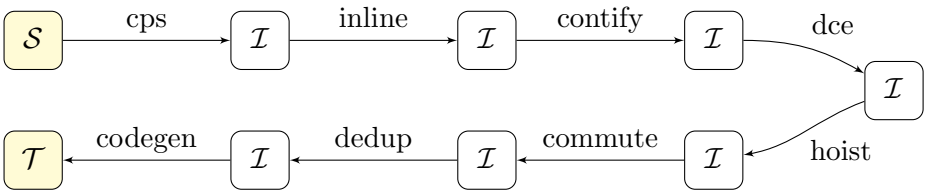


Figure 1.1: Structure of the Pilsner compiler

1.4.4 Proof Mechanization

Both PBs and PILS are formalized using the Coq proof assistant. The extensive development, available at `https://people.mpi-sws.org/~neis/thesis/`, includes machine-checked proofs of all theorems. We also provide glueing code to extract some definitions to OCaml, for instance in order to obtain an executable compiler.

Chapter 2

Background: Logical Relations and Protocols

Contents

Logical relations are one of the best-known methods for local reasoning about observational equivalence[1] in higher-order, typed languages. The basic idea is to lift the notion of observable equivalence at base types to a compositional one at higher types by defining a notion of "logical equivalence" inductively on the type structure of the language. The Curry-Howard correspondence dictates which logical connective to use in assigning a relational operation to each type constructor. For instance, logical relatedness at *function type* is defined with the help of *implication* in terms of logical relatedness at argument type and logical relatedness at result type: two functions are logically related if relatedness of their arguments implies relatedness of their results; similarly, two *existential packages* are logically related if there *exists* a relational interpretation of their hidden type representations that is preserved by their operations; and so forth.

[1] Or, more generally, for local reasoning about observational *refinement*. Here we talk in terms of equivalence, because this is what most of the work focusses on.

2.1 Kripke Logical Relations

Since Reynolds' seminal paper on *relational parametricity* [69], which presented logical relations for reasoning about the pure, strongly normalizing System F, there has been a lot of work on generalizing and extending the method to handle increasingly realistic languages [64, 65, 51, 8, 3, 13]. Most relevant for this thesis is the line of work on modelling languages with state, where an equivalence may only hold due to particular invariants being imposed by the programs on parts of their memory. Naturally, these invariants must play a crucial role in any proof of that equivalence.

This is where *Kripke* logical relations come in. Kripke logical relations [65] are logical relations indexed by a *possible world* W, which codifies memory constraints. Roughly speaking, programs e_1 and e_2 are related under W only if they behave "the same" when run under any memories h_1 and h_2, respectively, that satisfy the constraints of W.

When reasoning about programs that maintain some *local* state, *i.e.,* state that is encapsulated and cannot be freely accessed by the context, possible worlds allow us to impose whatever constraints on the local state we want, so long as we ensure that those constraints are preserved by the code that accesses the state. (Global state, *e.g.,* a reference that is passed to the context at a ref type, will have to obey the canonical invariants dictated by that type.)

In early Kripke logical relations, such as those of Pitts and Stark [65], worlds essentially take the form of simple memory relations, i.e., memory invariants. To make things concrete, consider the following simple example (written in an ML-like language):

$$\begin{array}{lcl} \tau & = & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int} \\ e_1 & = & \lambda f.\,(f\ \langle\rangle; 1) \\ e_2 & = & \mathsf{let}\ x = \mathsf{ref}\ 1\ \mathsf{in}\ \lambda f.\,(f\ \langle\rangle; !x) \end{array}$$

Each program evaluates to a higher-order function that, when applied, calls its argument f (a "callback" function) and then returns a number. In e_1 this number is simply 1. In the case of e_2, however, this number is the value of reference x, which is initially set to 1. In this example, and in the variants to follow, keep in mind that the callback function f might itself call the higher-order function in question (here: the value of e_2).

We would like to show that e_1 and e_2 are observationally equivalent at type τ. The reason, intuitively, is obvious: x is kept private (*i.e.,* it is never leaked to the context) and thus references local state; since it is never modified by the function returned by e_2, it will always point to 1.

To prove this using Kripke logical relations, we would set out to prove that e_1 and e_2 are related under an *arbitrary initial world* W. So suppose we evaluate the two terms under heaps h_1 and h_2 that satisfy W. Since the evaluation of e_1 results in the allocation of some *fresh* heap location l_x for x (*i.e.,* $l_x \notin \mathrm{dom}(h_2)$), we know that the initial world W cannot already contain any constraints governing the contents of l_x. (If it contained such a constraint, h_1 would have had to satisfy it, and hence l_x

would have to be in $\mathrm{dom}(h_1)$.) So we may extend W with a *new* invariant stating that $l_x \hookrightarrow 1$ (*i.e.*, l_x points to 1). More formally, this invariant can be represented as a binary heap relation of the following form:

$$\{(h_1, h_2) \mid h_2(l_x) = 1\}$$

(Note that while in this example no constraints are imposed on e_1's heap, in general such a heap relation can talk about the contents of both heaps and even relate them to each other.)

It then remains to show that the two λ-abstractions are logically related under this extended world—*i.e.,* under the assumption that $l_x \hookrightarrow 1$. The key part of this concerns reasoning about the calls to f. We are guaranteed that if both calls return, the resulting memories again satisfy the extended world. This lets us conclude that both computations return 1 (assuming f terminates).

2.2 State Transition Systems

2.2.1 The Limitation of Worlds as Memory Relations

The applicability of worlds as growing collections of memory relations is fairly limited. To see why, let us look at Pitts and Stark's "awkward" example [65], a slight modification of the previous one.

$$\begin{array}{lcl} \tau & = & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int} \\ e_1 & = & \lambda f.\,(f\ \langle\rangle; 1) \\ e_3 & = & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda f.\,(x := 1; f\ \langle\rangle; !x) \end{array}$$

Only the second program has changed: e_3 is very similar to e_2 from the previous section, but (i) it initializes x with 0 instead of 1, and (ii) the returned function sets x to 1 before calling f.

Clearly, e_1 and e_3 are still equivalent. When the function returned by e_3—say v_3—is called for the very first time, it immediately writes 1 to x. Since there are no other writes, from that point on x will always hold 1 and so the value returned by the first and by any subsequent invocation of v_3 will always be 1 as well.

However, using the notion of worlds from the previous section, we cannot prove this equivalence. The reason is simple: the only "interesting" invariant concerning x is that it points to *either* 0 or 1, but this invariant is insufficient to rule out that after the call to f, the contents of x have changed back to 0. Hence a proof attempt would get stuck.

While this example is clearly contrived, it is also a minimal representative of a common class of programs in which changes to local state occur in some monotonic fashion. As Ahmed *et al.*[4] pointed out, this includes well-known *generative* (or *state-dependent*) abstract data types (ADTs), in which the interpretation of an abstract type grows over time in correspondence with changes to some local state.

2.2.2 The Power of Worlds as State Transition Systems

To address this limitation, Ahmed *et al.*[4] proposed generalizing possible worlds to include the ability for a memory relation to *evolve* over time. Dreyer *et al.*[22] later streamlined and extended Ahmed *et al.*'s approach in various ways, and cast Ahmed *et al.*'s possible worlds as collections of *state transition systems* (STS's), where each state determines a particular heap property, and where the transitions determine how the heap property may evolve. This approach lets one express complex protocols according to which programs manipulate their memory.

Let us illustrate how this works for the example above. Instead of extending the initial world W with a simple heap relation governing location l_x as we did in Section 2.1, we extend W with the following (very simple) STS with initial state s_0:

This expresses that the value at l_x (in e_3's heap) right after allocation is 0, but that it may eventually change to 1, from which point on it will stay 1 forever.

When showing that the returned λ-abstractions, say v_3 and v_4, are logically related in this extended world, it is important that we not only consider their application when our STS resides in its initial state s_0 (in which these functions were returned), but also in any future state (here s_1). Intuitively, this is because we are effectively reasoning about an arbitrary application of v_3 and v_4, not necessarily the first one (*i.e.,* l_x might already have been set to 1 by a previous call). If the calls happen in state s_0, then we transition to state s_1 when v_4 assigns 1 to l_x. If they happen in s_1, then the assignment doesn't change anything, so we just stay in that state. In either case, we conform to the protocol described by our STS and end up in s_1 after the assignment. Hence the call to f always happens when we are in the $l_x \hookrightarrow 1$ state, and since there there is no transition out of it, we know that f can only return in the same state. Consequently, v_4's dereferencing l_x can only result in 1.

2.2.3 Private Transitions

A very useful extension to STSs, also due to Dreyer *et al.*[22], is the distinction between *public* and *private* transitions. Let us motivate this with the help of yet another modification of the "awkward" example. It is originally due to Jacob Thamsborg and has been dubbed the "very awkward" example or "well-bracketed state change" example in the literature.

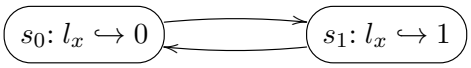
$$\begin{array}{lcl} \tau & = & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int} \\ e_4 & = & \lambda f.\,(f\ \langle\rangle; f\ \langle\rangle; 1) \\ e_5 & = & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda f.\,(x := 0; f\ \langle\rangle; x := 1; f\ \langle\rangle; !x) \end{array}$$

This time both programs have changed: they now make two calls to the callback f, instead of just one. Moreover, in e_5, x is actually set to 0 before the first call, and only afterwards to 1, right before the second call.

Assuming that the language does not provide control effects such as call/cc or exceptions[2], e_4 and e_5 are in fact again equivalent. Intuitively, we can see that whenever x is set to 0, it will eventually be set to 1—no matter what the callback function does (as long as it terminates). Consequently, the final dereferencing will always yield 1.

But let's try to prove this equivalence using STSs. First, recall the STS that we used in order to prove the "awkward" example. It's easy to see why this STS is insufficient for our present purpose: suppose v_5, the function value resulting from evaluating e_5, is applied in the $l_x \hookrightarrow 1$ state. The first thing that happens is that l_x is set to 0. However, as there is no transition from state s_1 back to s_0, there is no way we can continue the proof.

So how about adding that transition from $l_x \hookrightarrow 1$ to $l_x \hookrightarrow 0$, *i.e.,* using the following STS instead?



While doing so clears the first hurdle, it also erects a new one: according to the updated STS, it is now possible that, after the second call to f, we end up in state s_0—even though this situation (l_x pointing to 0 after that call) cannot actually arise in reality. And indeed, if l_x could point to 0 at that point, our proof would be doomed (in fact, using this STS is effectively the same as using the insufficient simple invariant that l_x points to either 0 or 1). In summary, while we would like this transition to be *available to us*, we would at the same time like it to be *unavailable to the context.* This is the idea behind *private transitions.*

Private transitions are a new class of transitions in our STSs, separate from the ordinary transitions that we have seen so far (and which we henceforth call *public transitions*). The basic idea is very simple: when reasoning about the relatedness of terms, we must show that—when viewed *extensionally*—they appear to be making only public transitions, and correspondingly we may assume that the context only makes public transitions as well. Internally, however, *within* a computation, we may make use of both public and private transitions.

Concretely, we can use the following STS to prove our last example, where the dashed arrow denotes a private transition:

First, if v_5 is called in state s_1, the presence of the private transition allows us to "lawfully" transition from s_1t o s_0 when l_x is set to 0. Second, we know that, because we are in the $l_x \hookrightarrow 1$ state before the second call to f and there is no *public* transition from there to any other state, we must still (or again) be in that same state when f returns. Hence we know that l_x holds 1 at that point, as desired. Lastly, although the body of v_5 makes a private transition internally when called in state s_1, it appears

[2]The other examples hold even in the presence of such control features.

extensionally to make a public transition, since its final state (s_1) is obviously publicly accessible from itself.

This distinction of public and private transitions is also very useful in modelling and reasoning about concepts in low-level languages such as stack discipline, as we will see in Chapter 3.

2.3 Dealing With Recursive Language Features

The previous example programs store only integers in the heap, *i.e.,* they are all limited to first-order store. In a higher-order language, however, programs can also store references and functions in the heap. It turns out[2] that, in order for Kripke logical relations to support reasoning about higher-order store, one wants their heap relations to be world-dependent as well. This allows one, for instance, to express the constraint that whatever functions heaps h_1 and h_2 store (at some fixed pair of locations), these functions are logically related *in whatever the current world is.*

However, this immediately leads to a circularity in the construction of the whole model: the universe of worlds depends on the universe of heap relations and the universe of heap relations in turn depends on the universe of worlds, in an way that has no solution.

A different problem arises when trying to support general recursive types. One would like to define logical relatedness at such a type, say $\mu\alpha.\,\tau$, with the help of logical relatedness at the unfolded type $\tau[\mu\alpha.\,\tau/\alpha]$. Since the latter is generally larger, the logical relation can no longer be defined by recursion on types.

A third difficulty concerns reasoning about recursive functions. In the process of showing the relatedness of two recursive functions, one typically must establish the relatedness of the recursive calls, which in turn requires one to show the relatedness of the recursive functions—the very task one set out to do.

2.3.1 Step-Indexing

An elementary technique that solves all three problems at once is *step-indexing* [5, 2]. The idea is simply to stratify the construction of the logical relation by a natural number (or "step index"), representing roughly the number of steps of computation for which the programs in question behave in a related manner.

Regarding the first issue above, worlds of step-index n are then defined in terms of heap relations that can depend on worlds of step-index $n-1$. Intuitively, this is sufficient because, in order for a program to make use of a value stored in the heap, it has to take at least one step (dereferencing a pointer). Regarding the second issue, one defines two values of recursive type to be related for n steps iff their unfoldings are related for $n-1$ steps. Regarding the third issue, one can prove the relatedness of two recursive functions by induction on the step-index, effectively assuming that the functions are already related in any recursive calls.

Due to the tedious threading of step counting throughout proofs [21], it can be somewhat annoying to work with step-indexed logical relations (SILRs). There is a more serious catch, however: it seems fundamentally difficult to compose SILR proofs transitively.

Ahmed studied the transitivity problem in her first paper on binary SILRs [3]. There, she observes a serious problem in naively proving that Appel and McAllester's original binary SILRs formed a PER. She proposes a way of regaining transitivity, but (i) she is concerned only with a very simple pure language, (ii) her proof relies on only working with syntactically well-typed terms, and (iii) it also depends on the second and third program (in the statement of transitivity) coming from the same language. Such an approach is unlikely to scale to reasoning about the intermediate and low-level languages of a compiler, which in general may be untyped. Moreover, we are not aware of any successful attempts to generalize her technique to SILRs for richer languages.

Birkedal & Bizjak [11] propose a different trick to obtain transitivity, but their technique is similarly unsuitable for an inter-language setting.

The lack of transitivity in many of the intra-language models is actually not a big issue, simply because they are only used to reason about contextual equivalence (or refinement), and so one can always transitively compose proofs at the level of contextual equivalence instead. However, when the purpose of the model is precisely to *define* a notion of equivalence—*e.g.,* because contextual equivalence does not make sense—then transitivity is critical.

Chapter 3

Parametric Bisimulations

Contents

3.1 Overview

In this chapter we present our technique for relational reasoning about programs, called **Parametric Bisimulations (PBs)**. PBs marry together some of the most appealing features of KLRs and bisimulations, while circumventing their limitations.

In particular, PBs show how the use of state transition systems (from KLRs) can be synthesized with the coinductive, step-index-free style of reasoning (from bisimulations), thereby enabling clean and elegant proofs about local state *and* recursive features simultaneously. Thus, concerning the long-standing open question of whether there is a fundamental tradeoff between KLRs and bisimulations, we provide a definitive answer: no, there is not.

We explore PBs here in the setting of $\mathsf{F}^{\mu!}$—a call-by-value λ-calculus with general recursive types, products, sums, universals, existentials, and general references [4]—as this provides a clear point of comparison with work on both KLRs [22] and bisimulations [79]. With one notable exception (see Section 3.10), we believe PBs are capable of reasoning effectively about all the challenging $\mathsf{F}^{\mu!}$ equivalences studied in these papers, and we demonstrate PBs' effectiveness on several such equivalences.

In this chapter we focus on intra-language reasoning. In order to enable the generalization to inter-language reasoning, the subject of Chapter 4, PBs were designed

so as to avoid the use of any limiting technical devices. To achieve this goal, we had to come up with a novel way of accounting for higher-order functions in the context of a coinductive bisimulation-like proof method, without relying on the "syntactic" devices that previous bisimulation methods use. Our solution—a new technique we call **global vs. local knowledge**—is one of the major contributions of this work. Relying heavily on this new technique, we have proven that PB equivalence proofs *are* transitively composable, which suggests they may serve as a superior foundation to KLRs for inter-language reasoning.

The remainder of this chapter is structured as follows. In Section 3.3, we define $\mathsf{F}^{\mu!}$, the language under consideration. In Section 3.4, we motivate our key novel technical idea of global vs. local knowledge. We then present the formal development of PBs. For pedagogical reasons, we begin in Section 3.5 with the presentation of a relational model for λ^{μ} (a pure subset of $\mathsf{F}^{\mu!}$ with recursive types), and then proceed in Section 3.6 to extend that model to handle the full language $\mathsf{F}^{\mu!}$. Section 3.7 details the meta theory of the full model. The proof of transitivity merits its own section, Section 3.9. In Section 3.8, we demonstrate the expressive power of our method by proving several challenging equivalences from the literature. Afterwards, in Section 3.10, we explore a variant of PBs that we call *stuttering* PBs, and discuss its applications. In Section 3.11, we revisit local knowledges and point out how applying *parametric coinduction* [34] to their consistency property enables a convenient proof style. Finally, in Section 3.12, we recap and conclude with a comparison between PBs and logical relations.

Because of this chapter's focus on establishing contextual equivalences, it is convenient to develop PBs as symmetric relations. This is not essential, however. When turning to refinement and compiler correctness in the next chapter, we will develop PILS using an asymmetric formulation.

3.2 Notation

Let us briefly establish some basic notational conventions that we use in the remainder of this thesis (not necessarily in the next section).

- We sometimes write meta-level functions in lambda notation (*e.g.,* $\boldsymbol{\lambda} x.x + 42$).
- For a binary relation R and a natural number n, we write R^n for the $n-1$-times iterated relational composition $R \circ \cdots \circ R$.
- For a unary relation S and a natural number n, we write S^n for the $n-1$-times iterated cartesian product $S \times \cdots \times S$.
- We write R^* for the reflexive transitive closure of relation R and R^+ for its transitive closure.
- Given sets X and Y, we write $X \uplus Y$ to denote $X \cup Y$ when we want to emphasize that X and Y are disjoint.

- We write $X_\perp$ short for X extended with a distinct element $\perp$.
- We write $X \rightharpoonup Y$ short for $X \to Y_\perp$, *i.e.,* the set of partial functions from X to Y.
- Given $f \in X \rightharpoonup Y$, we write $\mathrm{dom}(f)$ to denote $\{x \in X \mid f(x) \neq \perp\}$, *i.e.,* the largest subset of X for which f is total.
- We write $X \overset{\text{fin}}{\rightharpoonup} Y$ short for $\{f \in X \rightharpoonup Y \mid \mathrm{dom}(f) \text{ is finite}\}$, *i.e.,* for the set of partial functions from X to Y whose domain is finite.
- Given $f, g \in X \rightharpoonup Y$, we write $f \sqcup g$ to denote either the merge of f and g (if their domains are disjoint) or $\perp$ (if they aren't):

$$f \sqcup g \in (X \rightharpoonup Y)_\perp$$
$$f \sqcup g = \begin{cases} \perp & \text{if } \mathrm{dom}(f) \cap \mathrm{dom}(g) \neq \emptyset \\ h & \text{otherwise} \end{cases}$$

$$\text{where } h(x) = \begin{cases} f(x) & \text{if } x \in \mathrm{dom}(f) \\ g(x) & \text{if } x \in \mathrm{dom}(g) \\ \perp & \text{otherwise} \end{cases}$$

- For better readability, we sometimes write properties (*e.g.,* lemmas) in the form of proof rules. Moreover, in these rules, we often write multiple conclusions in the same style as one writes multiple premises. For instance, the rule

$$\frac{P_1 \quad P_2 \quad P_3}{C_1 \quad C_2}$$

  represents the property $P_1 \wedge P_2 \wedge P_3 \implies C_1 \wedge C_2$.

3.3 The Language $\mathsf{F}^{\mu!}$

3.3.1 Syntax and Static Semantics

Figure 3.2 presents the surface syntax of $\mathsf{F}^{\mu!}$, a call-by-value PCF-like language extended with features such as polymorphism and state. We assume a countably infinite set of type variables α and term variables x.

Types σ consist of variables (α), base types (including natural numbers), binary product and sum types, higher-order function types, universal and existential types, (iso-)recursive types, and general (higher-order) references.

Programs p consists of variables (x) and introduction and elimination constructs for each type form (some of these carrying explicit typing annotations):

- For the unit type: the unit value ($\langle\rangle$).

$$\sigma \in \mathsf{Ty} \;::=\; \alpha \mid \mathsf{unit} \mid \mathsf{nat} \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid \sigma_1 \to \sigma_2 \mid \forall\alpha.\,\sigma \mid \exists\alpha.\,\sigma \mid \mu\alpha.\,\sigma \mid \mathsf{ref}\,\sigma$$

$$\begin{aligned} p \in \mathsf{Prg} \;::=\; & x \mid \langle\rangle \mid n \mid p_1 \odot p_2 \mid \mathsf{ifnz}\; p_0 \;\mathsf{then}\; p_1 \;\mathsf{else}\; p_2 \mid \langle p_1, p_2\rangle \mid p.1 \mid p.2 \mid \mathsf{inl}_\sigma\; p \mid \\ & \mathsf{inr}_\sigma\; p \mid \mathsf{case}\; p\; (x.\; p_1)\; (x.\; p_2) \mid \mathsf{fix}\, f(x{:}\sigma_1){:}\sigma_2.\, p \mid p_1\; p_2 \mid \\ & \Lambda\alpha.\, p \mid p\;[\sigma] \mid \mathsf{pack}\; \langle\sigma, p\rangle\; \mathsf{as}\; \exists\alpha.\,\sigma' \mid \mathsf{unpack}\; p_1\; \mathsf{as}\; \langle\alpha, x\rangle\; \mathsf{in}\; p_2 \mid \\ & \mathsf{roll}_\sigma\; p \mid \mathsf{unroll}\; p \mid \mathsf{ref}\; p \mid !p \mid p_1 := p_2 \mid p_1 == p_2 \end{aligned}$$

Figure 3.1: Surface syntax of $\mathsf{F}^{\mu!}$.

- For the natural number type: constants (n), standard binary arithmetic operations ($p_1 \odot p_2$) such as addition and multiplication, and an "if not zero" conditional ($\mathsf{ifnz}\; p_0 \;\mathsf{then}\; p_1 \;\mathsf{else}\; p_2$).
- For product types: product formation ($\langle p_1, p_2\rangle$) and projections ($p.1$, $p.2$).
- For sum types: injections ($\mathsf{inl}_\sigma\; p$, $\mathsf{inr}_\sigma\; p$) and case analysis ($\mathsf{case}\; p\; (x.\; p_1)\; (x.\; p_2)$).
- For function types: (recursive) abstraction ($\mathsf{fix}\, f(x{:}\sigma_1){:}\sigma_2.\, p$) and application ($p_1\; p_2$).
- For universal types: generalization ($\Lambda\alpha.\, p$) and instantiation ($p\;[\sigma]$).
- For existential types: packing ($\mathsf{pack}\; \langle\sigma, p\rangle\; \mathsf{as}\; \exists\alpha.\,\sigma'$) and unpacking ($\mathsf{unpack}\; p_1\; \mathsf{as}\; \langle\alpha, x\rangle\; \mathsf{in}\; p_2$).
- For recursive types: folding ($\mathsf{roll}_\sigma\; p$) and unfolding ($\mathsf{unroll}\; p$).
- For reference types: allocation ($\mathsf{ref}\; p$), reading ($!p$), writing ($p_1 := p_2$), and comparing ($p_1 == p_2$).

The static semantics of $\mathsf{F}^{\mu!}$, given in Figure 3.2, is completely standard. It defines the program typing judgment $\Delta; \Gamma \vdash p : \sigma$, stating that program p has type σ in environments Δ and Γ. Δ lists the type variables that may appear free in p, while Γ lists the term variables and their types. We write $\sigma[\sigma'/\alpha]$ for the standard capture-avoiding substitution of σ' for variable α in σ.

3.3.2 Dynamic Semantics

The operational semantics of $\mathsf{F}^{\mu!}$ is shown in Figure 3.3 and defined in terms of a standard substitution-based small-step reduction, $\hookrightarrow$, of run-time entities. These run-time entities, representing programs in execution, are pairs consisting of a *heap* h and an *expression* e. Initially, an expression is obtained simply by erasing the type annotations from a program p in the obvious way (written $|p|$). Expressions, however, can also contain heap *locations* l, as created by an allocation step (we

Type environments $\Delta ::= \cdot \mid \Delta, \alpha$
Term environments $\Gamma ::= \cdot \mid \Gamma, x{:}\sigma$

$\boxed{\Delta; \Gamma \vdash p : \sigma}$

$$\frac{\Delta \vdash \Gamma \quad x{:}\sigma \in \Gamma}{\Delta; \Gamma \vdash x : \sigma} \qquad \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash n : \mathsf{nat}} \qquad \frac{\Delta; \Gamma \vdash p_1 : \mathsf{nat} \quad \Delta; \Gamma \vdash p_2 : \mathsf{nat}}{\Delta; \Gamma \vdash p_1 \odot p_2 : \mathsf{nat}}$$

$$\frac{\Delta; \Gamma \vdash p_0 : \mathsf{nat} \quad \Delta; \Gamma \vdash p_1 : \sigma \quad \Delta; \Gamma \vdash p_2 : \sigma}{\Delta; \Gamma \vdash \mathsf{ifnz}\ p_0\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 : \sigma}$$

$$\frac{\Delta; \Gamma \vdash p_1 : \sigma_1 \quad \Delta; \Gamma \vdash p_2 : \sigma_2}{\Delta; \Gamma \vdash \langle p_1, p_2 \rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Delta; \Gamma \vdash p : \sigma_1 \times \sigma_2}{\Delta; \Gamma \vdash p.1 : \sigma_1} \qquad \frac{\Delta; \Gamma \vdash p : \sigma_1 \times \sigma_2}{\Delta; \Gamma \vdash p.2 : \sigma_2}$$

$$\frac{\Delta; \Gamma \vdash p : \sigma_1 \quad \Delta; \Gamma \vdash \sigma_2}{\Delta; \Gamma \vdash \mathsf{inl}_{\sigma_2}\ p : \sigma_1 + \sigma_2} \qquad \frac{\Delta; \Gamma \vdash \sigma_1 \quad \Delta; \Gamma \vdash p : \sigma_2}{\Delta; \Gamma \vdash \mathsf{inr}_{\sigma_1}\ p : \sigma_1 + \sigma_2}$$

$$\frac{\Delta; \Gamma \vdash p_0 : \sigma_1 + \sigma_2 \quad \Delta; \Gamma, x{:}\sigma_1 \vdash p_1 : \sigma \quad \Delta; \Gamma, x{:}\sigma_2 \vdash p_2 : \sigma}{\Delta; \Gamma \vdash \mathsf{case}\ p_0\ (x.\ p_1)\ (x.\ p_2) : \sigma}$$

$$\frac{\Delta; \Gamma, f{:}(\sigma_1 \to \sigma_2), x{:}\sigma_1 \vdash p : \sigma_2}{\Delta; \Gamma \vdash \mathsf{fix}\, f(x{:}\sigma_1){:}\sigma_2.\, p : \sigma_1 \to \sigma_2} \qquad \frac{\Delta; \Gamma \vdash p_1 : \sigma_1 \to \sigma_2 \quad \Delta; \Gamma \vdash p_2 : \sigma_1}{\Delta; \Gamma \vdash p_1\ p_2 : \sigma_2}$$

$$\frac{\Delta, \alpha; \Gamma \vdash p : \sigma}{\Delta; \Gamma \vdash \Lambda\alpha.\, p : \forall\alpha.\, \sigma} \qquad \frac{\Delta; \Gamma \vdash p : \forall\alpha.\, \sigma_1 \quad \Delta; \Gamma \vdash \sigma_2}{\Delta; \Gamma \vdash p\ [\sigma_2] : \sigma_1[\sigma_2/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash \sigma_1 \quad \Delta; \Gamma \vdash p : \sigma_2[\sigma_1/\alpha]}{\Delta; \Gamma \vdash \mathsf{pack}\ \langle \sigma_1, p \rangle\ \mathsf{as}\ \exists\alpha.\, \sigma_2 : \exists\alpha.\, \sigma_2}$$

$$\frac{\Delta; \Gamma \vdash p_1 : \exists\alpha.\, \sigma_1 \quad \Delta, \alpha; \Gamma, x{:}\sigma_1 \vdash p_2 : \sigma_2 \quad \Delta; \Gamma \vdash \sigma_2}{\Delta; \Gamma \vdash \mathsf{unpack}\ p_1\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ p_2 : \sigma_2}$$

$$\frac{\Delta; \Gamma \vdash p : \sigma[\mu\alpha.\, \sigma/\alpha]}{\Delta; \Gamma \vdash \mathsf{roll}_{\mu\alpha.\,\sigma}\ p : \mu\alpha.\, \sigma} \qquad \frac{\Delta; \Gamma \vdash p : \mu\alpha.\, \sigma}{\Delta; \Gamma \vdash \mathsf{unroll}\ p : \sigma[\mu\alpha.\, \sigma/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash p : \sigma}{\Delta; \Gamma \vdash \mathsf{ref}\ p : \mathsf{ref}\ \sigma} \qquad \frac{\Delta; \Gamma \vdash p_1 : \mathsf{ref}\ \sigma \quad \Delta; \Gamma \vdash p_2 : \sigma}{\Delta; \Gamma \vdash p_1 := p_2 : \mathsf{unit}}$$

$$\frac{\Delta; \Gamma \vdash p : \mathsf{ref}\ \sigma}{\Delta; \Gamma \vdash\ !p : \sigma} \qquad \frac{\Delta; \Gamma \vdash p_1 : \mathsf{ref}\ \sigma \quad \Delta; \Gamma \vdash p_2 : \mathsf{ref}\ \sigma}{\Delta; \Gamma \vdash p_1 == p_2 : \mathsf{nat}}$$

Figure 3.2: Static semantics of $\mathsf{F}^{\mu!}$

assume a countably infinite set of locations Loc). A heap is a mapping from a finite subset of Loc to *values* v, which are a syntactic subset of expressions.

The reduction relation follows an eager left-to-right evaluation strategy and is defined the usual way using *evaluation contexts* K. We write $K[e]$ for filling the hole $\bullet$ in K with expression e (defined in the obvious way). We also write $e[v/x]$ for the standard capture-avoiding substitution of value v for variable x in e, and $[l \mapsto v]$ for the singleton heap storing v at location l.

3.3.3 Contextual Equivalence

Our definition of *contextual equivalence* of $\mathsf{F}^{\mu!}$ programs is also fairly standard. Roughly, two programs of the same type are contextually equivalent iff putting them in the same arbitrary (but well-typed) program context yields the same termination behavior.

Formally, we first define program contexts C (programs with a hole) and their typing, analogous to programs. This is shown in Figure 3.4. Context typing has the property that if $\vdash C : (\Delta; \Gamma; \sigma) \rightsquigarrow (\Delta'; \Gamma'; \sigma')$ and $\Delta; \Gamma \vdash p : \sigma$, then $\Delta'; \Gamma' \vdash C[p] : \sigma'$.

To express the termination behavior it suffices to define *divergence*. Intuitively, $\langle h; e\rangle\uparrow$ holds iff the repeated reduction of $\langle h; e\rangle$ never results in a value and never gets stuck, *i.e.,* goes on forever.

Definition 1 (Divergence)**.** Coinductively, using a single recursive rule:

$$\frac{\langle h; e\rangle \hookrightarrow \langle h'; e'\rangle \quad \langle h'; e'\rangle\uparrow}{\langle h; e\rangle\uparrow}$$

Contextual equivalence is then defined as follows.

Definition 2 (Contextual equivalence)**.**

$$\boxed{\Delta; \Gamma \vdash p_1 \sim_{\text{ctx}} p_2 : \sigma} \overset{\text{def}}{\iff} \Delta; \Gamma \vdash p_1 : \sigma \ \wedge\ \Delta; \Gamma \vdash p_2 : \sigma \ \wedge\ \forall C, \sigma', h.\ \vdash C : (\Delta; \Gamma; \sigma) \rightsquigarrow (\epsilon; \epsilon; \sigma') \implies (\langle h; |C[p_1]|\rangle\uparrow \iff \langle h; |C[p_2]|\rangle\uparrow)$$

3.3.4 Deterministic Allocation

Notice that reduction (3.3) is deterministic except for the rule for reference allocation (the $\mathsf{ref}\ e$ expression form), which is completely non-deterministic. For technical reasons, however, it is more convenient to work with a deterministic allocation rule.

Given an allocation function $alloc \in \mathsf{Heap} \to \mathsf{Loc}$, let $\sim_{\text{ctx}}^{alloc}$ be the contextual equivalence of a deterministic version of $\mathsf{F}^{\mu!}$, obtained by replacing the allocation rule with the following one (note the modified side condition):

$$\langle h; K[\mathsf{ref}\ v]\rangle \ \hookrightarrow\ \langle h \sqcup [l \mapsto v]; K[l]\rangle \qquad (l = alloc(h))$$

$l \in \mathsf{Loc}$

$$
\begin{array}{lcl}
e \in \mathsf{Exp} & ::= & x \mid \langle\rangle \mid n \mid e_1 \odot e_2 \mid \mathsf{ifnz}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \langle e_1, e_2\rangle \mid e.1 \mid e.2 \mid \mathsf{inl}\ e \mid \\
& & \mathsf{inr}\ e \mid \mathsf{case}\ e\ (x.\ e_1)\ (x.\ e_2) \mid \mathsf{fix}\, f(x).\, e \mid e_1\ e_2 \mid \Lambda.\, e \mid e\ [] \mid \\
& & \mathsf{pack}\ e \mid \mathsf{unpack}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 \mid \mathsf{roll}\ e \mid \mathsf{unroll}\ e \mid \mathsf{ref}\ e \mid\ !e \mid e_1 := e_2 \mid \\
& & e_1 == e_2 \mid l \\
v \in \mathsf{Val} & ::= & \langle\rangle \mid n \mid \langle v_1, v_2\rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \mathsf{fix}\, f(x).\, e \mid \Lambda.\, e \mid \mathsf{pack}\ v \mid \mathsf{roll}\ v \mid l \\
K \in \mathsf{Cont} & ::= & \bullet \mid K \odot e \mid v \odot K \mid \mathsf{ifnz}\ K\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \langle K, e\rangle \mid \langle v, K\rangle \mid K.1 \mid K.2 \mid \\
& & \mathsf{inl}\ K \mid \mathsf{inr}\ K \mid \mathsf{case}\ K\ (x.\ e_1)\ (x.\ e_2) \mid K\ e \mid v\ K \mid K\ [] \mid \\
& & \mathsf{pack}\ K \mid \mathsf{unpack}\ K\ \mathsf{as}\ x\ \mathsf{in}\ e \mid \mathsf{roll}\ K \mid \mathsf{unroll}\ K \mid \mathsf{ref}\ K \mid\ !K \mid K := e \mid \\
& & v := K \mid K == e \mid v == K \\
h \in \mathsf{Heap} & := & \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val}
\end{array}
$$

Reduction: $\boxed{\langle h; e\rangle \hookrightarrow \langle h'; e'\rangle}$

$$
\begin{array}{lcll}
\langle h; K[n_1 \odot n_2]\rangle & \hookrightarrow & \langle h; K[n]\rangle & (n = [\![\odot]\!]\,(n_1, n_2)) \\
\langle h; K[\mathsf{ifnz}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\rangle & \hookrightarrow & \langle h; K[e_2]\rangle & \\
\langle h; K[\mathsf{ifnz}\ n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\rangle & \hookrightarrow & \langle h; K[e_1]\rangle & (n \neq 0) \\
\langle h; K[\langle v_1, v_2\rangle.1]\rangle & \hookrightarrow & \langle h; K[v_1]\rangle & \\
\langle h; K[\langle v_1, v_2\rangle.2]\rangle & \hookrightarrow & \langle h; K[v_2]\rangle & \\
\langle h; K[\mathsf{case}\ (\mathsf{inl}\ v)\ (x.\ e_1)\ (x.\ e_2)]\rangle & \hookrightarrow & \langle h; K[e_1[v/x]]\rangle & \\
\langle h; K[\mathsf{case}\ (\mathsf{inr}\ v)\ (x.\ e_1)\ (x.\ e_2)]\rangle & \hookrightarrow & \langle h; K[e_2[v/x]]\rangle & \\
\langle h; K[(\mathsf{fix}\, f(x).\, e)\ v]\rangle & \hookrightarrow & \langle h; K[e[\mathsf{fix}\, f(x).\, e/f][v/x]]\rangle & \\
\langle h; K[(\Lambda.\, e)\ []]\rangle & \hookrightarrow & \langle h; K[e]\rangle & \\
\langle h; K[\mathsf{unpack}\ (\mathsf{pack}\ v)\ \mathsf{as}\ x\ \mathsf{in}\ e]\rangle & \hookrightarrow & \langle h; K[e[v/x]]\rangle & \\
\langle h; K[\mathsf{unroll}\ (\mathsf{roll}\ v)]\rangle & \hookrightarrow & \langle h; K[v]\rangle & \\
\langle h; K[\mathsf{ref}\ v]\rangle & \hookrightarrow & \langle h \sqcup [l \mapsto v]; K[l]\rangle & (l \notin \mathrm{dom}(h)) \\
\langle h \sqcup [l \mapsto v]; K[l := v']\rangle & \hookrightarrow & \langle h \sqcup [l \mapsto v']; K[\langle\rangle]\rangle & (l \notin \mathrm{dom}(h)) \\
\langle h \sqcup [l \mapsto v]; K[!l]\rangle & \hookrightarrow & \langle h \sqcup [l \mapsto v]; K[v]\rangle & (l \notin \mathrm{dom}(h)) \\
\langle h; K[l_1 == l_2]\rangle & \hookrightarrow & \langle h; K[1]\rangle & (l_1 = l_2) \\
\langle h; K[l_1 == l_2]\rangle & \hookrightarrow & \langle h; K[0]\rangle & (l_1 \neq l_2)
\end{array}
$$

Figure 3.3: Dynamic semantics of $\mathsf{F}^{\mu!}$.

$\boxed{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}$

$$\frac{\Delta;\Gamma \vdash \sigma}{\vdash \bullet : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta;\Gamma;\sigma)}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{nat}) \quad \Delta';\Gamma' \vdash p_2 : \mathsf{nat}}{\vdash C \odot p_2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{nat})}$$

$$\frac{\Delta';\Gamma' \vdash p_1 : \mathsf{nat} \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{nat})}{\vdash p_1 \odot C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{nat})}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{nat}) \quad \Delta';\Gamma' \vdash p_1 : \sigma' \quad \Delta';\Gamma' \vdash p_2 : \sigma'}{\vdash \mathsf{ifnz}\ C\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}$$

$$\frac{\Delta';\Gamma' \vdash p_0 : \mathsf{nat} \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma') \quad \Delta';\Gamma' \vdash p_2 : \sigma'}{\vdash \mathsf{ifnz}\ p_0\ \mathsf{then}\ C\ \mathsf{else}\ p_2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}$$

$$\frac{\Delta';\Gamma' \vdash p_0 : \mathsf{nat} \quad \Delta';\Gamma' \vdash p_1 : \sigma' \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}{\vdash \mathsf{ifnz}\ p_0\ \mathsf{then}\ p_1\ \mathsf{else}\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1) \quad \Delta';\Gamma' \vdash p_2 : \sigma_2}{\vdash \langle C, p_2\rangle : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1 \times \sigma_2)}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1 \times \sigma_2)}{\vdash C.1 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1)} \qquad \frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1 \times \sigma_2)}{\vdash C.2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_2)}$$

$$\frac{\Delta';\Gamma' \vdash \sigma_2 \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1)}{\vdash \mathsf{inl}_{\sigma_2}\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1 + \sigma_2)} \qquad \frac{\Delta';\Gamma' \vdash \sigma_1 \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_2)}{\vdash \mathsf{inr}_{\sigma_1}\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1 + \sigma_2)}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1 + \sigma_2) \quad \Delta';\Gamma', x{:}\sigma_1 \vdash p_1 : \sigma' \quad \Delta';\Gamma', x{:}\sigma_2 \vdash p_2 : \sigma'}{\vdash \mathsf{case}\ C\ (x.\ p_1)\ (x.\ p_2) : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}$$

$$\frac{\Delta';\Gamma' \vdash p_0 : \sigma_1 + \sigma_2 \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma', x{:}\sigma_1;\sigma') \quad \Delta';\Gamma', x{:}\sigma_2 \vdash p_2 : \sigma'}{\vdash \mathsf{case}\ p_0\ (x.\ C)\ (x.\ p_2) : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}$$

$$\frac{\Delta';\Gamma' \vdash p_0 : \sigma_1 + \sigma_2 \quad \Delta';\Gamma', x{:}\sigma_1 \vdash p_1 : \sigma' \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma', x{:}\sigma_2;\sigma')}{\vdash \mathsf{case}\ p_0\ (x.\ p_1)\ (x.\ C) : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma', f{:}(\sigma_1 \to \sigma_2), x{:}\sigma_1;\sigma_2)}{\vdash \mathsf{fix}\ f(x{:}\sigma_1){:}\sigma_2.\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1 \to \sigma_2)}$$

Figure 3.4: Program contexts in $\mathsf{F}^{\mu!}$.

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1 \to \sigma_2) \quad \Delta';\Gamma' \vdash p_2 : \sigma_1}{\vdash C\ p_2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_2)}$$

$$\frac{\Delta';\Gamma' \vdash p_1 : \sigma_1 \to \sigma_2 \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1)}{\vdash p_1\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_2)}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta',\alpha;\Gamma';\sigma_1)}{\vdash \Lambda\alpha.\, C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\forall\alpha.\,\sigma_1)} \qquad \frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\forall\alpha.\,\sigma_1) \quad \Delta';\Gamma' \vdash \sigma_2}{\vdash C\ [\sigma_2] : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1[\sigma_2/\alpha])}$$

$$\frac{\Delta';\Gamma' \vdash \sigma_1 \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_2[\sigma_1/\alpha])}{\vdash \mathsf{pack}\ \langle\sigma_1, C\rangle\ \mathsf{as}\ \exists\alpha.\,\sigma_2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\exists\alpha.\,\sigma_2)}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\exists\alpha.\,\sigma_1) \quad \Delta',\alpha;\Gamma',x{:}\sigma_1 \vdash p_2 : \sigma_2 \quad \Delta';\Gamma' \vdash \sigma_2}{\vdash \mathsf{unpack}\ C\ \mathsf{as}\ \langle\alpha,x\rangle\ \mathsf{in}\ p_2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_2)}$$

$$\frac{\Delta';\Gamma' \vdash p_1 : \exists\alpha.\,\sigma_1 \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta',\alpha;\Gamma',x{:}\sigma_1;\sigma_2) \quad \Delta';\Gamma' \vdash \sigma_2}{\vdash \mathsf{unpack}\ p_1\ \mathsf{as}\ \langle\alpha,x\rangle\ \mathsf{in}\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_2)}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma'[\mu\alpha.\,\sigma'/\alpha])}{\vdash \mathsf{roll}_{\mu\alpha.\,\sigma'}\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mu\alpha.\,\sigma')}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mu\alpha.\,\sigma')}{\vdash \mathsf{unroll}\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma'[\mu\alpha.\,\sigma'/\alpha])}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1)}{\vdash \mathsf{ref}\ C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{ref}\ \sigma_1)} \qquad \frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{ref}\ \sigma')}{\vdash !C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{ref}\ \sigma_1) \quad \Delta';\Gamma' \vdash p_2 : \sigma_1}{\vdash C := p_2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{unit})}$$

$$\frac{\Delta';\Gamma' \vdash p_1 : \mathsf{ref}\ \sigma_1 \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma_1)}{\vdash p_1 := C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{unit})}$$

$$\frac{\vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{ref}\ \sigma') \quad \Delta';\Gamma' \vdash p_2 : \mathsf{ref}\ \sigma'}{\vdash C == p_2 : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{nat})}$$

$$\frac{\Delta';\Gamma' \vdash p_1 : \mathsf{ref}\ \sigma' \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{ref}\ \sigma')}{\vdash p_1 == C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\mathsf{nat})}$$

Figure 3.4 (cont.): Program contexts in $\mathsf{F}^{\mu!}$.

Here, *alloc* determines the choice of the fresh location.

Fortunately, it is easy to show that if two programs are contextually equivalent under all deterministic allocators (*i.e.,* according to $\sim_{\text{ctx}}^{alloc}$), then they are contextually equivalent under a non-deterministic allocator (*i.e.,* according to $\sim_{\text{ctx}}$).

Theorem 1. If

$$\Delta; \Gamma \vdash p_1 \sim_{\text{ctx}}^{alloc} p_2 : \sigma$$

for any $alloc \in \mathsf{Heap} \to \mathsf{Loc}$ satisfying $\forall h.\ alloc(h) \notin \text{dom}(h)$, then:

$$\Delta; \Gamma \vdash p_1 \sim_{\text{ctx}} p_2 : \sigma$$

Proof. We are given a heap h and a suitably typed context C and must show that $\langle h; C[p_1]\rangle\uparrow$ iff $\langle h; C[p_2]\rangle\uparrow$. We sketch the "only if" direction here, the proof of the "if" direction is symmetric.

Suppose $\langle h; C[p_1]\rangle\uparrow$. We can look at the execution trace and observe the heap locations that are allocated (and the order in which they are allocated) and choose any deterministic allocator *alloc* that implements this allocation "strategy".

Instantiating the premise then tells us that $\Delta; \Gamma \vdash p_1 \sim_{\text{ctx}}^{alloc} p_2 : \sigma$. Since $\langle h; C[p_1]\rangle$ diverges according to the *alloc*-determinized semantics (by choice of *alloc*), we learn that $\langle h; C[p_2]\rangle$ diverges according to the determinized semantics as well. It therefore also diverges according to the non-deterministic semantics, *i.e.,* $\langle h; C[p_2]\rangle\uparrow$. □

In the remainder of this chapter, we therefore assume a valid allocator *alloc* as an axiom and work with the correspondingly determinized version of $\mathsf{F}^{\mu!}$. From now on, we simply write $\sim_{\text{ctx}}$ for $\sim_{\text{ctx}}^{alloc}$ and never talk about the original $\sim_{\text{ctx}}$ again.

3.4 Global vs. Local Knowledge

Our method of parametric bisimulations (PBs) is essentially *coinductive*, following the style of previous bisimulation techniques in many respects. Coinductive reasoning makes it easy to deal with recursive features—such as recursive types and higher-order store—without requiring the use of step-indexed constructions.

The two main ways in which PBs differ from previous bisimulation techniques are in their treatment of:

Local state. From recent work on KLRs [4, 22], we borrow the idea of using state transition systems (STSs) to establish invariants on how a module's local state may evolve over time. STSs enable one to encode more flexible state invariants than are expressible using "environmental" bisimulations [74, 79].

Higher-order functions. In order to reason about higher-order functions in a coinductive style, but without confining ourselves to single-language reasoning, we employ a novel technical idea: *global vs. local knowledge.*

The treatment of local state using state transition systems follows prior work very closely, which is already discussed in Chapter 2. Here we focus attention on motivating our idea of global vs. local knowledge.

Coinductive Reasoning Recall the way we defined contextual equivalence in the previous section. An alternative but equivalent way of formulating contextual equivalence is as the *largest adequate congruence relation* [64]. Being adequate means that if two terms of base type are related, then either they both diverge (run forever) or they both evaluate to the same value (*e.g.,* if one term evaluates to 3, then the other must evaluate to 3 as well). Being a congruence means the relation is closed under all the constructs of the language (*e.g.,* if f_1 and f_2 are related at $\tau' \to \tau$, and e_1 and e_2 are related at τ', then $f_1\, e_1$ and $f_2\, e_2$ are related at τ).

To prove using coinduction that two terms e_1 and e_2 are contextually equivalent at type τ, one must exhibit a (type-indexed) term relation L that contains (τ, e_1, e_2) and then prove that L is an adequate congruence. The relation L serves as a "generalized coinduction hypothesis", by which one proves equivalence for all pairs of terms related by L simultaneously. However, while it is possible for one to employ this kind of "brute-force" coinductive proof, it is typically not very pleasant, because proving a relation to be a congruence directly can be incredibly tedious.

Bisimulation techniques help make coinductive proofs manageable by lightening the congruence proof burden. Typically, this is achieved by only requiring one to show that L is closed under type-directed *uses* (*i.e.,* evaluation or deconstruction) of the terms it relates. This results in proof obligations that look like the following (we are not being totally formal here):

(1) If $(\tau, e_1, e_2) \in L$, then either $e_1\!\uparrow$ and $e_2\!\uparrow$,
or $\exists v_1, v_2.\ e_1 \hookrightarrow^* v_1$ and $e_2 \hookrightarrow^* v_2$ and $(\tau, v_1, v_2) \in L$.

(2) If $(\mathsf{int}, v_1, v_2) \in L$, then $\exists n.\ v_1 = v_2 = n$.

(3) If $(\tau' \times \tau'', v_1, v_2) \in L$, then $\exists v_1', v_1'', v_2', v_2''.\ v_i = \langle v_i', v_i'' \rangle$
and $(\tau', v_1', v_2') \in L$ and $(\tau'', v_1'', v_2'') \in L$.

(4) If $(\mu\alpha.\,\tau, v_1, v_2) \in L$, then $\exists v_1', v_2'.\ v_i = \mathsf{roll}\ v_i'$
and $(\tau[\mu\alpha.\,\tau/\alpha], v_1', v_2') \in L$.

The most problematic proof obligation is the one for function values. It usually looks something like this (simplifying fix to λ):

(5) If $(\tau' \to \tau, v_1, v_2) \in L$, then $\exists x, e_1, e_2.\ v_i = \lambda x.e_i$ and
$\forall v_1', v_2'.\ (\tau', v_1', v_2') \in \boxed{G} \Rightarrow (\tau, e_1[v_1'/x], e_2[v_2'/x]) \in L$.

In other words, if L relates function values v_1 and v_2, then applying them to any "equivalent arguments" v_1' and v_2' should produce results that are also related by L. The big question is: what is this relation G from which the arguments v_1' and v_2' are drawn?

Global vs. Local Knowledge First, some (non-standard) terminology: There exist many equivalent terms, but when we do a bisimulation proof, we only make a claim about some of them. So let us make a distinction between "local" and "global"

knowledge about term equivalence. The relation L describes our *local knowledge*: these are the terms whose equivalence we aim to validate in our proof. The relation G, on the other hand, embodies the *global knowledge* about all terms out there that are equivalent. In proof obligation (5), we draw equivalent function arguments from G (rather than L) since they might indeed originate from "somewhere else" in the program (some unknown client code), and thus our local knowledge L may not be sufficient to justify their equivalence. This leaves us with the question of how to define G.

Whence Global Knowledge? Coming up with a sound (and practically usable) choice for G is far from obvious, and existing bisimulation methods make a variety of different choices. For example:

- *Applicative* bisimulations [1] define G to be the syntactic identity relation on closed values.

This is a nice, simple choice, which works well for pure λ-calculus. Unfortunately, for higher-order stateful languages like $\mathsf{F}^{\mu!}$, it is unsound [40], so more advanced approaches are needed:

- *Environmental* bisimulations [81, 41, 74, 79] take G to be the "context closure" of L, *i.e.,* the relation that extends the syntactic identity relation on closed values by including closures of open values v with pairs of values (w_1, w_2) that are related by L (formally: $\{(\sigma, v[\overline{w_1/y}], v[\overline{w_2/y}]) \mid \{\overline{(\sigma', w_1, w_2)}\} \subseteq L\}$).[1]

- *Normal form* (or *open*) bisimulations [43, 78, 44, 45] sidestep the whole question by choosing a fresh variable name x and representing equivalent arguments by the same x. As a result, these bisimulations are built over open terms, and proof obligation (1) above must be updated to account for the possibility that the evaluations of e_1 and e_2 get stuck trying to deconstruct the same free variable x (more about that below).

All of these methods define global knowledge in a very "syntactic" way that is well suited to proving contextual equivalences. However, we wish to develop a method that will be capable of generalizing to the setting of inter-language reasoning, where G may relate different languages. We therefore seek an account of global knowledge that is more "semantic".

Parameterizing Over Global Knowledge The essential difficulty in choosing G has to do with higher-order functions: if the argument type τ' is (or contains) a function type, then equivalence at τ' is very hard to characterize directly.[2] Our

[1] We are glossing over a lot of details here. To be precise, environmental bisimulations are actually *sets* of L's. For more details, see Section 6.2.

[2] Conversely, if τ' were arrow-free (*e.g.,* in a first-order language), it would be easy to characterize equivalence at τ' directly.

solution is simple: we don't try to define the global knowledge at all; instead, we take G to be a *parameter* of our model!

Our key observation is that it is not necessary to pin down exactly what G is, so long as we make our coinductive proof for L as parametric as possible with respect to it. (We will clarify what "as parametric as possible" means in Section 3.6.) This parametricity makes our proofs quite robust by allowing G to be instantiated in a variety of different ways. In particular, we make no assumptions whatsoever about the values that G relates at function type. For all we know, G might even include "garbage" like $(\mathsf{int} \to \mathsf{int}, 4, \langle\rangle)$.[3]

Our approach can be viewed as a more semantic account of the idea behind normal form bisimulations (see above), which is to model "equivalent arguments" as black boxes about which nothing is known. Consequently, just as for normal form bisimulations, we need to adapt proof obligation (1) above to account for the possibility that e_1 and e_2 get stuck. For normal form bisimulations, e_1 and e_2 may get stuck if they try to deconstruct a free variable x, and so normal form bisimulations loosen proof obligation (1) to allow e_1 and e_2 to reduce to terms of the form $K_1[x\ v_1]$ and $K_2[x\ v_2]$, where K_1 and K_2 are equivalent continuations and v_1 and v_2 are equivalent values. In our case, e_1 and e_2 may get stuck if they try to apply some bogus functions that are equivalent according to the global knowledge but that turn out (like 4 and $\langle\rangle$) to not even be functions. Hence, we will allow e_1 and e_2 to reduce to terms of the form $K_1[f_1\, v_1]$ and $K_2[f_2\, v_2]$, where K_1 and K_2 are equivalent continuations, and where $\{(\tau' \to \tau, f_1, f_2), (\tau', v_1, v_2)\} \subseteq G$. In this way, the parameter G serves as a semantic analogue of free variables in normal form bisimulations.

Intuitively, although the idea of parameterizing over the global knowledge may seem surprising at first, we find it to be comfortingly reminiscent of Girard's method for modeling System F [27]. In Girard's method, a potential cycle in the definition of the logical relation for impredicative universal types $\forall\alpha.\tau$ is avoided by parameterizing over an arbitrary relational interpretation of the abstract type α. In our scenario, the problem of how to define the global knowledge is avoided by parameterizing over an arbitrary relational interpretation of *function types*. In essence, we are treating a function type $\tau_1 \to \tau_2$ as an unusual kind of abstract type: the coinductive proofs about different "modules" in a program all treat the global interpretation of $\tau_1 \to \tau_2$ abstractly, while simultaneously they each contribute to defining it.

Parameterizing over the global knowledge turns out to be *very* useful. First and foremost, it makes it easy to soundly compose our coinductive proofs for different "modules" together (and hence prove soundness of our method w.r.t. contextual equivalence). Second, it enables us to reason about open terms (Section 3.5) and higher-order state invariants (Section 3.6), replacing the use of context closure or free variables for those purposes in environmental and normal form bisimulations, respectively. Finally, it is the key to establishing transitivity for our proof method (Sections 3.5.4 and 3.9).

[3]The ability to instantiate G with a "trashy" relation is surprisingly useful. We will make critical use of it in our transitivity proof.

3.5 Warmup: Parametric Bisimulations for λ^μ

To ease the presentation of parametric bisimulations (PBs), we begin in this section by using the idea of global vs. local knowledge, motivated in the previous section, to define a relational model for λ^μ, a pure fragment of $\mathsf{F}^{\mu!}$. This sub-language is obtained by restricting the definitions in Section 3.3 to base, function, product, sum, and recursive types (*i.e.,* by excluding universal, existential, and reference types, as well as the associated constructs and rules). Regarding reduction, the heap component h becomes unused and is erased. Regarding typing, the type variable environment Δ becomes unused and is erased.

Due to the simplicity of the language, the model developed here does not feature STSs. These will come into play when dealing with state in Section 3.6. However, by ignoring the transition-systems aspect of PBs for the time being, we can focus attention on other aspects of the model.

3.5.1 Definitions

Figure 3.5 lists the various semantic domains we will be using. Here, CTy denotes the set of closed types of λ^μ (*i.e.,* types with no free type variables α), CVal denotes the set of closed values (*i.e.,* values with no free term variables x), etc. The first four definitions are standard: relations on values, expressions, continuations, and heaps, indexed by the relevant types (in case of KRel, input and output types).

Next, we define what we call the *flexible* types, CTyF, along with the *flexible relations*, VRelF, which are just relations on closed values indexed by such flexible types. Whereas bisimulation methods typically allow terms of arbitrary type to be included in the bisimulation, we find it useful to restrict local and global knowledges to relate only values of "flexible" types. Intuitively, these are the types at which value equivalence may depend on module-specific knowledge. In $\mathsf{F}^{\mu!}$, there will be several kinds of flexible types, but in λ^μ, the only flexible types are function types.

In contrast, value equivalence at the remaining types—base, product, sum, and recursive types, which we call *rigid*—is fixed and agreed upon by all modules once the meaning of the flexible types is defined. This is achieved by a closure operation that takes a relation $R \in \mathsf{VRelF}$ and returns its value closure $\overline{R} \in \mathsf{VRel}$. It is defined as the least fixed-point of the set of equations in Figure 3.6. For instance, two pairs are related by $\overline{R}$ if and only if they are related componentwise. Note that R only occurs covariantly[4], so $\overline{R}$ is inductively well defined, even though the type gets bigger on the r.h.s. in the case of $\mu\alpha.\tau$.

Local and Global Knowledge As shown in Figure 3.7, a local knowledge $L \in \mathrm{LK}$ is essentially a flexible relation, except that this relation is actually itself parameterized by the global knowledge, G. In effect, $L(G)$ describes the values that we wish

[4]We don't need to worry about functions introducing contravariance because the relation on function types is already given by R itself.

$$
\begin{array}{lcl}
\mathsf{VRel} & := & \mathsf{CTy} \to \mathcal{P}(\mathsf{CVal} \times \mathsf{CVal}) \\
\mathsf{ERel} & := & \mathsf{CTy} \to \mathcal{P}(\mathsf{CExp} \times \mathsf{CExp}) \\
\mathsf{KRel} & := & \mathsf{CTy} \times \mathsf{CTy} \to \mathcal{P}(\mathsf{CCont} \times \mathsf{CCont}) \\
\mathsf{HRel} & := & \mathcal{P}(\mathsf{Heap} \times \mathsf{Heap}) \\
\mathsf{CTyF} & := & \{(\tau_1 \to \tau_2) \in \mathsf{CTy}\} \\
\mathsf{VRelF} & := & \mathsf{CTyF} \to \mathcal{P}(\mathsf{CVal} \times \mathsf{CVal})
\end{array}
$$

Figure 3.5: Semantic domains for λ^μ.

$$
\begin{array}{lcl}
\overline{R}(\tau) & := & R(\tau) \qquad \text{if } \tau \in \mathsf{CTyF} \\
\overline{R}(\mathsf{unit}) & := & \{(\langle\rangle, \langle\rangle)\} \\
\overline{R}(\mathsf{nat}) & := & \{(n, n)\} \\
\overline{R}(\tau_1 \times \tau_2) & := & \{(\langle v_1, v_1'\rangle, \langle v_2, v_2'\rangle) \mid (v_1, v_2) \in \overline{R}(\tau_1) \wedge (v_1', v_2') \in \overline{R}(\tau_2)\} \\
\overline{R}(\tau_1 + \tau_2) & := & \{(\mathsf{inl}\ v_1, \mathsf{inl}\ v_2) \mid (v_1, v_2) \in \overline{R}(\tau_1)\} \cup \{(\mathsf{inr}\ v_1, \mathsf{inr}\ v_2) \mid (v_1, v_2) \in \overline{R}(\tau_2)\} \\
\overline{R}(\mu\alpha.\,\tau) & := & \{(\mathsf{roll}\ v_1, \mathsf{roll}\ v_2) \mid (v_1, v_2) \in \overline{R}(\tau[\mu\alpha.\,\tau/\alpha])\}
\end{array}
$$

Figure 3.6: Value closure for λ^μ (if $R \in \mathsf{VRelF}$, then $\overline{R} \in \mathsf{VRel}$).

$$
\begin{array}{lcl}
\mathit{step}(e) & := & \begin{cases} e' & \text{if } e \hookrightarrow e' \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{FixVal} & := & \{f \in \mathsf{CVal} \mid \forall v.\ \mathit{step}(f\ v) \neq \bot\} \\
R' \supseteq R & := & \forall \tau.\ R'(\tau) \supseteq R(\tau) \\
 & & \\
\mathrm{LK} & := & \{L \in \mathsf{VRelF} \to \mathsf{VRelF} \mid L \text{ is monotone w.r.t. } \subseteq\ \wedge \\
 & & \qquad\qquad \forall G, \tau.\ L(G)(\tau) \subseteq \mathrm{FixVal} \times \mathrm{FixVal}\} \\
\mathsf{GK}(L) & := & \{G \in \mathsf{VRelF} \mid G \supseteq L(G)\}
\end{array}
$$

Figure 3.7: Specification of local and global knowledges for λ^μ.

to prove are equivalent, assuming that G correctly represents the global knowledge. This parameterization is necessary in order to reason about open terms, and we will see its utility below in the proof of compatibility for fix. We require that L is monotone w.r.t. G: intuitively, passing in a larger global knowledge should never result in fewer terms being related by L.

We also require that the values related by the local knowledge at function type are indeed functions, in the sense that their application to an arbitrary value should not be stuck, but should be reducible at least for one step. This is a technical requirement that is used in our transitivity proof in Section 3.5.4. The idea is that there should be classes of values that may be related by the global knowledge but not by the local knowledge.

We want to restrict attention to global knowledges that are closed w.r.t. the local knowledge in question: For a particular L, we define $\mathsf{GK}(L)$ to be the set of flexible relations G s.t. $G \supseteq L(G)$. This requirement makes sense since the global knowledge must by definition be a superset of any local knowledge. Observe, however, that we do not restrict what other values G relates. Indeed, G may relate values at function type that are not actually functions, or that are obviously inequivalent (*e.g.,* 4 and $\langle\rangle$, cf. Section 3.4).

Relating Expressions and Continuations So far, we have seen how local knowledges describe equivalence between values of flexible type, and how this equivalence can be lifted to arbitrary types.

Figure 3.8 shows how equivalence is defined for (closed) expressions, e, and continuations, K. Specifically, we introduce two new relations, $\mathbf{E} \in \mathsf{VRelF} \to \mathsf{ERel}$ and $\mathbf{K} \in \mathsf{VRelF} \to \mathsf{KRel}$, which are defined coinductively.

Given a type τ, a local knowledge $L \in \mathrm{LK}$, and a global knowledge $G \in \mathsf{GK}(L)$, we say that two expressions are "locally" equivalent, written $(e_1, e_2) \in \mathbf{E}(G)(\tau)$, if they either both diverge or both terminate producing related values. Along the way, however, they may make calls to "external" functions, that is, functions that are related by G, but not necessarily by the local knowledge $L(G)$. More precisely, we say two closed expressions are locally equivalent if and only if one of the following three cases holds:

Case DIV. Both expressions diverge (run forever).

Case EVAL. Both expressions run successfully to completion, producing related values.

Case CALL. Both expressions reduce after some number of steps to some expressions of the form $K_i[f_i\, v_i]$, where both the f_i and v_i are related by the global knowledge G at the appropriate types, and the continuations, K_1 and K_2, are equivalent. We say that two continuations are equivalent if instantiating them with equivalent values (according to the global knowledge G) yields equivalent expressions.

$$
\begin{array}{lcl}
\mathbf{E}(G)(\tau) & := & \{(e_1, e_2) \mid \\
& & \quad (\text{DIV}) \quad e_1{\uparrow} \wedge e_2{\uparrow} \\
& & \vee \quad (\text{EVAL}) \quad \exists v_1, v_2.\ e_1 \hookrightarrow^* v_1 \wedge e_2 \hookrightarrow^* v_2 \wedge (v_1, v_2) \in \overline{G}(\tau) \\
& & \vee \quad (\text{CALL}) \quad \exists \tau', K_1, K_2, e_1', e_2'. \\
& & \qquad\qquad e_1 \hookrightarrow^* K_1[e_1'] \wedge e_2 \hookrightarrow^* K_2[e_2'] \wedge \\
& & \qquad\qquad (e_1', e_2') \in \mathbf{U}(G, G)(\tau') \wedge \\
& & \qquad\qquad (K_1, K_2) \in \mathbf{K}(G)(\tau', \tau)\} \\
\\
\mathbf{K}(G)(\tau_1, \tau_2) & := & \{(K_1, K_2) \mid \forall (v_1, v_2) \in \overline{G}(\tau_1).\ (K_1[v_1], K_2[v_2]) \in \mathbf{E}(G)(\tau_2)\} \\
\\
\mathbf{U}(R, R')(\tau) & := & \{(v_1\ v_1', v_2\ v_2') \mid \exists \tau'.\ (v_1, v_2) \in R(\tau' \to \tau) \wedge (v_1', v_2') \in \overline{R'}(\tau')\} \\
\\
\mathsf{consistent}(L) & := & \forall G \in \mathsf{GK}(L).\ \forall \tau.\ \forall (e_1, e_2) \in \mathbf{U}(L(G), G)(\tau). \\
& & (\mathit{step}(e_1), \mathit{step}(e_2)) \in \mathbf{E}(G)(\tau) \\
\\
\Gamma \vdash e_1 \sim_L e_2 : \tau & := & \mathsf{consistent}(L) \ \wedge\ \forall G \in \mathsf{GK}(L).\ \forall \gamma_1, \gamma_2 \in \mathrm{dom}(\Gamma) \to \mathsf{CVal}. \\
& & (\forall x{:}\tau' \in \Gamma.\ (\gamma_1 x, \gamma_2 x) \in \overline{G}(\tau')) \implies (\gamma_1 e_1, \gamma_2 e_2) \in \mathbf{E}(G)(\tau) \\
\\
\Gamma \vdash e_1 \sim e_2 : \tau & := & \exists L.\ \ \Gamma \vdash e_1 \sim_L e_2 : \tau \\
\\
\Gamma \vdash p_1 \sim p_2 : \tau & := & \Gamma \vdash |p_1| \sim |p_2| : \tau
\end{array}
$$

Figure 3.8: Mutually coinductive definitions of expression equivalence, $\mathbf{E} \in \mathsf{VRelF} \to \mathsf{ERel}$, and continuation equivalence, $\mathbf{K} \in \mathsf{VRelF} \to \mathsf{KRel}$, and definitions of consistency and program equivalence for λ^{μ}.

Intuitively, expressions that are locally equivalent w.r.t global knowledge G do not necessarily have the same observable behavior, but they can be understood to have the same local behavior, *i.e.,* behave equivalently modulo what happens during calls to functions related by G. If G happens to relate functions that behave differently, this is G's fault, not ours.

Consistency and Program Equivalence We say that a local knowledge L is consistent (in Figure 3.8) if and only if any functions that it declares equivalent yield in fact equivalent expressions when applied to equivalent arguments. In the formal definition, we parameterize over an arbitrary global knowledge $G \in \mathsf{GK}(L)$; the auxiliary construction $\mathbf{U}$ (characterizing the "uses" of functions) draws the functions being tested for equivalence from the local knowledge, $L(G)$, while the arguments to which they are applied are drawn from the global knowledge, G.

In order for PB's coinductive reasoning to be sound, we must be quite careful in the definition of consistency: for each pair of functions (v_1, v_2) related by $L(G)$, and arguments (v_1', v_2') related by $\overline{G}$, we cannot simply require $(v_1\ v_1', v_2\ v_2')$ to be related by $\mathbf{E}(G)$ because, via CALL, this is a tautology! Instead, we demand that $v_i\ v_i' \hookrightarrow e_i$, and that e_1 and e_2 are related by $\mathbf{E}(G)$ (see the use of *step* in the formal definition). Requiring the terms to take a step of reduction at this point ensures that "progress" is made in the coinductive argument, *i.e.,* that the coinduction is *guarded*. We will say more about this guardedness condition in Section 3.10.

We call two expressions equivalent at type τ in context Γ, written $\Gamma \vdash e_1 \sim e_2 : \tau$, if and only if there exists a consistent local knowledge L that shows that $\gamma_1 e_1$ and $\gamma_2 e_2$ are equivalent at type τ for value substitutions γ_1 and γ_2 and an arbitrary global knowledge G for L. (We write γe for substituting in e according to γ.) These substitutions are also arbitrary except that they must be pointwise related by G at Γ.

Finally, two programs are equivalent simply if and only if their type-erased versions are equivalent expressions.

A note on coinduction. As $\mathbf{E}$ and $\mathbf{K}$ are defined mutually dependent over a complete lattice (powerset lattice lifted pointwise to function spaces) and all operations involved are monotone, we can take the meaning of these definitions to be either the least or the greatest fixed point. We choose the greatest fixed point, corresponding to coinduction, because this can in principle relate more terms and is somewhat easier to work with. This means that PBs actually come with *two* coinductive reasoning principles:

1. The first is enabled by $\mathbf{E}$'s CALL case and the use of the global knowledge, which contains the local knowledge. Here, the coinductive assumption is about L (and thus G). This coinduction principle is used all the time in examples, for instance in Section 3.5.2 below. For a quick intuition, think of reasoning about recursive functions: we construct a local knowledge relating them and then need to show its consistency. In the proof of consistency, after taking a

step, we may get to recursive calls. At that point, we are allowed to make use of the CALL since we already know that the functions are related by the global knowledge (after all, it contains the local knowledge that we constructed).

2. The second is enabled by the definition of **E** as a greatest fixed point, as we just discussed above. Here, the coinductive assumption is about **E**. This coinduction principle is only used in the metatheory (*e.g.,* Lemma 3 and its generalization Lemma 11), not in any example proofs. The reason is that in λ^{μ} and $\mathsf{F}^{\mu!}$ there is no way for a program to loop other than via function calls, so we can always use the first principle. Were we to extend the language with other forms of recursion (such as while loops or primitive recursion), the coinductive interpretation of **E** would become essential.

When presenting PILS in Chapter 4, we will see that it is possible to get away with a single form of coinductive reasoning, without losing expressivity.

3.5.2 Example Proof

Consider the following example concerning streams as functions (taken from Sumii and Pierce [81]):

$$\begin{array}{rcl} \tau & := & \mu\alpha.\,\mathsf{unit} \to \mathsf{nat} \times \alpha \\ \mathit{ones} : \mathsf{unit} \to \mathsf{nat} \times \tau & := & \mathsf{fix}\, f(x).\, \langle 1, \mathsf{roll}\ f\rangle \\ \mathit{twos} : \mathsf{unit} \to \mathsf{nat} \times \tau & := & \mathsf{fix}\, f(x).\, \langle 2, \mathsf{roll}\ f\rangle \\ \mathit{succ} : \tau \to \tau & := & \mathsf{fix}\, f(s).\, \mathsf{let}\ \langle n, s'\rangle = \mathsf{unroll}\ s\ \langle\rangle\ \mathsf{in} \\ & & \quad \mathsf{roll}\ \lambda x.\, \langle n{+}1, f\ s'\rangle \end{array}$$

(Here and elsewhere we write $\lambda x.\, e$ short for $\mathsf{fix}\, g(x).\, e$, where g does not occur free in e.) As the names suggest, *ones* encodes the stream consisting of only 1's (*i.e.,* $111\ldots$), while *twos* encodes the stream consisting of only 2's. The goal is to show that incrementing each component of *ones* by one results in a stream equivalent to *twos*:

$$\vdash \mathsf{roll}\ \mathit{twos} \sim \mathit{succ}\ (\mathsf{roll}\ \mathit{ones}) : \tau$$

Constructing a Suitable Local Knowledge. Note that we have

$$\mathit{succ}\ (\mathsf{roll}\ \mathit{ones}) \hookrightarrow^{*} \mathsf{roll}\ \mathit{twos}'$$

for $\mathit{twos}' := \lambda x.\, \langle 1{+}1, \mathit{succ}\ (\mathsf{roll}\ \mathit{ones})\rangle$. We define a local knowledge L that relates exactly *twos* and twos':

$$L(G)(\tau') := \{(\mathit{twos}, \mathit{twos}') \mid \tau' = \mathsf{unit} \to \mathsf{nat} \times \tau\}$$

Proving Its Consistency. Suppose $(e_1, e_2) \in \mathbf{U}(L(G), G)(\tau')$, where $G \in \mathsf{GK}(L)$. By construction of L we know that $e_1 = \mathit{twos}\ \langle\rangle$, $e_2 = \mathit{twos}'\ \langle\rangle$, and $\tau' = \mathsf{nat} \times \tau$. Hence we must show:

$$(\langle 2, \mathsf{roll}\ \mathit{twos}\rangle, \langle 1{+}1, \mathit{succ}\ (\mathsf{roll}\ \mathit{ones})\rangle) \in \mathbf{E}(G)(\mathsf{nat} \times \tau)$$

Using the EVAL case in $\mathbf{E}$ and the definition of $\overline{G}$, this reduces to showing $(2, 2) \in \overline{G}(\mathsf{nat})$ and $(\mathsf{roll}\ \mathit{twos}, \mathsf{roll}\ \mathit{twos}') \in \overline{G}(\tau)$. The former is trivial. The latter is equivalent to $(\mathit{twos}, \mathit{twos}') \in \overline{G}(\mathsf{unit} \to \mathsf{nat} \times \tau)$, which holds by construction because G extends $L(G)$.

Showing the Programs Related By It. It remains to show:

$$\forall G \in \mathsf{GK}(L).\ (\mathsf{roll}\ \mathit{twos}, \mathit{succ}\ (\mathsf{roll}\ \mathit{ones})) \in \mathbf{E}(G)(\tau)$$

Again using EVAL, we end up having to show $(\mathsf{roll}\ \mathit{twos}, \mathsf{roll}\ \mathit{twos}') \in \overline{G}(\tau)$, which we have already done above.

3.5.3 Basic Properties and Soundness

We move on to some properties of our constructions. These lemmas will be generalized in a later section (3.7) to the full $\mathsf{F}^{\mu!}$ setting.

The first lemma says that a global knowledge that is valid for the union of two local knowledges is also valid for each local knowledge in isolation. (The reason why we don't express this more generally using $\subseteq$ will become clear in Section 3.7.)

Lemma 1. If $L_1, L_2 \in \mathrm{LK}$ and $G \in \mathsf{GK}(L_1 \cup L_2)$, then $G \in \mathsf{GK}(L_1) \cap \mathsf{GK}(L_2)$.

The following lemma states that consistency of local knowledges is preserved under (pointwise) union. This is important for ensuring that equivalence proofs for different subterms, which rely on different local knowledges, can be soundly composed.

Lemma 2. If $\mathsf{consistent}(L)$ and $\mathsf{consistent}(L')$, then $\mathsf{consistent}(L \cup L')$.

Figure 3.9 shows some of the basic properties of our program equivalence relation. First, we have a set of rules stating that equivalence is compatible with all the language constructs. These rules state that if two terms start with the same term constructor and their immediate subterms are component-wise equivalent, then so are the composite terms. For brevity, we just present the rules for recursive function definition (FIX) and function application (APP), whose proofs are the most interesting.

The proof of FIX is particularly interesting because it requires coinductive reasoning: when showing the consistency of our local knowledge, *i.e.,* when proving applications of the recursive functions related, we deal with the recursion by resorting to the fact that the local knowledge relates the functions.

$$\frac{\Gamma, f{:}(\tau' \to \tau), x{:}\tau' \vdash e_1 \sim e_2 : \tau}{\Gamma \vdash \mathsf{fix}\, f(x).\, e_1 \sim \mathsf{fix}\, f(x).\, e_2 : \tau' \to \tau}\ \textsc{Fix}$$

$$\frac{\Gamma \vdash e_1 \sim e_2 : \tau' \to \tau \qquad \Gamma \vdash e_1' \sim e_2' : \tau'}{\Gamma \vdash e_1\ e_1' \sim e_2\ e_2' : \tau}\ \textsc{App}$$

$$\frac{\Gamma \vdash p : \tau}{\Gamma \vdash |p| \sim |p| : \tau}\ \textsc{Refl} \qquad \frac{\Gamma \vdash e_2 \sim e_1 : \tau}{\Gamma \vdash e_1 \sim e_2 : \tau}\ \textsc{Symm}$$

$$\frac{\Gamma \vdash e_1 \sim e_2 : \tau \qquad \Gamma \vdash e_2 \sim e_3 : \tau}{\Gamma \vdash e_1 \sim e_3 : \tau}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash e_1 \sim e_2 : \tau \qquad \vdash C : (\Gamma;\tau) \rightsquigarrow (\Gamma';\tau')}{\Gamma' \vdash |C|[e_1] \sim |C|[e_2] : \tau'}\ \textsc{Cong}$$

$$\frac{\Gamma, x{:}\tau' \vdash e_1 \sim e_2 : \tau \qquad \Gamma \vdash v_1 \sim v_2 : \tau'}{\Gamma \vdash e_1[v_1/x] \sim e_2[v_2/x] : \tau}\ \textsc{Subst}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1' \sim e_2' : \tau \\ \forall \gamma \in \mathrm{dom}(\Gamma) \to \mathsf{CVal}.\ \gamma e_1 \hookrightarrow^* \gamma e_1' \qquad \forall \gamma \in \mathrm{dom}(\Gamma) \to \mathsf{CVal}.\ \gamma e_2 \hookrightarrow^* \gamma e_2'\end{array}}{\Gamma \vdash e_1 \sim e_2 : \tau}\ \textsc{Expand}$$

$$\frac{\Gamma, x{:}\tau' \vdash e_1 \sim e_2 : \tau \qquad \Gamma \vdash v_1 \sim v_2 : \tau'}{\Gamma \vdash (\lambda x.\, e_1)\ v_1 \sim e_2[v_2/x] : \tau}\ \textsc{Beta}$$

Figure 3.9: Some basic properties of our equational model.

More formally, from the premise there exists L such that $\Gamma, f{:}(\tau' \to \tau), x{:}\tau' \vdash e_1 \sim_L e_2 : \tau$ and $\mathsf{consistent}(L)$. We define:

$$\begin{aligned} L'(G)(\tau_1) := \{(\gamma_1 \mathsf{fix}\, f(x).\, e_1, \gamma_2 \mathsf{fix}\, f(x).\, e_2) \mid \\ \tau_1 = \tau' \to \tau \wedge \gamma_i \in \mathrm{dom}(\Gamma) \to \mathsf{CVal} \wedge \\ \forall y{:}\tau_2 \in \Gamma.\ (\gamma_1(y), \gamma_2(y)) \in \overline{G}(\tau_2)\} \end{aligned}$$

Note here how the parameterization of L' over G provides it with a source from which to construct the closing substitutions γ_1 and γ_2.

The goal now is to prove

$$\Gamma \vdash \mathsf{fix}\, f(x).\, e_1 \sim \mathsf{fix}\, f(x).\, e_2 : \tau' \to \tau.$$

The local knowledge that we choose is $L \cup L'$. Showing that L' (and thus $L \cup L'$) relates any appropriately closed instances of the two functions is simply a matter of unfolding definitions (by construction of L'). It therefore remains to establish $\mathsf{consistent}(L \cup L')$. By Lemma 2, this boils down to showing

$$(\gamma_1' e_1, \gamma_2' e_2) \in \mathbf{E}(G)(\tau)$$

for any $G \in \mathsf{GK}(L \cup L')$, where

- $\gamma_i \in \mathrm{dom}(\Gamma) \to \mathsf{CVal}$,
- $\forall y{:}\tau' \in \Gamma.\ (\gamma_1(y), \gamma_2(y)) \in \overline{G}(\tau')$,
- $(v_1, v_2) \in \overline{G}(\tau')$,
- $\gamma_i' = \gamma_i, f{\mapsto}(\gamma_i \mathsf{fix}\, f(x).\, e_i), x{\mapsto}v_i$.

Since $(\gamma_1 \mathsf{fix}\, f(x).\, e_1, \gamma_2 \mathsf{fix}\, f(x).\, e_2) \in L'(G)(\tau' \to \tau) \subseteq \overline{G}(\tau' \to \tau)$ and by Lemma 1 $G \in \mathsf{GK}(L)$, we can instantiate $\Gamma, f{:}\tau' \to \tau, x{:}\tau' \vdash e_1 \sim_L e_2 : \tau$ with γ_i' and are done.

The proof of rule App relies on Lemma 2 as well, in order to show that the consistent local knowledges for its two premises combine to form a consistent local knowledge for the conclusion. In addition, the proof relies on the following lemma about plugging equivalent expressions or continuations into equivalent continuations, proved by mutual coinduction and case analysis.

Lemma 3. If $(K_1, K_2) \in \mathbf{K}(G)(\tau', \tau)$, then:

1. $(e_1, e_2) \in \mathbf{E}(G)(\tau') \implies (K_1[e_1], K_2[e_2]) \in \mathbf{E}(G)(\tau)$
2. $(K_1', K_2') \in \mathbf{K}(G)(\tau'', \tau') \implies (K_1[K_1'], K_2[K_2']) \in \mathbf{K}(G)(\tau'', \tau)$

In the proof of App, we apply the first case of this lemma with $e_i := \gamma_i\, e_i$ and $K_i := \bullet\ \gamma_i e_i'$, which leaves us to prove K_1 and K_2 to be equivalent according to $\mathbf{K}$. Unfolding the definition of $\mathbf{K}$, we have to show that for arbitrary equivalent values v_1 and v_2, $(v_1\ \gamma_1 e_1', v_2\ \gamma_2 e_2')$ is in $\mathbf{E}$, for which we apply Lemma 3 again, this time

with $e_i := \gamma_i\, e_i'$ and $K_i := v_i\ \bullet$. Then we are left to prove $v_1\ \bullet$ and $v_2\ \bullet$ equivalent, *i.e.,* that $(v_1\ v_1', v_2\ v_2')$ is in $\mathbf{E}$ for arbitrary equivalent values v_1' and v_2', which follows from the third disjunct of the $\mathbf{E}$ definition.

As a consequence of these "compatibility" rules, by a straightforward induction on the typing derivation, we can show that equivalence is reflexive on well-typed programs (rule Refl). This corresponds to the "fundamental property" of logical relations. Equivalence is also symmetric: this follows trivially from the symmetric nature of our definition. Transitivity holds as well, but its much more subtle and warrants its own section, namely Section 3.5.4.

By induction on the typing derivation of contexts, we can show that our equivalence is a congruence: if two equivalent terms are placed in the same contexts, the resulting compositions are equivalent. Next, we have a substitutivity property for values, an expansion law for pure execution steps, and finally a direct corollary of these two, namely β-equivalence (on value arguments).

We move to a key lemma about $\mathbf{E}$. Given a consistent local knowledge L, if the global knowledge extends L with some additional external knowledge $\boldsymbol{\mathcal{R}}$, then the call case in the definition of $\mathbf{E}$ can be restricted so that it applies only to *external* function calls (*i.e.,* calls to functions related by $\boldsymbol{\mathcal{R}}$).

Lemma 4 (External call)**.** For any $\mathsf{consistent}(L)$, any $G \in \mathsf{GK}(L)$ and $\boldsymbol{\mathcal{R}} \in \mathsf{VRelF}$, we have:

$$G = L(G) \cup \boldsymbol{\mathcal{R}} \implies \mathbf{E}(G) = \mathbf{E}^{\boldsymbol{\mathcal{R}}}(G)$$

where the definition of $\mathbf{E}^{\boldsymbol{\mathcal{R}}}$ is the same as $\mathbf{E}$ except that, in the third disjunct, $\mathbf{U}(G, G)$ is replaced by $\mathbf{U}(\boldsymbol{\mathcal{R}}, G)$.

The $\supseteq$ follows directly from the observation that $\boldsymbol{\mathcal{R}} \subseteq G$. To prove the other direction, we essentially have to eliminate all uses of the third disjunct of $\mathbf{E}$ where the functions being invoked are related by $G \setminus \boldsymbol{\mathcal{R}}$. Since all such functions are by definition in $L(G)$, and since we know $\mathsf{consistent}(L)$, we can in fact always "inline" the equivalence proofs for all such function calls.

For $G = \mu R.L(R)$ (denoting the least fixed point of L) and $\boldsymbol{\mathcal{R}} = \emptyset$, Lemma 4 yields *adequacy*, which says that equivalent closed terms either both diverge or both terminate returning proper values. Note that this implies safety: the terms never get stuck during evaluation.

Lemma 5 (Adequacy)**.** If $\vdash e_1 \sim e_2 : \tau$, then:

$$(e_1{\uparrow} \wedge e_2{\uparrow}) \ \vee\ \exists v_1, v_2.\ e_1 \hookrightarrow^* v_1 \wedge e_2 \hookrightarrow^* v_2$$

Finally, combining adequacy and congruence, we show our main soundness theorem: for well-typed programs, our equivalence relation is included in contextual equivalence.

Theorem 2 (Soundness)**.** If $\Gamma \vdash p_1 : \tau$ and $\Gamma \vdash p_2 : \tau$, then:

$$\Gamma \vdash p_1 \sim p_2 : \tau \quad\implies\quad \Gamma \vdash p_1 \sim_{\mathrm{ctx}} p_2 : \tau$$

3.5.4 Transitivity

We now briefly sketch our proof of transitivity. We will go into more details in Section 3.9 presenting the proof of transitivity of our PB model for full $\mathsf{F}^{\mu!}$.

It suffices to show the following lemma:

Lemma 6. If $\Gamma \vdash e_1 \sim_{L_1} e_2 : \tau$ and $\Gamma \vdash e_2 \sim_{L_2} e_3 : \tau$, then there exists L such that $\Gamma \vdash e_1 \sim_{L} e_3 : \tau$.

Naturally, we can expect L to be some sort of composition of the given local knowledges L_1 and L_2. Defining this composition is, however, quite subtle. The problem is that the local knowledge takes the global knowledge G as a parameter, but then what global knowledges G_{L_1} and G_{L_2} should be passed on to L_1 and L_2, respectively, in constructing L? Assuming for now we somehow pick G_{L_1} and G_{L_2} appropriately, L can naturally be defined as follows:

$$L(G)(\tau) := L_1(G_{L_1})(\tau) \circ L_2(G_{L_2})(\tau)$$

where $\circ$ stands for ordinary relational composition.

The key part of the proof is showing transitivity of $\mathbf{E}$:

$$\forall G \in \mathsf{GK}(L).\ (e_1', e_2') \in \mathbf{E}(G_{L_1})(\tau) \wedge (e_2', e_3') \in \mathbf{E}(G_{L_2})(\tau) \implies (e_1', e_3') \in \mathbf{E}(G)(\tau)$$

In order to prove this, we want the disjunct of $\mathbf{E}$ by which e_1' and e_2' are related to match the disjunct of $\mathbf{E}$ by which e_2' and e_3' are related (recall the three disjuncts in the definition of $\mathbf{E}$). To illustrate, say e_1' and e_2' are related because they reduce to related values (second disjunct). Now consider the three cases regarding e_2' and e_3':

Case DIV. They both diverge. Fortunately, this contradicts our assumption about e_2', so this case cannot arise.

Case EVAL. They are related for the same reason as e_1' and e_2' are—*i.e.,* e_2' and e_3' reduce to related values. This is the "good" case. Relying on determinacy of reduction, we are done if we can show transitivity of the value relation. Formally, we need to show that $\overline{G_{L_1}}(\tau) \circ \overline{G_{L_2}}(\tau) \subseteq \overline{G}(\tau)$.

Case CALL. They reduce to related function calls with related continuations. It is unclear how to make progress in this situation, so we would very much like to rule it out!

In order to make case EVAL straightforward to show, while simultaneously ruling out case CALL from consideration, we will need to define G_{L_1} and G_{L_2} carefully.

The key idea is as follows: for each pair of function values $(f_1, f_3) \in G$, we come up with a value, v_2, that (i) uniquely identifies f_1 and f_3, and that (ii) is not a normal function, but rather a "bad" value that gets stuck when applied to an argument (and hence does not belong to FixVal). The first requirement allows us to ensure $G_{L_1}(\tau) \circ G_{L_2}(\tau) = G(\tau)$, as needed in proving transitivity of the value relation, by

$$\begin{array}{lcl} \tau \in \mathsf{Ty} & ::= & \ldots \mid \mathbf{n} \\ \mathsf{CTyF} & := & \{(\tau_1 \to \tau_2) \in \mathsf{CTy}\} \cup \{(\forall\alpha.\,\tau) \in \mathsf{CTy}\} \cup \{(\mathsf{ref}\,\tau) \in \mathsf{CTy}\} \cup \{\mathbf{n} \in \mathsf{CTy}\} \end{array}$$

Figure 3.10: Semantic domains for $\mathsf{F}^{\mu!}$.

relating $(f_1, v_2) \in G_{L_1}$ and $(v_2, f_3) \in G_{L_2}$. The second requirement, together with Lemma 4, rules out the "bad" case CALL above.

Formally, since CTy and CVal are countable sets, there exists an injective function $\mathbf{I} \in \mathsf{CTy} \times \mathsf{CTy} \times \mathsf{CVal} \times \mathsf{CVal} \to \mathsf{CVal}$ (*e.g.,* returning an integer value that uniquely encodes all the arguments). Using this function, one can decompose $G \in \mathsf{VRelF}$ as follows:

$$\begin{array}{rcl} G_{(1)}(\tau_1 \to \tau_2) & := & \{(f_1, \mathbf{I}(\tau_1, \tau_2, f_1, f_3)) \mid (f_1, f_3) \in G(\tau_1 \to \tau_2)\} \\ G_{(2)}(\tau_1 \to \tau_2) & := & \{(\mathbf{I}(\tau_1, \tau_2, f_1, f_3), f_3) \mid (f_1, f_3) \in G(\tau_1 \to \tau_2)\} \end{array}$$

Taking G_{L_1} to be $G_{(1)}$ is, however, incorrect because if L_1 relates any values at function type, then the global knowledge will not be closed w.r.t. it (*i.e.,* $G_{(1)} \notin \mathsf{GK}(L_1)$). To address this problem, we simply close $G_{(1)}$ accordingly, *i.e.,* we take G_{L_1} to be the least solution to the fixed-point equation $G_{L_1} = L_1(G_{L_1}) \cup G_{(1)}$ (and similarly for G_{L_2}).

With these definitions, we can show $\overline{G_{L_1}}(\tau) \circ \overline{G_{L_2}}(\tau) = \overline{G}(\tau)$ if $G \in \mathsf{GK}(L)$, and use it in case EVAL above.

In the problematic case CALL, we make use of the key Lemma 4, which lets us conclude that the functions being called—say, (f_2, f_3)—are *really* external, *i.e.,* related not merely by G_{L_2} but more precisely by $G_{(2)}$. But by construction, this means that f_2 is an integer and thus e_2' gets stuck, contradicting the prior assumption that e_1' and e_2' reduce to values.

3.6 Parametric Bisimulations for $\mathsf{F}^{\mu!}$

In this section, we present the full-blown parametric bisimulations for $\mathsf{F}^{\mu!}$. This model generalizes the model from the previous section in a superficially very simple way: whereas previously we proved two terms equivalent by exhibiting a consistent local knowledge L, we now do so by exhibiting a consistent *world* W.

3.6.1 Worlds

Worlds are state transition systems (equipped with "public" and "private" transitions, just as described in Section 2.2) that control how the local knowledge of a module and the properties of its local state may evolve over time. Formally (Figure 3.12), a transition system T consists of a (possibly infinite) state space (S), the private (or full) transition relation ($\sqsubseteq$) and a smaller public transition relation ($\sqsubseteq_{\text{pub}}$),

$$
\begin{array}{lcll}
\overline{R}(\tau) & := & R(\tau) & \text{if } \tau \in \mathsf{CTyF} \\
 & \vdots & & \\
\overline{R}(\exists\alpha.\,\tau) & := & \{(\mathsf{pack}\ v_1, \mathsf{pack}\ v_2) \mid \exists\tau'.\ (v_1, v_2) \in \overline{R}(\tau[\tau'/\alpha])\} &
\end{array}
$$

Figure 3.11: Value closure for $\mathsf{F}^{\mu!}$ (if $R \in \mathsf{VRelF}$, then $\overline{R} \in \mathsf{VRel}$).

$$
step(e) := \begin{cases} e' & \text{if } \forall h.\ \langle h; e\rangle \hookrightarrow \langle h; e'\rangle \\ \bot & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
\mathrm{FixVal} := \{v \in \mathsf{CVal} \mid \forall v'.\ step(v\ v') \neq \bot\} \\
\mathrm{GenVal} := \{v \in \mathsf{CVal} \mid step(v\ []) \neq \bot\}
\end{array}
$$

$$
\begin{array}{l}
T \in \mathsf{TrSys} := \{(\mathsf{S}, \sqsupseteq, \sqsupseteq_{\mathrm{pub}}) \in \mathsf{Set} \times \mathcal{P}(\mathsf{S}\times\mathsf{S}) \times \mathcal{P}(\mathsf{S}\times\mathsf{S}) \mid \\
\quad (\forall s, s'.\ s' \sqsupseteq_{\mathrm{pub}} s \implies s' \sqsupseteq s) \wedge \\
\quad (\forall s.\ s \sqsupseteq_{\mathrm{pub}} s) \wedge \\
\quad (\forall s_1, s_2, s_3.\ s_3 \sqsupseteq s_2 \wedge s_2 \sqsupseteq s_1 \implies s_3 \sqsupseteq s_1) \wedge \\
\quad (\forall s_1, s_2, s_3.\ s_3 \sqsupseteq_{\mathrm{pub}} s_2 \wedge s_2 \sqsupseteq_{\mathrm{pub}} s_1 \implies s_3 \sqsupseteq_{\mathrm{pub}} s_1)\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{VRelF}^S := \{f \in S \to \mathsf{VRelF} \to \mathsf{VRelF} \mid \\
\quad \forall s, s', R, R'.\ s' \sqsupseteq s \wedge R' \supseteq R \implies f(s')(R') \supseteq f(s)(R)\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{HRel}^S := \{f \in S \to \mathsf{VRelF} \to \mathsf{HRel} \mid \\
\quad \forall s, R, R'.\ R' \supseteq R \implies f(s)(R') \supseteq f(s)(R)\}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{World}^S := \{(\mathsf{T}, \mathsf{L}, \mathsf{H}, \mathsf{N}) \in \mathsf{TrSys} \times \mathsf{VRelF}^{S\times T.\mathsf{S}} \times \mathsf{HRel}^{S\times T.\mathsf{S}} \times \mathcal{P}(\mathsf{TyNam}) \mid \\
\quad (\forall s, R.\ \forall \mathbf{n} \notin \mathcal{P}(\mathsf{TyNam}).\ \mathsf{L}(s)(R)(\mathbf{n}) = \emptyset) \wedge \\
\quad (\forall s, R, \tau, \tau'.\ \mathsf{L}(s)(R)(\tau \to \tau') \subseteq \mathrm{FixVal} \times \mathrm{FixVal}) \wedge \\
\quad (\forall s, R, \alpha, \tau.\ \mathsf{L}(s)(R)(\forall\alpha.\,\tau) \subseteq \mathrm{GenVal} \times \mathrm{GenVal})\}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{LWorld} := \{w \in \mathrm{World}^{W_{\mathrm{ref}}.\mathsf{S}} \mid \forall s, R, \tau.\ w.\mathsf{L}(s)(R)(\mathsf{ref}\,\tau) = \emptyset\} \\
\mathrm{World} \ := \{W \in \mathrm{World}^1\}
\end{array}
$$

Figure 3.12: Definition of worlds.

both preorders. A world then consists of a transition system (T), a mapping from states to local knowledges (L), a mapping from states to heap relations (H), and a set of type names that are used to represent abstract types (N). For now, ignore the distinction between different kinds of worlds in the figure.

As before, the local knowledge (at each state) is parameterized by—and must be monotone in—the global knowledge (R). The same applies to the heap relation, which describes pairs of subheaps that are "owned" by the world. The parameter R here provides a way of referring to the global equivalence on values when establishing invariants on the contents of local heaps; this is especially critical in dealing with higher-order state.

While the local knowledge mapping must be monotone in its state index (w.r.t. the full transition relation), the heap relation mapping need not be. Since a module's local state is hidden from the environment, there is no necessity to require that heaps related in one state will continue to be related in future states. The freedom this gives us is critical even in very simple examples: imagine a transition system of two states, one in which a particular heap location contains 0 and the other in which the same location contains 1.

We have seen in the previous section how a local knowledge and its closure relate values at λ^μ types. This carries over to the full setting. But how do we deal with the additional types of $\mathsf{F}^{\mu!}$, *i.e.,* with universal, existential, and reference types?

3.6.2 Treatment of Universal and Existential Types

Universal types, like function types, are considered flexible. That is, a world's local knowledge can relate any values at any closed type $\forall\alpha.\,\tau$, as long as, when instantiated, those values can run for at least one step.

Existential types, like product types, are considered rigid, and thus their interpretation is given by the value closure (Figure 3.11). Note that the witness of related packages must be the *same* type τ'. How, then, do we support reasoning about parametricity?

The key is that the witness type τ' may be an abstract *type name*. We extend the syntax of $\mathsf{F}^{\mu!}$ types in our PBS model with type names $\mathbf{n}$ (Figure 3.10), and the local knowledge of a world can pick a subset of these names and interpret them however it wants. To avoid conflicts with other worlds, the choice of names must be recorded in the N component of the world (the local knowledge must not relate anything at other names). Note that N cannot be inferred from the local knowledge as the set of names whose interpretation is nonempty, because the empty relation is a perfectly valid interpretation of a type name.

3.6.3 Treatment of Reference Types

Reference types are considered flexible, but they really are a special case. Intuitively, the collection of all reference types can be seen as a separate module that is used by all other modules. Accordingly, we construct a designated world W_{ref} (explained below)

$$
\begin{aligned}
&W_{\text{ref}}.\mathsf{S} := \{s_{\mathsf{ref}} \in \mathcal{P}_{\text{fin}}(\mathsf{CTy} \times \mathsf{Loc} \times \mathsf{Loc}) \mid \\
&\quad \forall(\tau, l_1, l_2) \in s_{\mathsf{ref}}.\ \forall(\tau', l_1', l_2') \in s_{\mathsf{ref}}. \\
&\quad (l_1 = l_1' \implies \tau = \tau' \wedge l_2 = l_2') \wedge \\
&\quad (l_2 = l_2' \implies \tau = \tau' \wedge l_1 = l_1')\}
\end{aligned}
$$

$$
\begin{aligned}
W_{\text{ref}}.\sqsubseteq &:= \subseteq \\
W_{\text{ref}}.\sqsubseteq_{\text{pub}} &:= \subseteq
\end{aligned}
$$

$$
W_{\text{ref}}.\mathsf{L}(s_{\mathsf{ref}})(G)(\mathsf{ref}\,\tau) := \{(l_1, l_2) \mid (\tau, l_1, l_2) \in s_{\mathsf{ref}}\}
$$

$$
\begin{aligned}
&W_{\text{ref}}.\mathsf{H}(s_{\mathsf{ref}})(G) := \{(h_1, h_2) \mid \\
&\quad \text{dom}(h_1) = \{l_1 \mid \exists\tau, l_2.\ (\tau, l_1, l_2) \in s_{\mathsf{ref}}\} \wedge \\
&\quad \text{dom}(h_2) = \{l_2 \mid \exists\tau, l_1.\ (\tau, l_1, l_2) \in s_{\mathsf{ref}}\} \wedge \\
&\quad \forall(\tau, l_1, l_2) \in s_{\mathsf{ref}}.\ (h_1(l_1), h_2(l_2)) \in \overline{G}(\tau)\}
\end{aligned}
$$

$$
W_{\text{ref}}.\mathsf{N} := \emptyset
$$

Figure 3.13: W_{ref} provides the meaning of reference types.

that interprets $\mathsf{ref}\,\tau$, and bar ordinary worlds from relating anything at such types. We therefore distinguish between two kinds of worlds: *local worlds* (LWorld) and *full worlds* (World). For conciseness, both are defined in terms of the same underlying structure of *dependent worlds* (DepWorld). We sometimes call W_{ref} a *shared world* or *global world*, because it is not local to the programs we are reasoning about.

A world depending on state space S is a world as described above, except that its local knowledge and heap relation additionally take a state from S as argument (more precisely, their usual state argument is paired with a state from S). Intuitively, S is the state transition system of some other world. Thus, a full world $W \in \text{World}$ is simply a world depending on nothing (a singleton set $1 := \emptyset$). In contrast, a local world $w \in \text{LWorld}$ is a world that depends on—will later be "linked" with—W_{ref} and does not itself relate any values at reference types.

As a matter of notational convenience: if $W \in \text{World}$ and $s \in W.\mathsf{S}$, then we will usually just write $W.\mathsf{L}(s)$ for $W.\mathsf{L}(\emptyset, s)$, and similarly for the H component. (We use the dot notation to project components out of a world.)

3.6.3.1 The World for Reference Types

Figure 3.13 defines $W_{\text{ref}} \in \text{World}$, the world that provides the meaning of reference types. Its states are finite ternary relations (between a type τ and two heap locations l_1, l_2) that are functional in the location arguments. They associate each allocated location on the left with the corresponding one on the right and the type of values stored. The relations are finite because only a finite number of locations can ever be allocated. And, as dictated by the language, they can only grow over time.

Its local knowledge $W_{\text{ref}}.\mathsf{L}(s_{\mathsf{ref}})$ relates precisely the locations related by the current state s_{ref}, at the corresponding reference types. The heap relation $W_{\text{ref}}.\mathsf{H}(s_{\mathsf{ref}})$ relates heaps that contain exactly the locations related by the current state s_{ref} and that store (globally) related values at those locations. Note the critical use of the global knowledge parameter G in defining $W_{\text{ref}}.\mathsf{H}$.

3.6.4 Lifting and Separating Conjunction of Local Worlds

Now, if we have a local world $w \in \text{LWorld}$, then we can link it with W_{ref}, thereby *lifting* it to a full world $w{\uparrow} \in \text{World}$. This operation is defined in Figure 3.14. The full world's transition system $w{\uparrow}.\mathsf{T}$ is the synchronous product of W_{ref}'s and w's. Its local knowledge relates values iff they are related by either component's local knowledge, and its heap relation relates heaps iff they can be split into disjoint parts that are related by $W_{\text{ref}}.\mathsf{H}$ and $w.\mathsf{H}$, respectively. Note how the state s_{ref} of the reference world W_{ref} is passed to $w.\mathsf{L}$ and $w.\mathsf{H}$ along with the state of w itself.

Similarly, given two local worlds $w_1, w_2 \in \text{LWorld}$ that own disjoint sets of abstract types (*i.e.,* $w_1.\mathsf{N} \cap w_2.\mathsf{N} = \emptyset$), we can construct their separating conjunction $w_1 \otimes w_2 \in \text{LWorld}$. The definition is also given in Figure 3.14. Note how the same state s_{ref} is passed to the L and H components of both w_1 and w_2. While this construction is not needed for defining the model itself, it is critical for composing proofs (*e.g.,* when proving soundness). In fact, separating conjunction of local worlds is a generalization of the union operation on local knowledges, which we have seen in Section 3.5.3.

3.6.5 Program Equivalence

With these constructions in hand, we can now describe the definition of program equivalence in Figures 3.15 and 3.16.

We say that two programs are equivalent ($\sim$) iff there exists a local world w that (1) does not depend on a particular choice of names to represent its abstract types; (2) is *stable*; and (3) when lifted, relates the expressions. Stability means that the local world's heap relation in some sense tolerates "environmental" changes: whenever the shared world W_{ref} is advanced to a future state s'_{ref}, then w should be able to respond to that change by moving to a public future state s' such that any local heaps that were related previously by $w.\mathsf{H}$ are still related at s'. This is a technical condition that is required for compositionality (see Lemma 9 later on) but is satisfied trivially in the common case that $w.\mathsf{H}$ does not actually depend on its s_{ref} parameter.

A world W (such as $w{\uparrow}$) relates two expressions ($\sim_W$) iff (3a) it is *inhabited*; (3b) it is *consistent*; and (3c) the expressions, when closed using pointwise related substitutions, are related by the expression relation (see below). Inhabitance says there exists a state at which $W.\mathsf{H}$ relates the empty heaps. Consistency is essentially the same as for λ^{μ}, but extended straightforwardly to universal types. In all these definitions, the global knowledge G is drawn from $\mathsf{GK}(W)$. As before, this enforces

$$
\begin{array}{l}
(-) \times (-) \in \mathsf{TrSys} \to \mathsf{TrSys} \to \mathsf{TrSys} \\
\begin{array}{lcl}
(T_1 \times T_2).\mathsf{S} & := & T_1.\mathsf{S} \times T_2.\mathsf{S} \\
(T_1 \times T_2).\sqsubseteq & := & \{((s_1, s_2), (s_1', s_2')) \mid s_1 \sqsubseteq s_1' \wedge s_2 \sqsubseteq s_2'\} \\
(T_1 \times T_2).\sqsubseteq_{\mathrm{pub}} & := & \{((s_1, s_2), (s_1', s_2')) \mid s_1 \sqsubseteq_{\mathrm{pub}} s_1' \wedge s_2 \sqsubseteq_{\mathrm{pub}} s_2'\}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
(-) \otimes (-) \in \mathsf{HRel} \to \mathsf{HRel} \to \mathsf{HRel} \\
H_1 \otimes H_2 := \{(h_1 \sqcup h_1', h_2 \sqcup h_2') \mid (h_1, h_2) \in H_1 \wedge (h_1', h_2') \in H_2 \wedge \\
\qquad\qquad\qquad\qquad h_1 \sqcup h_1' \neq \bot \wedge h_2 \sqcup h_2' \neq \bot\}
\end{array}
$$

$$
\begin{array}{l}
(-)\uparrow \in \mathrm{LWorld} \to \mathrm{World} \\
\begin{array}{lcl}
w\uparrow.\mathsf{T} & := & W_{\mathrm{ref}}.\mathsf{T} \times w.\mathsf{T} \\
w\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s)(G) & := & W_{\mathrm{ref}}.\mathsf{L}(s_{\mathsf{ref}})(G) \cup w.\mathsf{L}(s_{\mathsf{ref}}, s)(G) \\
w\uparrow.\mathsf{H}(s_{\mathsf{ref}}, s)(G) & := & W_{\mathrm{ref}}.\mathsf{H}(s_{\mathsf{ref}})(G) \otimes w.\mathsf{H}(s_{\mathsf{ref}}, s)(G) \\
w\uparrow.\mathsf{N} & := & w.\mathsf{N}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
(-) \otimes (-) \in \mathrm{LWorld} \to \mathrm{LWorld} \rightharpoonup \mathrm{LWorld} \\
\begin{array}{lcl}
(w_1 \otimes w_2).\mathsf{T} & := & w_1.\mathsf{T} \times w_2.\mathsf{T} \\
(w_1 \otimes w_2).\mathsf{L}(s_{\mathsf{ref}}, s)(G) & := & w_1.\mathsf{L}(s_{\mathsf{ref}}, s)(G) \cup w_2.\mathsf{L}(s_{\mathsf{ref}}, s)(G) \\
(w_1 \otimes w_2).\mathsf{H}(s_{\mathsf{ref}}, s)(G) & := & w_1.\mathsf{H}(s_{\mathsf{ref}}, s)(G) \otimes w_2.\mathsf{H}(s_{\mathsf{ref}}, s)(G) \\
(w_1 \otimes w_2).\mathsf{N} & := & w_1.\mathsf{N} \uplus w_2.\mathsf{N}
\end{array}
\end{array}
$$

Figure 3.14: Lifting and separating conjunction of local worlds.

$$
\begin{array}{l}
R' \supseteq^{\mathcal{N}}_{\mathrm{ref}} R := \\
\quad R' \supseteq R \wedge (\forall \tau.\ R'(\mathsf{ref}\ \tau) = R(\mathsf{ref}\ \tau)) \wedge (\forall \mathbf{n} \in \mathcal{N}.\ R'(\mathbf{n}) = R(\mathbf{n}))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{GK}(W) := \{G \in W.\mathsf{S} \to \mathsf{VRelF} \mid \\
\quad (\forall s, s'.\ s' \sqsupseteq s \implies G(s') \supseteq G(s)) \ \wedge\ (\forall s.\ G(s) \supseteq^{W.\mathsf{N}}_{\mathrm{ref}} W.\mathsf{L}(s)(G(s)))\}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{E}_W(G)(s_0)(s)(\tau) := \{(e_1, e_2) \mid \\
\quad \forall (h_1, h_2) \in W.\mathsf{H}(s)(G(s)).\ \forall h_1^{\mathrm{F}}, h_2^{\mathrm{F}}.\ h_1 \sqcup h_1^{\mathrm{F}} \neq \bot \wedge h_2 \sqcup h_2^{\mathrm{F}} \neq \bot \implies \\
\qquad (\textsc{div}) \quad \langle h_1 \sqcup h_1^{\mathrm{F}}; e_1\rangle\!\uparrow \wedge \langle h_2 \sqcup h_2^{\mathrm{F}}; e_2\rangle\!\uparrow \\
\qquad \vee\ (\textsc{eval}) \ \exists h_1', h_2', v_1, v_2, s'. \\
\qquad\qquad \langle h_1 \sqcup h_1^{\mathrm{F}}; e_1\rangle \hookrightarrow^* \langle h_1' \sqcup h_1^{\mathrm{F}}; v_1\rangle \wedge \langle h_2 \sqcup h_2^{\mathrm{F}}; e_2\rangle \hookrightarrow^* \langle h_2' \sqcup h_2^{\mathrm{F}}; v_2\rangle \wedge \\
\qquad\qquad s' \sqsupseteq s \wedge s' \sqsupseteq_{\mathrm{pub}} s_0 \wedge (h_1', h_2') \in W.\mathsf{H}(s')(G(s')) \wedge (v_1, v_2) \in \overline{G(s')}(\tau) \\
\qquad \vee\ (\textsc{call}) \ \exists h_1', h_2', \tau', K_1, K_2, e_1', e_2', s'. \\
\qquad\qquad (h_1 \sqcup h_1^{\mathrm{F}}, e_1) \hookrightarrow^* (h_1' \sqcup h_1^{\mathrm{F}}, K_1[e_1']) \wedge \\
\qquad\qquad (h_2 \sqcup h_2^{\mathrm{F}}, e_2) \hookrightarrow^* (h_2' \sqcup h_2^{\mathrm{F}}, K_2[e_2']) \wedge \\
\qquad\qquad s' \sqsupseteq s \wedge (h_1', h_2') \in W.\mathsf{H}(s')(G(s')) \wedge (e_1', e_2') \in \mathbf{U}(G(s'), G(s'))(\tau') \wedge \\
\qquad\qquad \forall s'' \sqsupseteq_{\mathrm{pub}} s'.\ \forall G' \supseteq G.\ (K_1, K_2) \in \mathbf{K}_W(G')(s_0)(s'')(\tau', \tau)\}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{K}_W(G)(s_0)(s)(\tau)(\tau') := \{(K_1, K_2) \mid \\
\quad \forall (v_1, v_2) \in \overline{G(s)}(\tau).\ (K_1[v_1], K_2[v_2]) \in \mathbf{E}_W(G)(s_0)(s)(\tau')\}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{U}(R, R')(\tau) := \\
\quad \{(v_1\ v_1', v_2\ v_2') \mid \exists \tau'.\ (v_1, v_2) \in R(\tau' \to \tau) \wedge (v_1', v_2') \in \overline{R'}(\tau')\} \cup \\
\quad \{(v_1\ [], v_2\ []) \mid \exists \tau_1, \tau_2.\ \tau = \tau_1[\tau_2/\alpha] \wedge (v_1, v_2) \in R(\forall \alpha.\ \tau_1)\}
\end{array}
$$

Figure 3.15: Mutually coinductive definitions of expression equivalence, $\mathbf{E}_W \in \mathsf{GK}(W) \to W.\mathsf{S} \times W.\mathsf{S} \to \mathsf{ERel}$, and continuation equivalence, $\mathbf{K}_W \in \mathsf{GK}(W) \to W.\mathsf{S} \times W.\mathsf{S} \to \mathsf{KRel}$ for $\mathsf{F}^{\mu!}$.

$\mathsf{inhabited}(W) :=$
$\quad \forall G \in \mathsf{GK}(W).\ \exists s_0.\ (\emptyset, \emptyset) \in W.\mathsf{H}(s_0)(G(s_0))$

$\mathsf{consistent}(W) :=$
$\quad \forall G \in \mathsf{GK}(W).\ \forall s.\ \forall \tau.\ \forall (e_1, e_2) \in \mathbf{U}(W.\mathsf{L}(s)(G(s)), G(s))(\tau).$
$\quad (\mathit{step}(e_1), \mathit{step}(e_2)) \in \mathbf{E}_W(G)(s, s)(\tau)$

$\mathsf{stable}(w) :=$
$\quad \forall G \in \mathsf{GK}(w{\uparrow}).\ \forall s_{\mathsf{ref}}, s.\ \forall (h_1, h_2) \in w.\mathsf{H}(s_{\mathsf{ref}}, s)(G(s_{\mathsf{ref}}, s)).\ \forall s'_{\mathsf{ref}} \sqsupseteq s_{\mathsf{ref}}.$
$\quad \forall (h^1_{\mathrm{ref}}, h^2_{\mathrm{ref}}) \in W_{\mathrm{ref}}.\mathsf{H}(s'_{\mathsf{ref}})(G(s'_{\mathsf{ref}}, s)).\ h^1_{\mathrm{ref}} \sqcup h_1 \neq \bot \wedge h^2_{\mathrm{ref}} \sqcup h_2 \neq \bot \implies$
$\quad \exists s' \sqsupseteq_{\mathrm{pub}} s.\ (h_1, h_2) \in w.\mathsf{H}(s'_{\mathsf{ref}}, s')(G(s'_{\mathsf{ref}}, s'))$

$\mathsf{Env}(\Gamma, R) :=$
$\quad \{(\gamma_1, \gamma_2) \in (\mathrm{dom}(\Gamma) \to \mathsf{CVal})^2 \mid \forall x{:}\tau \in \Gamma.\ (\gamma_1 x, \gamma_2 x) \in \overline{R}(\tau)\}$

$\Delta; \Gamma \vdash e_1 \sim_W e_2 : \tau :=$
$\quad \mathsf{inhabited}(W) \wedge \mathsf{consistent}(W) \wedge$
$\quad \forall G \in \mathsf{GK}(W).\ \forall s.\ \forall \delta \in \Delta \to \mathsf{CTy}.\ \forall (\gamma_1, \gamma_2) \in \mathsf{Env}(\delta\Gamma, G(s)).$
$\quad (\gamma_1 e_1, \gamma_2 e_2) \in \mathbf{E}_W(G)(s)(s)(\delta\tau)$

$\Delta; \Gamma \vdash e_1 \sim e_2 : \tau :=$
$\quad \forall \mathcal{N} \in \mathcal{P}(\mathsf{TyNam}).\ \mathcal{N}$ countably infinite $\implies$
$\quad \exists w.\ w.\mathsf{N} \subseteq \mathcal{N} \wedge \mathsf{stable}(w) \wedge \Delta; \Gamma \vdash e_1 \sim_{w{\uparrow}} e_2 : \tau$

Figure 3.16: Definitions of world consistency and program equivalence for $\mathsf{F}^{\mu!}$.

that G must *contain* the local knowledge. At reference types, and at abstract type names owned by W, however, $\mathsf{GK}(W)$ also enforces that G must *not extend* the local knowledge. Intuitively, this is because W should completely control the meaning of those types. Furthermore, since G is state-indexed, it must, like the local knowledge of W, be monotone w.r.t. any state changes.

3.6.6 Expression and Continuation Equivalence

The new definitions of $\mathbf{E}$ and $\mathbf{K}$ are also given in Figure 3.15. Notice that they are now defined relative to a world W (as $\mathbf{E}_W$ and $\mathbf{K}_W$) and that their types have changed to $\mathsf{GK}(W) \to W.\mathsf{S} \to W.\mathsf{S} \to \mathsf{ERel}$ and $\mathsf{GK}(W) \to W.\mathsf{S} \times W.\mathsf{S} \to \mathsf{KRel}$, respectively: they take both an "initial" state, s_0, and a "current" state, s, as arguments.

Given a world W, a global knowledge $G \in \mathsf{GK}(W)$, states $s_0, s \in W.\mathsf{S}$, and a type τ, we say that two expressions are "locally" equivalent, written $(e_1, e_2) \in \mathbf{E}_W(G)(s_0)(s)(\tau)$, iff, when executed starting in heaps that satisfy the heap relation of W at the current state s, then (as before) one of three cases holds:

Case DIV. Both expressions diverge (run forever).

Case EVAL. Both expressions run successfully to completion, producing related values. In this case, the values need not be related in the current state s, but rather in some future state, $s' \sqsupseteq s$, which, however, must also be a public future state of the initial state of the expression: $s' \sqsupseteq_{\text{pub}} s_0$. Moreover, this future state must be consistent with the resulting heaps: $(h_1', h_2') \in W.\mathsf{H}(s')(G(s'))$.

Case CALL. Both expressions reduce after some number of steps to some expressions of the form $K_i[e_i']$, where e_i' are either both applications or both instantiations that are related at some future state $s' \sqsupseteq s$. This state must be consistent with the corresponding heaps. Finally, the continuations, K_1 and K_2, must be equivalent under any public future state $s'' \sqsupseteq_{\text{pub}} s'$ and any (pointwise) larger global knowledge $G' \supseteq G$.

We restrict s'' to be a public future state of s' rather than an arbitrary future state because the end-to-end effect of a function call (or universal instantiation) is assumed to always be a public transition (recall the discussion of public vs. private transitions in Section 2.2). For this assumption to be sound, in return we will have to ensure that the end-to-end behaviors of equivalent function bodies indeed change the state only into public future states. This is why we thread the s_0 argument through the mutual recursion and check that the final state in case EVAL is a public future state of s_0.

The intuitive reason for quantifying over a larger global knowledge G' in $\mathbf{K}$ is this: At the point when the continuations are run, not only might W's state s' have changed to a future state s'', but also the states of all *other* "modules", which is reflected by the growth of the global knowledge (also see the next section). This was not needed in Section 3.5 due to the purity of λ^μ.

In all three cases, the definition quantifies over "frame" heaps h_1^{F} and h_2^{F}: the execution of e_1 and e_2 should not modify any part of the heap that they do not own according to the heap relation of the current state. This framing aspect of our definition is a semantic version of the frame rule of separation logic and allows us to concentrate the reasoning about the heaps only on the parts of the heaps accessed by the program. (Baking the frame rule into the semantic model is quite common in more recent models of separation logic [10, 88], essentially because it allows one to avoid proving any "safety monotonicity" or "frame" properties of the operational semantics itself.)

3.6.7 Living in a Different World

In Chapter 2, we reviewed at a high level the idea of STSs and how they are used in KLRs. While PBs build very closely on this technique, there are some big differences in the implementation, which are not obvious due to the informal style of Chapter 2.

In KLRs, when proving the equivalence of two programs, we consider them in an aribtrary initial world. As we step through the programs, we may move to a future world, *i.e.,* extend the current world with new state transition systems governing newly allocated pieces of local memory and advance the states of existing state transition systems that we know about. In return, when establishing the relatedness of functions or continuations, we are always forced to consider their behaviour in future worlds. Intuitively, this means two things:

1. Since we extend the world "on the fly", the STSs that we add are *instance-specific*. They normally refer to the particular locations that were allocated in one run of the programs that we are reasoning about.

2. However, due to the consideration of arbitrary initial and future worlds, the worlds we work with do not only contain the STSs that we explicitly added (even if we only explicitly care about those in our proof). They also contain (*i*) those belonging to other instances of our programs, and (*ii*) those belonging to the environment, *i.e.,* governing other modules that our programs were linked with.

Let us contrast this with how STSs function in PBs. Recall that here, in order to prove two programs equivalent, we need to come up with a suitable world, and thus a single STS, a priori.

1. This world must hence describe *all* possible instances of our programs, *i.e.,* it must be *module-specific* rather than *instance-specific* (or, *static* rather than *dynamic*). This typically means that the STS that we construct is an n-ary version of what we would construct in a KLR setting (for unbounded n) and thus more complex. The examples in Section 3.8 will make this clearer. The same section also shows how this extra complexity can be reduced.

2. What was the quantification over future worlds in KLRs, now is (i) the quantification over future states, plus (ii) the quantification over larger global knowledges. The latter is necessary because the world only contains our module's constraints, not those of other modules.

3.7 Metatheory

3.7.1 Basics

In a typical proof using PBS, we are given a global knowledge $G \in \mathsf{GK}(W)$ (recall the definition of $\sim$). Sometimes, however, it is important that we can *construct* such a global knowledge. We now define the least global knowledge for a given world.

Definition 3. For a monotone function $F \in \mathsf{VRelF} \to \mathsf{VRelF}$ and $R \in \mathsf{VRelF}$, we define $[F]^*_R$ as the least fixed point of the monotone function $F(-) \cup R$:

$$[F]^*_R := \mu X.\ F(X) \cup R$$

Definition 4 (Least global knowledge)**.** For $W \in \text{World}$ we define $[W] \in W.\mathsf{T}.\mathsf{S} \to \mathsf{VRelF}$ as follows:

$$[W](s) := [W.\mathsf{L}(s)]^*_\emptyset$$

Lemma 7 (Least global knowledge)**.** For any W, we have $[W] \in \mathsf{GK}(W)$. Moreover, $[W] \subseteq G$ whenever $G \in \mathsf{GK}(W)$.

Next are three lemmas concerning product worlds. The first one says that a valid global knowledge for a (lifted) product world is also valid for the (lifted) constituent worlds, after fixing the respectively "missing" state. This generalizes Lemma 1.

Lemma 8. If $w_1, w_2 \in \text{LWorld}$ and $G \in \mathsf{GK}((w_1 \otimes w_2)\uparrow)$, then:

1. $\forall s_2 \in w_2.\mathsf{T}.\mathsf{S}.\ G(-,-,s_2) \in \mathsf{GK}(w_1\uparrow)$.
2. $\forall s_1 \in w_1.\mathsf{T}.\mathsf{S}.\ G(-,s_1,-) \in \mathsf{GK}(w_2\uparrow)$.

(Note that the way the product world $w_1 \otimes w_2$ is used as an argument to $(-)\uparrow$ implicitly assumes that the product is defined, *i.e.,* $w_1.\mathsf{N}$ and $w_2.\mathsf{N}$ are disjoint.)

The second lemma relates a product world's **E** and **K** relations to those of its constituent worlds. Its proof is illustrative because (i) it is the only one that relies on stability, and (ii) its structure is representative of other metatheoretical proofs that work by coinduction on **E** and **K**.

Lemma 9. Suppose $w = w_1 \otimes w_2$ and $G \in \mathsf{GK}(w\uparrow)$ and $s^0_{\mathsf{ref}}, s_{\mathsf{ref}} \in W_{\text{ref}}.\mathsf{S}$ and $s^0_1, s_1 \in w_1.\mathsf{S}$ and $s^0_2, s_2 \in w_2.\mathsf{S}$. If $s_2 \sqsupseteq_{\text{pub}} s^0_2$ and $\mathsf{stable}(w_2)$, then:

1. $\mathbf{E}_{w_1\uparrow}(G(-,-,s_2))(s^0_{\mathsf{ref}}, s^0_1)(s_{\mathsf{ref}}, s_1) \subseteq \mathbf{E}_{w\uparrow}(G)(s^0_{\mathsf{ref}}, s^0_1, s^0_2)(s_{\mathsf{ref}}, s_1, s_2)$

2. $\mathbf{K}_{w_1\uparrow}(G(-,-,s_2))(s^0_{\mathsf{ref}}, s^0_1)(s_{\mathsf{ref}}, s_1) \subseteq \mathbf{K}_{w\uparrow}(G)(s^0_{\mathsf{ref}}, s^0_1, s^0_2)(s_{\mathsf{ref}}, s_1, s_2)$

And similarly for w_2 if $s_1 \sqsupseteq_{\text{pub}} s^0_1$ and $\mathsf{stable}(w_1)$.

Proof. In order to be able to apply coinduction, we define auxiliaries E and K as follows:

$$E(G)(s^0_{\mathsf{ref}}, s^0_1, s^0_2)(s_{\mathsf{ref}}, s_1, s_2)(\tau) = \{(e_1, e_2) \mid \\ s_2 \sqsupseteq_{\text{pub}} s^0_2 \wedge (e_1, e_2) \in \mathbf{E}_{w_1\uparrow}(G(-,-,s_2))(s^0_{\mathsf{ref}}, s^0_1)(s_{\mathsf{ref}}, s_1)(\tau)\}$$

$$K(G)(s^0_{\mathsf{ref}}, s^0_1, s^0_2)(s_{\mathsf{ref}}, s_1, s_2)(\tau')(\tau) = \{(K_1, K_2) \mid \\ s_2 \sqsupseteq_{\text{pub}} s^0_2 \wedge (K_1, K_2) \in \mathbf{K}_{w_1\uparrow}(G(-,-,s_2))(s^0_{\mathsf{ref}}, s^0_1)(s_{\mathsf{ref}}, s_1)(\tau')(\tau)\}$$

Note that it suffices to prove $E \subseteq \mathbf{E}_{w\uparrow}$ and $K \subseteq \mathbf{K}_{w\uparrow}$ (assuming stability of w_2), which we do by coinduction. Concretely, we have to show:

1. $\forall G, s^0_{\mathsf{ref}}, s^0_1, s^0_2, s_{\mathsf{ref}}, s_1, s_2, \tau.\ \forall (e_1, e_2) \in E(G)(s^0_{\mathsf{ref}}, s^0_1, s^0_2)(s_{\mathsf{ref}}, s_1, s_2)(\tau).$
   $\forall (h_1, h_2) \in w\uparrow.\mathsf{H}(s_{\mathsf{ref}}, s_1, s_2)(G(s_{\mathsf{ref}}, s_1, s_2)).\forall h^{\mathrm{F}}_1, h^{\mathrm{F}}_2.$
   $h_1 \sqcup h^{\mathrm{F}}_1 \neq \bot \wedge h_2 \sqcup h^{\mathrm{F}}_2 \neq \bot \implies$
   $(\langle h_1 \sqcup h^{\mathrm{F}}_1; e_1\rangle\uparrow \wedge \langle h_2 \sqcup h^{\mathrm{F}}_2; e_2\rangle\uparrow)$
   $\vee\ (\exists h'_1, h'_2, v_1, v_2, s'.$
   $\langle h_1 \sqcup h^{\mathrm{F}}_1; e_1\rangle \hookrightarrow^* \langle h'_1 \sqcup h^{\mathrm{F}}_1; v_1\rangle \wedge \langle h_2 \sqcup h^{\mathrm{F}}_2; e_2\rangle \hookrightarrow^* \langle h'_2 \sqcup h^{\mathrm{F}}_2; v_2\rangle \wedge$
   $s' \sqsupseteq (s_{\mathsf{ref}}, s_1, s_2) \wedge s' \sqsupseteq_{\text{pub}} (s^0_{\mathsf{ref}}, s^0_1, s^0_2) \wedge$
   $(h'_1, h'_2) \in w\uparrow.\mathsf{H}(s')(G(s')) \wedge (v_1, v_2) \in \overline{G(s')}(\tau))$
   $\vee\ (\exists h'_1, h'_2, \tau', K_1, K_2, e'_1, e'_2, s'.$
   $(h_1 \sqcup h^{\mathrm{F}}_1, e_1) \hookrightarrow^* (h'_1 \sqcup h^{\mathrm{F}}_1, K_1[e'_1]) \wedge (h_2 \sqcup h^{\mathrm{F}}_2, e_2) \hookrightarrow^* (h'_2 \sqcup h^{\mathrm{F}}_2, K_2[e'_2]) \wedge$
   $s' \sqsupseteq (s_{\mathsf{ref}}, s_1, s_2) \wedge (h'_1, h'_2) \in w\uparrow.\mathsf{H}(s')(G(s')) \wedge (e'_1, e'_2) \in \mathbf{U}(G(s'), G(s'))(\tau') \wedge$
   $(K_1, K_2) \in K(G)(s^0_{\mathsf{ref}}, s^0_1, s^0_2)(s')(\tau', \tau))$

2. $\forall G, s^0_{\mathsf{ref}}, s^0_1, s^0_2, s_{\mathsf{ref}}, s_1, s_2, \tau', \tau.$
   $\forall (K_1, K_2) \in K(G)(s^0_{\mathsf{ref}}, s^0_1, s^0_2)(s_{\mathsf{ref}}, s_1, s_2)(\tau', \tau).\ \forall (v_1, v_2) \in \overline{G(s_{\mathsf{ref}}, s_1, s_2)}(\tau').$
   $(K_1[v_1], K_2[v_2]) \in E(G)(s^0_{\mathsf{ref}}, s^0_1, s^0_2)(s_{\mathsf{ref}}, s_1, s_2)(\tau)$

Part (2) is straightforward. Let us look at part part (1). We are given heaps h_1, h_2 related by the lifted product world $w\uparrow$. We know that these are composed as follows:

$$h_1 = h'_1 \uplus h''_1$$
$$h_2 = h'_2 \uplus h''_2$$

$$(h^a_1, h^a_2) \in w_1\uparrow.\mathsf{H}(s_{\mathsf{ref}}, s_1)(G(s_{\mathsf{ref}}, s_1, s_2)) \qquad (*)$$
$$(h^b_1, h^b_2) \in w_2.\mathsf{H}(s_{\mathsf{ref}}, s_2)(G(s_{\mathsf{ref}}, s_1, s_2))$$

Naturally, we use $(*)$ to instantiate the relatedness of (e_1, e_2) by $\mathbf{E}_{w_1\uparrow}$ (in the definition of E). As frame heaps, we choose $h^b_1 \uplus h^{\mathrm{F}}_1$ and $h^b_2 \uplus h^{\mathrm{F}}_2$, so that the programs cannot violate the constraints imposed by w_2. We do a case split on the disjunct according to which e_1 and e_2 are related and in each case choose to show the corresponding disjunct of the goal.

Stability is required for the cases EVAL and CALL. For instance, suppose e_1 and e_2 are related because they both terminate with related values in state $(s'_{\mathsf{ref}}, s'_1) \sqsupseteq (s_{\mathsf{ref}}, s_1)$. We know that the frame heaps have been preserved and that there are $h_1^{a\prime}, h_2^{a\prime}$ (disjoint from the frames) and future states s'_{ref}, s'_1 with $s'_{\mathsf{ref}} \sqsupseteq s_{\mathsf{ref}}$, $s'_{\mathsf{ref}} \sqsupseteq_{\text{pub}} s^0_{\mathsf{ref}}$, $s'_1 \sqsupseteq s_1$, $s_1 \sqsupseteq_{\text{pub}} s^0_1$, and

$$(h_1^{a\prime}, h_2^{a\prime}) \in w_1{\uparrow}.\mathsf{H}(s'_{\mathsf{ref}}, s'_1)(G(s'_{\mathsf{ref}}, s'_1, s_2)).$$

To show the goal, we must now find s'_2 with $s'_2 \sqsupseteq s_2$ and $s'_2 \sqsupseteq_{\text{pub}} s^0_2$ such that

$$(h_1^{a\prime} \uplus h_1^b, h_2^{a\prime} \uplus h_2^b) \in w{\uparrow}.\mathsf{H}(s'_{\mathsf{ref}}, s'_1, s'_2)(G(s'_{\mathsf{ref}}, s'_1, s'_2)),$$

i.e., (using monotonicity of H and G) such that:

$$(h_1^b, h_2^b) \in w_2.\mathsf{H}(s'_{\mathsf{ref}}, s'_2)(G(s'_{\mathsf{ref}}, s'_1, s'_2))$$

Relying on $s_2 \sqsupseteq_{\text{pub}} s^0_2$ (enforced by the definition of E), stability of w_2 yields exactly what we need. □

The third property of product worlds generalizes Lemma 2.

Lemma 10. If $w = w_1 \otimes w_2$ with $\mathsf{stable}(w_1)$ and $\mathsf{stable}(w_2)$, then:

1. $\mathsf{stable}(w)$
2. If $\mathsf{inhabited}(w_1{\uparrow})$ and $\mathsf{inhabited}(w_2{\uparrow})$, then $\mathsf{inhabited}(w{\uparrow})$.
3. If $\mathsf{consistent}(w_1{\uparrow})$ and $\mathsf{consistent}(w_2{\uparrow})$, then $\mathsf{consistent}(w{\uparrow})$.

The next lemma talks about composing continuations with other continuations or expressions and generalizes Lemma 3.

Lemma 11. Given $G, s_0, s'_0, s, \tau, \tau', K_1, K_2$ such that

$$\forall G' \supseteq G.\ \forall s'.\ s' \sqsupseteq s \wedge s' \sqsupseteq_{\text{pub}} s'_0 \implies (K_1, K_2) \in \mathbf{K}_W(G')(s_0)(s')(\tau')(\tau),$$

we have:

1. $(e_1, e_2) \in \mathbf{E}_W(G)(s'_0, s)(\tau') \implies (K_1[e_1], K_2[e_2]) \in \mathbf{E}_W(G)(s_0, s)(\tau)$
2. $(K'_1, K'_2) \in \mathbf{K}_W(G)(s'_0, s)(\tau'')(\tau') \implies (K_1[K'_1], K_2[K'_2]) \in \mathbf{K}_W(G)(s_0, s)(\tau'')(\tau)$

Let us now come to the external call lemma, the generalization of Lemma 4. First we parameterize $\mathbf{E}$ over the relation by which functions are related via CALL.

Definition 5.

$$
\begin{aligned}
&\mathbf{E}^{\boldsymbol{\mathcal{R}}}_W(G)(s_0)(s)(\tau) := \{(e_1, e_2) \mid \\
&\quad \forall (h_1, h_2) \in W.\mathsf{H}(s)(G(s)).\ \forall h_1^{\mathrm{F}}, h_2^{\mathrm{F}}.\ h_1 \sqcup h_1^{\mathrm{F}} \neq \bot \wedge h_2 \sqcup h_2^{\mathrm{F}} \neq \bot \implies \\
&\qquad (\langle h_1 \sqcup h_1^{\mathrm{F}}; e_1\rangle\!\uparrow \wedge \langle h_2 \sqcup h_2^{\mathrm{F}}; e_2\rangle\!\uparrow) \\
&\qquad \vee (\exists h_1', h_2', v_1, v_2, s'. \\
&\qquad\quad \langle h_1 \sqcup h_1^{\mathrm{F}}; e_1\rangle \hookrightarrow^* \langle h_1' \sqcup h_1^{\mathrm{F}}; v_1\rangle \wedge \langle h_2 \sqcup h_2^{\mathrm{F}}; e_2\rangle \hookrightarrow^* \langle h_2' \sqcup h_2^{\mathrm{F}}; v_2\rangle \wedge \\
&\qquad\quad s' \sqsupseteq s \wedge s' \sqsupseteq_{\mathrm{pub}} s_0 \wedge (h_1', h_2') \in W.\mathsf{H}(s')(G(s')) \wedge (v_1, v_2) \in \overline{G(s')}(\tau)) \\
&\qquad \vee (\exists h_1', h_2', \tau', K_1, K_2, e_1', e_2', s'. \\
&\qquad\quad (h_1 \sqcup h_1^{\mathrm{F}}, e_1) \hookrightarrow^* (h_1' \sqcup h_1^{\mathrm{F}}, K_1[e_1']) \wedge (h_2 \sqcup h_2^{\mathrm{F}}, e_2) \hookrightarrow^* (h_2' \sqcup h_2^{\mathrm{F}}, K_2[e_2']) \wedge \\
&\qquad\quad s' \sqsupseteq s \wedge (h_1', h_2') \in W.\mathsf{H}(s')(G(s')) \wedge (e_1', e_2') \in \mathbf{U}(\boldsymbol{\mathcal{R}}(s'), G(s'))(\tau') \wedge \\
&\qquad\quad (K_1, K_2) \in \mathbf{K}_W(G)(s_0)(s')(\tau')(\tau)) \\
&\}
\end{aligned}
$$

Note how we use $\boldsymbol{\mathcal{R}}(s')$ rather than $G(s')$ as first argument to $\mathbf{U}$. Other than that, $\mathbf{E}^{\boldsymbol{\mathcal{R}}}$ is identical to $\mathbf{E}$. It is actually defined in terms of $\mathbf{E}$, and not recursive itself.

Lemma 12 (External call)**.** Suppose $\mathsf{consistent}(W)$, $G \in \mathsf{GK}(W)$ and $\boldsymbol{\mathcal{R}} \in W.\mathsf{T}.\mathsf{S} \to \mathsf{VRelF}$. If $\boldsymbol{\mathcal{R}}$ is the "external" part of G, *i.e.,*

$$\forall s.\ G(s) = W.\mathsf{L}(s)(G(s)) \cup \boldsymbol{\mathcal{R}}(s),$$

then $\mathbf{E}_W(G) = \mathbf{E}^{\boldsymbol{\mathcal{R}}}_W(G)$.

Proof. The $\supseteq$-direction is trivial due to monotonicity of $\mathbf{U}$. For the other direction, the high-level proof idea is the same as in the simple setting: we eliminate all uses of the CALL disjunct of $\mathbf{E}$ where the invoked functions are related by W by relying on W's consistency.

More formally, we prove by induction on n that

$$\forall n.\ \mathbf{E}_W(G) \subseteq X_n(G),$$

where X_n is defined like $\mathbf{E}^{\boldsymbol{\mathcal{R}}}_W$ except that divergence ($\uparrow$) in the case DIV is replaced with $\uparrow^n$, meaning that the two programs can take at least n steps. Notice how this universally quantified property is therefore equivalent to $\mathbf{E}_W(G) \subseteq \mathbf{E}^{\boldsymbol{\mathcal{R}}}_W(G)$.

In the base case ($n = 0$), we are done immediately by choosing the DIV disjunct of X_0 (clearly any program can take at least 0 steps).

The inductive case is the interesting one. We assume $\mathbf{E}_W(G) \subseteq X_{n-1}(G)$ and must show $\mathbf{E}_W(G) \subseteq X_n(G)$. So suppose $(e_1, e_2) \in \mathbf{E}_W(G)(s^0)(s)(\tau)$. We are now given related heaps and frame heaps as dictated by the definition of X_n and use these to instantiate $(e_1, e_2) \in \mathbf{E}_W(G)(s^0)(s)(\tau)$. This yields three cases, one per disjunct. In case DIV, we are done because $\uparrow$ implies $\uparrow^n$. In case EVAL, we are done trivially. In case CALL we have $e_1 \hookrightarrow^* K_1[e_1']$ and $e_2 \hookrightarrow^* K_2[e_2']$ where e_1' and e_2' are calls to G-related functions. Now, if these functions are related by $\boldsymbol{\mathcal{R}}$, the external part of G, then we are done trivially again. Otherwise they are related by the local

knowledge $W.\mathsf{L}$. Since the world is consistent, we can exploit this fact to learn that the beta-reduced calls $step(e_1')$ and $step(e_2')$ are related by $\mathbf{E}_W(G)$. By Lemma 11, $K_1[step(e_1')]$ and $K_2[step(e_2')]$ are related by $\mathbf{E}_W(G)$ as well, and hence by $X_{n-1}(G)$. Since the initial programs take at least one step each to reach these intermediate forms (*i.e.*, $e_i \hookrightarrow^+ K_i[step(e_i')]$), we can conclude that e_1 and e_2 are in fact related by $X_n(G)$. □

3.7.2 Symmetry

We now show that PBs are symmetric (see the final Theorem 3). The proof is not complicated, but it does require a few definitions and lemmas.

Definition 6. Given $R \in \mathsf{VRel}$ (or VRelF), we define its inverse $R^{-1} \in \mathsf{VRel}$ (or VRelF) pointwise:

$$R^{-1} := \lambda\tau.\ R(\tau)^{-1}$$

Lemma 13. $\forall R \in \mathsf{VRelF}.\ (\overline{R})^{-1} = \overline{R^{-1}}$

Proof. By an easy induction. □

Lemma 14. $\forall R, R' \in \mathsf{VRel}.\ \mathbf{U}(R^{-1}, R'^{-1}) = \big(\mathbf{U}(R, R')\big)^{-1}$

Definition 7. Given $w \in \mathrm{LWorld}$, we define $w^{-1} \in \mathrm{LWorld}$ as follows:

$$\begin{array}{lcl} w^{-1}.\mathsf{T} & := & w.\mathsf{T} \\ w^{-1}.\mathsf{L}(s_{\mathsf{ref}}, s)(R) & := & \big(w.\mathsf{L}(s_{\mathsf{ref}}^{-1}, s)(R^{-1})\big)^{-1} \\ w^{-1}.\mathsf{H}(s_{\mathsf{ref}}, s)(R) & := & \big(w.\mathsf{H}(s_{\mathsf{ref}}^{-1}, s)(R^{-1})\big)^{-1} \\ w^{-1}.\mathsf{N} & := & w.\mathsf{N} \end{array}$$

Here, s_{ref}^{-1} is short for $\{(\tau, l_2, l_1) \mid (\tau, l_1, l_2) \in s_{\mathsf{ref}}\}$.

Lemma 15. $\forall w, s_{\mathsf{ref}}, s, R.\ w^{-1}{\uparrow}.\mathsf{H}(s_{\mathsf{ref}}, s)(R) = \big(w{\uparrow}.\mathsf{H}(s_{\mathsf{ref}}^{-1}, s)(R^{-1})\big)^{-1}$

Proof.

$$\begin{array}{lcl} & & w^{-1}{\uparrow}.\mathsf{H}(s_{\mathsf{ref}}, s)(R) \\ & = & W_{\mathrm{ref}}.\mathsf{H}(s_{\mathsf{ref}})(R) \otimes w^{-1}.\mathsf{H}(s_{\mathsf{ref}}, s)(R) \\ & = & \Big(\big(W_{\mathrm{ref}}.\mathsf{H}(s_{\mathsf{ref}})(R)\big)^{-1} \otimes \big(w^{-1}.\mathsf{H}(s_{\mathsf{ref}}, s)(R)\big)^{-1}\Big)^{-1} \\ & = & \big(W_{\mathrm{ref}}.\mathsf{H}(s_{\mathsf{ref}}^{-1})(R^{-1}) \otimes w.\mathsf{H}(s_{\mathsf{ref}}^{-1}, s)(R^{-1})\big)^{-1} \\ & = & \big(w{\uparrow}.\mathsf{H}(s_{\mathsf{ref}}^{-1}, s)(R^{-1})\big)^{-1} \end{array}$$

□

Lemma 16. $\forall w, s_{\mathsf{ref}}, s, R.\ w^{-1}{\uparrow}.\mathsf{L}(s_{\mathsf{ref}}, s)(R) = \big(w{\uparrow}.\mathsf{L}(s_{\mathsf{ref}}^{-1}, s)(R^{-1})\big)^{-1}$

Proof. Analogous to Lemma 15. □

Definition 8. Given $G \in \mathsf{GK}(w^{-1}\uparrow)$, we define its inverse G^{-1} as follows:

$$G^{-1}(s_{\mathsf{ref}}, s) := G(s_{\mathsf{ref}}^{-1}, s)^{-1}$$

Lemma 17. If $G \in \mathsf{GK}(w^{-1}\uparrow)$, then $G^{-1} \in \mathsf{GK}(w\uparrow)$.

Proof. Monotonicity of G^{-1} follows immediately from monotonicity of G. It remains to show

$$G^{-1}(s_{\mathsf{ref}}, s) \supseteq_{\mathrm{ref}}^{w\uparrow.\mathsf{N}} w\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s)(G^{-1}(s_{\mathsf{ref}}, s))$$

for any s_{ref}, s. This is easy using Lemma 16:

$$\begin{array}{ll} & G^{-1}(s_{\mathsf{ref}}, s) \\ = & G(s_{\mathsf{ref}}^{-1}, s)^{-1} \\ \supseteq_{\mathrm{ref}}^{w\uparrow.\mathsf{N}} & \left(w^{-1}\uparrow.\mathsf{L}(s_{\mathsf{ref}}^{-1}, s)(G(s_{\mathsf{ref}}^{-1}, s))\right)^{-1} \\ = & w\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s)(G(s_{\mathsf{ref}}^{-1}, s)^{-1}) \\ = & w\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s)(G^{-1}(s_{\mathsf{ref}}, s)) \end{array}$$

□

Lemma 18. If $\mathsf{stable}(w)$, then $\mathsf{stable}(w^{-1})$.

Lemma 19. If $\mathsf{inhabited}(w\uparrow)$, then $\mathsf{inhabited}(w^{-1}\uparrow)$.

Lemma 20. If $G \in \mathsf{GK}(w^{-1}\uparrow)$, then:

$$\left(\mathbf{E}_{w\uparrow}(G^{-1})(s_{\mathsf{ref}\,0}^{-1}, s_0)(s_{\mathsf{ref}}^{-1}, s)\right)^{-1} \subseteq \mathbf{E}_{w^{-1}\uparrow}(G)(s_{\mathsf{ref}\,0}, s_0)(s_{\mathsf{ref}}, s)$$

Proof. By a straightforward coinduction.

□

Lemma 21. If $\mathsf{consistent}(w\uparrow)$, then $\mathsf{consistent}(w^{-1}\uparrow)$.

Theorem 3.

$$\frac{\Delta; \Gamma \vdash e_1 \sim e_2 : \sigma}{\Delta; \Gamma \vdash e_2 \sim e_1 : \sigma}$$

Proof. Given $\mathcal{N} \in \mathsf{TyNam}$, we get a stable w with $\Delta; \Gamma \vdash e_1 \sim_{w\uparrow} e_2 : \sigma$ as well as $w.\mathsf{N} \subseteq \mathcal{N}$. Using $w^{-1}.\mathsf{N} = w.\mathsf{N}$ and Lemma 18, it remains to show:

$$\Delta; \Gamma \vdash e_2 \sim_{w^{-1}\uparrow} e_1 : \sigma$$

With Lemmas 19 and 21, this in turn reduces to showing:

$$(\gamma_1 e_2, \gamma_2 e_1) \in \mathbf{E}_{w^{-1}\uparrow}(G)(s_{\mathsf{ref}}, s)(s_{\mathsf{ref}}, s)(\delta\sigma)$$

for any $s_{\mathsf{ref}}, s, \delta$ and $G \in \mathsf{GK}(w^{-1}\uparrow)$ and $(\gamma_1, \gamma_2) \in \mathsf{Env}(\delta\Gamma, G(s_{\mathsf{ref}}, s))$.

Note that $(\gamma_2, \gamma_1) \in \mathsf{Env}(\delta\Gamma, G(s_{\mathsf{ref}}, s)^{-1}) = \mathsf{Env}(\delta\Gamma, G^{-1}(s_{\mathsf{ref}}^{-1}, s))$ with the help of Lemma 13 and the definition of G^{-1}. Using Lemma 17, $\Delta; \Gamma \vdash e_1 \sim_{w\uparrow} e_2 : \sigma$ thus yields:

$$(\gamma_2 e_1, \gamma_1 e_2) \in \mathbf{E}_{w\uparrow}(G^{-1})(s_{\mathsf{ref}}^{-1}, s)(s_{\mathsf{ref}}^{-1}, s)(\delta\sigma)$$

We are done by Lemma 20. □

3.7.3 Compatibilities

Compatibilities are very useful lemmas stating that the language's constructs preserve our equivalence. For each typing rule we have a corresponding compatibility. Figure 3.17 lists them all. We have seen two of them—and their proof sketches—earlier in the simple setting (FIX and APP in Figure 3.9).

These proofs are somewhat tedious and, in general, similar to their counterparts in a KLR setting. One of the key differences lies in the proofs concerning functions, which we already sketched earlier. Here we show these proofs in detail, generalized to the full $\mathsf{F}^{\mu!}$ setting of this section.

Lemma 22 (Compatibility: App)**.**

$$\frac{\Delta;\Gamma \vdash p_1 \sim p_1' : \sigma_1 \to \sigma_2 \quad \Delta;\Gamma \vdash p_2 \sim p_2' : \sigma_1}{\Delta;\Gamma \vdash p_1\ p_2 \sim p_1'\ p_2' : \sigma_2}$$

Proof. Given a countably infinite set $\mathcal{N} \in \mathcal{P}(\mathsf{TyNam})$, we can split it into two countably infinite parts: $\mathcal{N} = \mathcal{N}_1 \uplus \mathcal{N}_2$. Using these to instantiate the first and second premise, respectively, we obtain two stable local worlds w_1 and w_2 whose product $w := w_1 \otimes w_2$ is defined and satisfies $w.\mathsf{N} \subseteq \mathcal{N}$. We use Lemma 10 to infer stability of w and consistency and inhabitance of $w{\uparrow}$. It thus remains to show

$$(\gamma(p_1\ p_2), \gamma'(p_1'\ p_2')) \in \mathbf{E}_{w\uparrow}(G)(s_{\mathsf{ref}}, s_1, s_2)(s_{\mathsf{ref}}, s_1, s_2)(\delta\sigma_2)$$

for substitutions $(\gamma,\ \gamma') \in \mathsf{Env}(\Gamma, G(s_{\mathsf{ref}}, s_1, s_2))$. The first instantiated premise lets us assume

$$(\gamma p_1, \gamma' p_1') \in \mathbf{E}_{w_1\uparrow}(G(-,-,s_2))(s_{\mathsf{ref}}, s_1)(s_{\mathsf{ref}}, s_1)(\delta(\sigma_1 \to \sigma_2)),$$

which Lemma 9 turns into

$$(\gamma p_1, \gamma' p_1') \in \mathbf{E}_{w\uparrow}(G)(s_{\mathsf{ref}}, s_1, s_2)(s_{\mathsf{ref}}, s_1, s_2)(\delta(\sigma_1 \to \sigma_2)).$$

By Lemma 11 it therefore suffices to show

$$(\bullet\ \gamma p_2, \bullet\ \gamma' p_2') \in \mathbf{K}_{w\uparrow}(G')(s_{\mathsf{ref}}, s_1, s_2)(s'_{\mathsf{ref}}, s_1', s_2')(\delta\sigma_1 \to \delta\sigma_2)(\delta\sigma_2)$$

for any $G' \supseteq G$ and any $(s'_{\mathsf{ref}}, s_1', s_2') \sqsupseteq_{\mathrm{pub}} (s_{\mathsf{ref}}, s_1, s_2)$. Given related values $(v_1, v_1') \in \overline{G'(s'_{\mathsf{ref}}, s_1', s_2')}(\delta\sigma_1 \to \delta\sigma_2)$, we therefore must show:

$$(v_1\ \gamma p_2, v_1'\ \gamma' p_2') \in \mathbf{E}_{w\uparrow}(G')(s_{\mathsf{ref}}, s_1, s_2)(s'_{\mathsf{ref}}, s_1', s_2')(\delta\sigma_2)$$

Now, the second instantiated premise lets us assume

$$(\gamma p_2, \gamma' p_2') \in \mathbf{E}_{w_2\uparrow}(G'(-,-,s_2'))(s_{\mathsf{ref}}, s_1)(s'_{\mathsf{ref}}, s_1')(\delta\sigma_1),$$

which Lemma 9 turns into

$$(\gamma p_2, \gamma' p_2') \in \mathbf{E}_{w\uparrow}(G')(s_{\mathsf{ref}}, s_1, s_2)(s'_{\mathsf{ref}}, s_1', s_2')(\delta\sigma_1).$$

$$\frac{\Delta \vdash \Gamma \quad x{:}\sigma \in \Gamma}{\Delta;\Gamma \vdash x \sim x : \sigma} \qquad \frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash n \sim n : \mathsf{nat}}$$

$$\frac{\Delta;\Gamma \vdash p_0 \sim p_0' : \mathsf{nat} \quad \Delta;\Gamma \vdash p_1 \sim p_1' : \sigma \quad \Delta;\Gamma \vdash p_2 \sim p_2' : \sigma}{\Delta;\Gamma \vdash \mathsf{ifz}\ p_0\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \sim \mathsf{ifz}\ p_0'\ \mathsf{then}\ p_1'\ \mathsf{else}\ p_2' : \sigma}$$

$$\frac{\Delta;\Gamma \vdash p_1 \sim p_1' : \sigma_1 \quad \Delta;\Gamma \vdash p_2 \sim p_2' : \sigma_2}{\Delta;\Gamma \vdash \langle p_1, p_2\rangle \sim \langle p_1', p_2'\rangle : \sigma_1 \times \sigma_2}$$

$$\frac{\Delta;\Gamma \vdash p \sim p' : \sigma_1 \times \sigma_2}{\Delta;\Gamma \vdash p.1 \sim p'.1 : \sigma_1} \qquad \frac{\Delta;\Gamma \vdash p \sim p' : \sigma_1 \times \sigma_2}{\Delta;\Gamma \vdash p.2 \sim p'.2 : \sigma_2}$$

$$\frac{\Delta;\Gamma \vdash p \sim p' : \sigma_1 \quad \Delta;\Gamma \vdash \sigma_2}{\Delta;\Gamma \vdash \mathsf{inl}_{\sigma_2}\ p \sim \mathsf{inl}_{\sigma_2}\ p' : \sigma_1 + \sigma_2} \qquad \frac{\Delta;\Gamma \vdash \sigma_1 \quad \Delta;\Gamma \vdash p \sim p' : \sigma_2}{\Delta;\Gamma \vdash \mathsf{inr}_{\sigma_1}\ p \sim \mathsf{inr}_{\sigma_2}\ p' : \sigma_1 + \sigma_2}$$

$$\frac{\Delta;\Gamma \vdash p_0 \sim p_0' : \sigma_1 + \sigma_2 \quad \Delta;\Gamma, x{:}\sigma_1 \vdash p_1 \sim p_1' : \sigma \quad \Delta;\Gamma, x{:}\sigma_2 \vdash p_2 \sim p_2' : \sigma}{\Delta;\Gamma \vdash \mathsf{case}\ p_0\ (x.\ p_1)\ (x.\ p_2) \sim \mathsf{case}\ p_0'\ (x.\ p_1')\ (x.\ p_2') : \sigma}$$

$$\frac{\Delta;\Gamma, f{:}(\sigma_1 \to \sigma_2), x{:}\sigma_1 \vdash p \sim p' : \sigma_2}{\Delta;\Gamma \vdash \mathsf{fix}\, f(x{:}\sigma_1){:}\sigma_2.\, p \sim \mathsf{fix}\, f(x{:}\sigma_1){:}\sigma_2.\, p' : \sigma_1 \to \sigma_2}$$

$$\frac{\Delta;\Gamma \vdash p_1 \sim p_1' : \sigma_1 \to \sigma_2 \quad \Delta;\Gamma \vdash p_2 \sim p_2' : \sigma_1}{\Delta;\Gamma \vdash p_1\ p_2 \sim p_1'\ p_2' : \sigma_2}$$

$$\frac{\Delta, \alpha;\Gamma \vdash p \sim p' : \sigma}{\Delta;\Gamma \vdash \Lambda\alpha.\, p \sim \Lambda\alpha.\, p' : \forall\alpha.\, \sigma} \qquad \frac{\Delta;\Gamma \vdash p \sim p' : \forall\alpha.\, \sigma_1 \quad \Delta;\Gamma \vdash \sigma_2}{\Delta;\Gamma \vdash p\ [\sigma_2] \sim p'\ [\sigma_2] : \sigma_1[\sigma_2/\alpha]}$$

$$\frac{\Delta;\Gamma \vdash \sigma_1 \quad \Delta;\Gamma \vdash p \sim p' : \sigma_2[\sigma_1/\alpha]}{\Delta;\Gamma \vdash \mathsf{pack}\ \langle\sigma_1, p\rangle\ \mathsf{as}\ \exists\alpha.\, \sigma_2 \sim \mathsf{pack}\ \langle\sigma_1, p'\rangle\ \mathsf{as}\ \exists\alpha.\, \sigma_2 : \exists\alpha.\, \sigma_2}$$

$$\frac{\Delta;\Gamma \vdash p_1 \sim p_1' : \exists\alpha.\, \sigma_1 \quad \Delta, \alpha;\Gamma, x{:}\sigma_1 \vdash p_2 \sim p_2' : \sigma_2 \quad \Delta;\Gamma \vdash \sigma_2}{\Delta;\Gamma \vdash \mathsf{unpack}\ p_1\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ p_2 \sim \mathsf{unpack}\ p_1'\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ p_2' : \sigma_2}$$

$$\frac{\Delta;\Gamma \vdash p \sim p' : \sigma[\mu\alpha.\, \sigma/\alpha]}{\Delta;\Gamma \vdash \mathsf{roll}_{\mu\alpha.\,\sigma}\ p \sim \mathsf{roll}_{\mu\alpha.\,\sigma}\ p' : \mu\alpha.\, \sigma} \qquad \frac{\Delta;\Gamma \vdash p' : \mu\alpha.\, \sigma}{\Delta;\Gamma \vdash \mathsf{unroll}\ p \sim \mathsf{unroll}\ p' : \sigma[\mu\alpha.\, \sigma/\alpha]}$$

$$\frac{\Delta;\Gamma \vdash p \sim p' : \sigma}{\Delta;\Gamma \vdash \mathsf{ref}\ p \sim \mathsf{ref}\ p' : \mathsf{ref}\,\sigma} \qquad \frac{\Delta;\Gamma \vdash p_1 \sim p_1' : \mathsf{ref}\,\sigma \quad \Delta;\Gamma \vdash p_2 \sim p_2' : \sigma}{\Delta;\Gamma \vdash p_1 := p_2 \sim p_1' := p_2' : \mathsf{unit}}$$

$$\frac{\Delta;\Gamma \vdash p \sim p' : \mathsf{ref}\,\sigma}{\Delta;\Gamma \vdash\, !p \sim\, !p' : \sigma} \qquad \frac{\Delta;\Gamma \vdash p_1 \sim p_1' : \mathsf{ref}\,\sigma \quad \Delta;\Gamma \vdash p_2 \sim p_2' : \mathsf{ref}\,\sigma}{\Delta;\Gamma \vdash p_1 == p_2 \sim p_1' == p_2' : \mathsf{nat}}$$

Figure 3.17: Compatibilities

Again by Lemma 11 it suffices to show

$$(v_1\ \bullet, v_1'\ \bullet) \in \mathbf{K}_{w\uparrow}(G'')(s_{\mathsf{ref}}, s_1, s_2)(s''_{\mathsf{ref}}, s''_1, s''_2)(\delta\sigma_1)(\delta\sigma_2)$$

for any $G'' \supseteq G'$ and any $(s''_{\mathsf{ref}}, s''_1, s''_2) \sqsupseteq (s'_{\mathsf{ref}}, s'_1, s'_2)$ with $(s''_{\mathsf{ref}}, s''_1, s''_2) \sqsupseteq_{\text{pub}} (s_{\mathsf{ref}}, s_1, s_2)$. Given related values $(v_2, v_2') \in \overline{G''(s''_{\mathsf{ref}}, s''_1, s''_2)}(\delta\sigma_1)$, we therefore must show

$$(v_1\ v_2, v_1'\ v_2') \in \mathbf{E}_{w\uparrow}(G'')(s_{\mathsf{ref}}, s_1, s_2)(s''_{\mathsf{ref}}, s''_1, s''_2)(\delta\sigma_2),$$

which is the interesting part of the proof. (In fact, the reasoning up to here is not specific to function application but common to many other program forms involving subterms. In our formal Coq development, some of the boilerplate is moved into shared lemmas.)

To show this last goal we appeal to the CALL disjunct of $\mathbf{E}$. Accordingly, it suffices to show three things:

- $(v_1, v_1') \in \overline{G''(s''_{\mathsf{ref}}, s''_1, s''_2)}(\delta\sigma_1 \to \delta\sigma_2)$
- $(v_2, v_2') \in \overline{G''(s''_{\mathsf{ref}}, s''_1, s''_2)}(\delta\sigma_1)$
- $(\bullet, \bullet) \in \mathbf{K}_{w\uparrow}(G''')(s_{\mathsf{ref}}, s_1, s_2)(s''')(\delta\sigma_2)(\delta\sigma_2)$ for any $G''' \supseteq G''$ and any $s''' \sqsupseteq (s''_{\mathsf{ref}}, s''_1, s''_2)$ with $s''' \sqsupseteq_{\text{pub}} (s_{\mathsf{ref}}, s_1, s_2)$

The second was an assumption, the first follows from the assumptions by monotonicity, and the third, which talks about empty continuations, is easy to verify. □

Lemma 23 (Compatibility: Fix)**.**

$$\frac{\Delta; \Gamma, f{:}\sigma_1 \to \sigma_2, x{:}\sigma_1 \vdash e_1 \sim e_2 : \sigma_2}{\Delta; \Gamma \vdash \mathsf{fix}\, f(x).\, e_1 \sim \mathsf{fix}\, f(x).\, e_2 : \sigma_1 \to \sigma_2}$$

Proof. Given $\mathcal{N} \in \mathcal{P}(\mathsf{TyNam})$, the premise yields a stable w_1 such that $w_1.\mathsf{N} \subseteq \mathcal{N}$ and $\Delta; \Gamma, f{:}\sigma_1 \to \sigma_2, x{:}\sigma_1 \vdash e_1 \sim_{w_1\uparrow} e_2 : \sigma_2$. We choose the local world in which we prove the two functions related to be $w := w_1 \otimes w_2$, where w_2 is defined as the following trivially stable single-state world:

$$\begin{array}{lcl}
w_2.\mathsf{T}.\mathsf{S} & := & \{1\} \\
w_2.\mathsf{L}(_)(_)(R)(\tau) & := & \{(\gamma_1 \mathsf{fix}\, f(x).\, e_1, \gamma_2 \mathsf{fix}\, f(x).\, e_2) \mid \exists \delta \in \Delta \to \mathsf{CTy}. \\
 & & \qquad \tau = \delta\sigma_1 \to \delta\sigma_2 \wedge (\gamma_1, \gamma_2) \in \mathsf{Env}(\delta\Gamma, R)\} \\
w_2.\mathsf{H}(_)(_)(_) & := & \{(\emptyset, \emptyset)\} \\
w_2.\mathsf{N} & := & \emptyset
\end{array}$$

With Lemma 10 and $w.\mathsf{N} = (w_1 \otimes w_2).\mathsf{N} = w_1.\mathsf{N} \subseteq \mathcal{N}$, it suffices to show:

$$\Delta; \Gamma \vdash \mathsf{fix}\, f(x).\, e_1 \sim_{w\uparrow} \mathsf{fix}\, f(x).\, e_2 : \sigma_1 \to \sigma_2$$

We need to show $\mathsf{inhabited}(w\uparrow)$ and $\mathsf{consistent}(w\uparrow)$—the rest follows by construction of w_2. Regarding inhabitance, by Lemma 10 it suffices to show $\mathsf{inhabited}(w_2\uparrow)$, which

is witnessed by state $(\emptyset, 1)$. Regarding consistency, using the same lemma, we could simply show $\mathsf{consistent}(w_2{\uparrow})$. However, since $w_2.\mathsf{L}$ refers to e_2 and e_2', there is no hope to show this. Instead, we have to prove $\mathsf{consistent}(w{\uparrow})$ directly.

The part of $\mathsf{consistent}(w{\uparrow})$ concerning universal types follows easily by Lemma 9 from $\mathsf{consistent}(w_1{\uparrow})$, because $w_2.\mathsf{L}$ is empty at universal types.

Concerning arrow types, we are given $G, s_{\mathsf{ref}}, s_1, s_2, \sigma_1', \sigma_2'$, related functions

$$(v_1, v_2) \in w{\uparrow}.\mathsf{L}(s_{\mathsf{ref}}, s_1, s_2)(G(s_{\mathsf{ref}}, s_1, s_2))(\sigma_1' \to \sigma_2')$$

and related arguments $(v_1', v_2') \in \overline{G(s_{\mathsf{ref}}, s_1, s_2)}(\sigma_1')$, and must show:

$$(\mathit{step}(v_1\ v_1'), \mathit{step}(v_2\ v_2')) \in \mathbf{E}_{w{\uparrow}}(G)(s_{\mathsf{ref}}, s_1, s_2)(s_{\mathsf{ref}}, s_1, s_2)(\sigma_2')$$

If the functions are related by $w_1{\uparrow}$,

$$(v_1, v_2) \in w_1{\uparrow}.\mathsf{L}(s_{\mathsf{ref}}, s_1)(G(s_{\mathsf{ref}}, s_1, s_2))(\sigma_1' \to \sigma_2')$$

then the claim follows from $\mathsf{consistent}(w_1{\uparrow})$ with Lemma 9.

Otherwise, the functions are necessarily from w_2:

$$(v_1, v_2) \in w_2.\mathsf{L}(s_{\mathsf{ref}})(s_2)(G(s_{\mathsf{ref}}, s_1, s_2))(\sigma_1' \to \sigma_2')$$

Hence $\sigma_1' \to \sigma_2' = \delta\sigma_1 \to \delta\sigma_2$ and $v_1 = \gamma_1 \mathsf{fix}\, f(x).\, e_1$ and $v_2 = \gamma_2 \mathsf{fix}\, f(x).\, e_2$ for some δ and $(\gamma_1, \gamma_2) \in \mathsf{Env}(\delta\Gamma, G(s_{\mathsf{ref}}, s_1, s_2))$. Writing γ_i' for $\gamma_i, f{\mapsto}v_i, x{\mapsto}v_i'$, it remains to show

$$(\gamma_1' e_1, \gamma_2' e_2) \in \mathbf{E}_{w{\uparrow}}(G)(s_{\mathsf{ref}}, s_1, s_2)(s_{\mathsf{ref}}, s_1, s_2)(\delta\sigma_2),$$

which reduces by Lemma 9 to

$$(\gamma_1' e_1, \gamma_2' e_2) \in \mathbf{E}_{w_1{\uparrow}}(G(-, -, s_2))(s_{\mathsf{ref}}, s_1)(s_{\mathsf{ref}}, s_1)(\delta\sigma_2).$$

This follows from the instantiated premise if we can show:

$$(\gamma_1', \gamma_2') \in \mathsf{Env}(\delta(\Gamma, f{:}\sigma_1 \to \sigma_2, x{:}\sigma_1), G(s_{\mathsf{ref}}, s_1, s_2))$$

Given what we know about γ_1, γ_2, it suffices to focus on their extensions. We already know that $(v_1', v_2') \in \overline{G(s_{\mathsf{ref}}, s_1, s_2)}(\delta\sigma_1)$. So it remains to show

$$(\gamma_1 \mathsf{fix}\, f(x).\, e_1, \gamma_2 \mathsf{fix}\, f(x).\, e_2) \in \overline{G(s_{\mathsf{ref}}, s_1, s_2)}(\delta\sigma_1 \to \delta\sigma_2),$$

which, exploiting that $G(-, s_1, -) \in \mathsf{GK}(w_2{\uparrow})$, we do by showing

$$(\gamma_1 \mathsf{fix}\, f(x).\, e_1, \gamma_2 \mathsf{fix}\, f(x).\, e_2) \in w_2{\uparrow}.\mathsf{L}(s_{\mathsf{ref}}, s_2)(G(s_{\mathsf{ref}}, s_1, s_2))(\delta\sigma_1 \to \delta\sigma_2).$$

Indeed, this holds by construction of w_2. □

3.7.4 Congruency

By an easy induction, the compatibility lemmas from Figure 3.17 yield reflexivity for well-typed programs.

Theorem 4 (Reflexivity)**.**

$$\frac{\Delta;\Gamma \vdash p : \sigma}{\Delta;\Gamma \vdash p \sim p : \sigma}$$

We also have transitivity of $\sim$, hence it is a proper equivalence relation (we already proved symmetry in Section 3.7.2).

Theorem 5 (Transitivity)**.**

$$\frac{\Delta;\Gamma \vdash p_1 \sim p_2 : \sigma \quad \Delta;\Gamma \vdash p_2 \sim p_3 : \sigma}{\Delta;\Gamma \vdash p_1 \sim p_3 : \sigma}$$

Proof. The subject of Section 3.9. □

Lemma 24 (Context closure)**.**

$$\frac{\Delta;\Gamma \vdash p_1 \sim p_2 : \sigma \quad \vdash C : (\Delta;\Gamma;\sigma) \rightsquigarrow (\Delta';\Gamma';\sigma')}{\Delta';\Gamma' \vdash C[p_1] \sim C[p_2] : \sigma'}$$

Proof. By induction on the derivation of the context typing, in each case using the corresponding compatibility lemma. For contexts containing subterms we also need Theorem 4. □

Lemma 25 (Substitutivity)**.**

$$\frac{\Delta;\Gamma, x{:}\tau' \vdash e_1 \sim e_2 : \tau \qquad \Delta;\Gamma \vdash v_1 \sim v_2 : \tau'}{\Delta;\Gamma \vdash e_1[v_1/x] \sim e_2[v_2/x] : \tau}$$

3.7.5 Soundness

Lemma 26 (Adequacy)**.** If $\vdash p_1 \sim p_2 : \tau$, then for any heaps h_1, h_2:

1. either $\langle h_1; |p_1| \rangle \uparrow \wedge \langle h_2; |p_2| \rangle \uparrow$,
2. or $\exists h_1', h_2', v_1, v_2.\ \langle h_1; |p_1| \rangle \hookrightarrow^* \langle h_1'; v_1 \rangle \wedge \langle h_2; |p_2| \rangle \hookrightarrow^* \langle h_2'; v_2 \rangle$.

Proof. We instantiate the premise with TyNam and get w with $\vdash p_1 \sim_{w\uparrow} p_2 : \tau$. Using the least global knowledge (Lemma 7), inhabitance of $w\uparrow$ tells us that there is s_0 with $(\emptyset, \emptyset) \in w\uparrow.\mathsf{H}(s_0)([w\uparrow](s_0))$. Now we use this state to obtain

$$(e_1, e_2) \in \mathbf{E}_{w\uparrow}([w\uparrow])(s_0)(s_0)(\tau),$$

which by key Lemma 12 implies

$$(e_1, e_2) \in \mathbf{E}^{\lambda s.\emptyset}_{w\uparrow}([w\uparrow])(s_0)(s_0)(\tau).$$

Instantiating this with the empty heaps $(\emptyset, \emptyset) \in w{\uparrow}.\mathsf{H}(s_0)([w{\uparrow}](s_0))$ as well as with empty frame heaps, yields three cases. Cases DIV and EVAL imply the goal. Case CALL cannot possibly hold, because the called functions would be related by the empty relation. □

Finally, combining adequacy and congruency, we can show our main soundness theorem: for well-typed programs, our equivalence relation is included in contextual equivalence.

Theorem 6 (Soundness)**.** If $\Delta; \Gamma \vdash p_1 : \sigma$ and $\Delta; \Gamma \vdash p_2 : \sigma$, then:

$$\Delta; \Gamma \vdash p_1 \sim p_2 : \sigma \implies \Delta; \Gamma \vdash p_1 \sim_{\text{ctx}} p_2 : \sigma$$

Proof. Suppose $\Delta; \Gamma \vdash p_1 \sim p_2 : \sigma$ as well as $\vdash C : (\Delta; \Gamma; \sigma) \rightsquigarrow (\epsilon; \epsilon; \tau)$. By Lemma 24 we have $\vdash C[p_1] \sim C[p_2] : \tau$. By adequacy (Lemma 26) we have $\langle h; |C[p_1]|\rangle{\uparrow} \iff \langle h; |C[p_2]|\rangle{\uparrow}$ for any h, so we are done. □

3.8 Examples

In this section, we present several example PB equivalence proofs. Here we will not actually rely on the ability of local knowledges and heap relations to depend on the state of the reference world W_{ref}. Consequently, the stability property in the definition of program equivalence will be trivially satisfied (by choosing $s' := s$). However, dependent worlds are of critical importance in the transitivity proof discussed in Section 3.9.

3.8.1 Well-Bracketed State Change

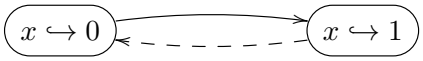
Recall the example from Section 2.2.3 and its high-level proof-sketch using an STS. We will now show in some formal detail how this proof is done using our method. Concretely, we prove $\vdash v_1 \sim e_2 : \tau$, where:

$$\begin{array}{lcl} \tau & := & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{nat} \\ v_1 & := & \lambda f.\, (f\ \langle\rangle; f\ \langle\rangle; 1) \\ e_2 & := & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ v_2 \\ v_2 & := & \lambda f.\, (x := 0; f\ \langle\rangle; x := 1; f\ \langle\rangle; !x) \end{array}$$

Constructing a Suitable World We construct a local world w that we will then show to be consistent and to relate v_1 and e_2.

$$\begin{array}{ll} w.\mathsf{T}.\mathsf{S} & := \mathsf{Loc} \xrightarrow{\text{fin}} \{0,1\} \subset \mathsf{Heap} \\ w.\mathsf{T}.\sqsubseteq & := \{(s, s') \mid \text{dom}(s) \subseteq \text{dom}(s')\} \\ w.\mathsf{T}.\sqsubseteq_{\text{pub}} & := \{(s, s') \in w.\sqsubseteq \mid \forall (l,1) \in s.\ (l,1) \in s'\} \\ w.\mathsf{L}(\_, s)(G)(\tau) & := \{(v_1, v_2[l/x]) \mid l \in \text{dom}(s)\} \\ w.\mathsf{H}(\_, s)(G) & := \{(\emptyset, s)\} \\ w.\mathsf{N} & := \emptyset \end{array}$$

The STS that we build into w is essentially the one from Section 2.2.3:



A state $s \in w.\mathsf{S}$ is to be understood as follows: for each running instance of e_2, identified by the location l_x that that instance initially allocated, $s(l_x)$ says whether the instance is in the (pictorially) left state (l_x points to 0) or in the right one (l_x points to 1). Accordingly, the heap relation $w.\mathsf{H}$ at state s is just $\{(\emptyset, s)\}$. Finally, the local knowledge $w.\mathsf{L}$ at state s relates v_1 with $v_2[l_x/x]$ for any location l_x belonging to an instance. Finally, since the programs don't involve abstract types, we can define $w.\mathsf{N}$ to be empty.

It is easy to see that $w \in \mathrm{LWorld}$. In particular, $w.\mathsf{L}$ and $w.\mathsf{H}$ are monotone as required. Note that $\mathsf{stable}(w)$ (the dependency is vacuous) and that $\mathsf{inhabited}(w{\uparrow})$ for $s_0 = (\emptyset, \emptyset)$. To show $\vdash v_1 \sim_{w\uparrow} e_2 : \tau$, two parts remain.

Proving Its Consistency Establishing $\mathsf{consistent}(w{\uparrow})$ is the real meat of the proof. Consider two functions related by $w{\uparrow}.\mathsf{L}$ at a state $(s^1_{\mathsf{ref}}, s_1)$. Clearly, one is v_1 and the other is $v_2[l/x]$ for some $l \in \mathrm{dom}(s_1)$. Now suppose we are given related arguments $(v_1', v_2') \in \overline{G_1(s^1_{\mathsf{ref}}, s_1)}(\mathsf{unit} \to \mathsf{unit})$. We need to show:

$$((v_1'\ \langle\rangle; v_1'\ \langle\rangle; 1), (l := 0; v_2'\ \langle\rangle; l := 1; v_2'\ \langle\rangle; !l)) \in \mathbf{E}_{w\uparrow}(G_1)(s^1_{\mathsf{ref}}, s_1)(s^1_{\mathsf{ref}}, s_1)(\mathsf{nat})$$

Note that for $(h_1, h_2) \in w.\mathsf{H}(s^1_{\mathsf{ref}}, s_1)(G(s^1_{\mathsf{ref}}, s_1))$ we know by construction that $h_1 = \emptyset$ and $h_2 = s_1$. Consequently, for any frame heaps $h_1^{\mathrm{F}}, h_2^{\mathrm{F}}$, we have

$$\begin{array}{ll} & \langle h_1 \sqcup h_1^{\mathrm{F}}; (v_1'\ \langle\rangle; v_1'\ \langle\rangle; 1)\rangle \\ \hookrightarrow^* & \langle h_1 \sqcup h_1^{\mathrm{F}}; (v_1'\ \langle\rangle; v_1'\ \langle\rangle; 1)\rangle \end{array}$$

$$\begin{array}{ll} & \langle h_2 \sqcup h_2^{\mathrm{F}}; (l := 0; v_2'\ \langle\rangle; l := 1; v_2'\ \langle\rangle; !l)\rangle \\ \hookrightarrow^* & \langle (s_1 \backslash l) \sqcup [l \mapsto 0] \sqcup h_2^{\mathrm{F}}; (v_2'\ \langle\rangle; l := 1; v_2'\ \langle\rangle; !l)\rangle \end{array}$$

where $s_1 \backslash l$ denotes the restriction of s_1 to domain $\mathrm{dom}(s_1) \setminus \{l\}$.

By the CALL disjunct in the definition of $\mathbf{E}_{w\uparrow}$ it now suffices to find $s_1' \sqsupseteq s_1$ such that:

1. $(\emptyset, (s_1 \backslash l) \sqcup [l \mapsto 0]) \in w.\mathsf{H}(s^1_{\mathsf{ref}}, s_1')(G_1(s^1_{\mathsf{ref}}, s_1'))$
2. $((\bullet; v_1'\ \langle\rangle; 1), (\bullet; l := 1; v_2'\ \langle\rangle; !l)) \in \mathbf{K}_{w\uparrow}(G_1)(s^1_{\mathsf{ref}}, s_1)(s^1_{\mathsf{ref}}, s_1')(\mathsf{unit}, \mathsf{nat})$

Naturally, we pick $s_1' = (s_1 \backslash l) \sqcup [l \mapsto 0]$. Note that $s_1' \sqsupseteq s_1$ because their domains coincide. Then (1) holds by construction of w and it remains to show (2).

After unfolding the definition of $\mathbf{K}$ and being given $(s^2_{\mathsf{ref}}, s_2) \sqsupseteq_{\mathrm{pub}} (s^1_{\mathsf{ref}}, s_1')$ as well as $G_2 \supseteq G_1$, we repeat the previous chain of arguments one more time and arrive at the goal of finding $s_2' \sqsupseteq s_2$ such that:

3. $(\emptyset, (s_2 \backslash l) \sqcup [l \mapsto 1]) \in w.\mathsf{H}(s^2_{\mathsf{ref}}, s_2')(G_2(s^2_{\mathsf{ref}}, s_2'))$

4. $((\bullet; 1), (\bullet; !l)) \in \mathbf{K}_{w\uparrow}(G_2)(s^1_{\mathsf{ref}}, s_1)(s^2_{\mathsf{ref}}, s'_2)(\mathsf{unit}, \mathsf{nat})$

Naturally, we pick $s'_2 = (s_2 \backslash l) \sqcup [l \mapsto 1] \sqsupseteq s_2$. Then (3) holds by construction of w and it remains to show (4).

We observe that, for any $(s^3_{\mathsf{ref}}, s_3) \sqsupseteq_{\text{pub}} (s^2_{\mathsf{ref}}, s'_2)$, it must be that $s_3(l) = 1$ since $s'_2(l) = 1$. Hence for $(h'_1, h'_2) \in w.\mathsf{H}(s^3_{\mathsf{ref}}, s_3)(G_3(s^3_{\mathsf{ref}}, s_3))$ we know by construction $h'_2(l) = 1$. Consequently, for any frame heaps $h_1^{\mathrm{F}'}, h_2^{\mathrm{F}'}$, we have:

$$\langle h'_1 \sqcup h_1^{\mathrm{F}'}; (\langle\rangle; 1)\rangle \hookrightarrow^* \langle h'_1 \sqcup h_1^{\mathrm{F}'}; 1\rangle$$

$$\langle h'_2 \sqcup h_2^{\mathrm{F}'}; (\langle\rangle; !l)\rangle \hookrightarrow^* \langle h'_2 \sqcup h_2^{\mathrm{F}'}; 1\rangle$$

Hence we are done if we can show $(s^3_{\mathsf{ref}}, s_3) \sqsupseteq_{\text{pub}} (s^1_{\mathsf{ref}}, s_1)$. Indeed, this is easy to verify.

Showing the Programs Related By It Given how we constructed our local world, this final goal is fairly easy to accomplish. Formally, we must show

$$(v_1, e_2) \in \mathbf{E}_{w\uparrow}(G)(s_{\mathsf{ref}}, s)(s_{\mathsf{ref}}, s)(\tau)$$

for any G, s_{ref}, s. Note that if $(h_1, h_2) \in w.\mathsf{H}(s_{\mathsf{ref}}, s)(G(s_{\mathsf{ref}}, s))$, then

$$\langle h_1; v_1\rangle \hookrightarrow^* \langle h_1; v_1\rangle$$

$$\langle h_2; e_2\rangle \hookrightarrow \langle h_2 \sqcup [l \mapsto 0]; v_2[l/x]\rangle$$

for some fresh l. It therefore suffices to find $s' \sqsupseteq_{\text{pub}} s$ such that the following hold:

5. $(h_1, h_2 \sqcup [l \mapsto 0]) \in w.\mathsf{H}(s_{\mathsf{ref}}, s')(G(s_{\mathsf{ref}}, s'))$

6. $(v_1, v_2[l/x]) \in \overline{G(s_{\mathsf{ref}}, s')}(\tau)$

We pick $s' = s \sqcup [l \mapsto 0]$. Note that s' is well-defined because l is fresh (so $l \notin \mathrm{dom}(s)$), and that $s' \sqsupseteq_{\text{pub}} s$ as required. To show (6), it suffices by definition of GK to show $(v_1, v_2[l/x]) \in w.\mathsf{L}(s_{\mathsf{ref}}, s')(G(s_{\mathsf{ref}}, s'))(\tau)$. This holds by construction of w and s', and so does (5).

Notice that our world in this proof was very simple—we didn't actually use the global knowledge parameter. An easy adaptation of the example programs, however, would necessitate that—for instance, if we were to replace the constants 0 and 1 with free variables a and b, respectively.

3.8.2 Twin Abstraction

This final example (originally due to Ahmed *et al.* [4]) demonstrates the interaction of local state with abstract types.

$$
\begin{array}{rcl}
\tau & := & \exists\alpha.\,\exists\beta.\,(\mathsf{unit} \to \alpha) \times (\mathsf{unit} \to \beta) \times (\alpha \times \beta \to \mathsf{nat}) \\
e_1 & := & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\
 & & \mathsf{pack}\ \langle \mathsf{nat}, \mathsf{pack}\ \langle \mathsf{nat}, \lambda\_.\, x := !x + 1; !x, \\
 & & \qquad\qquad\qquad\qquad \lambda\_.\, x := !x + 1; !x, \\
 & & \qquad\qquad\qquad\qquad \lambda p.\, p.1 = p.2 \rangle\rangle \\
e_2 & := & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\
 & & \mathsf{pack}\ \langle \mathsf{nat}, \mathsf{pack}\ \langle \mathsf{nat}, \lambda\_.\, x := !x + 1; !x, \\
 & & \qquad\qquad\qquad\qquad \lambda\_.\, x := !x + 1; !x, \\
 & & \qquad\qquad\qquad\qquad \lambda p.\, 0 \rangle\rangle
\end{array}
$$

Both e_1 and e_2 return a name generator ADT consisting of two abstract types α and β, together with a function for generating a fresh name of type α, a function for generating a fresh name of type β, and a function for comparing an α name and a β name for equality. Both implementations represent names as integers, and in e_1, the comparison operation really tests the names for equality. In e_2, however, the comparison just always returns false right away. Nevertheless, the two programs are contextually equivalent because the α names and the β names are generated by the *same* underlying integer counter, and thus no value can be both an α name and a β name at the same time.

We now show the construction of a local world w that can be used to prove $\vdash e_1 \sim e_2 : \tau$. Let a countably infinite $\mathcal{N} \in \mathcal{P}(\mathsf{TyNam})$ be given. Since Loc is also countably infinite, we can think of $\mathcal{N}$ as being split into $\{\mathbf{n}_l^\alpha \mid l \in \mathsf{Loc}\} \uplus \{\mathbf{n}_l^\beta \mid l \in \mathsf{Loc}\}$.

We define w as follows:

$$
\begin{array}{ll}
w.\mathsf{T}.\mathsf{S} & := \{s \in \mathsf{Loc} \times \mathsf{Loc} \xrightarrow{\text{fin}} \mathcal{P}(\mathbb{N}_{>0}) \times \mathcal{P}(\mathbb{N}_{>0}) \mid \\
& \qquad \mathrm{dom}(s) \text{ partial bijection } \wedge \\
& \qquad \forall (l_1, l_2, S_1, S_2) \in s.\ S_1 \cap S_2 = \emptyset\} \\
w.\mathsf{T}.\sqsubseteq & := \{(s, s') \mid \forall (l_1, l_2, S_1, S_2) \in s.\ \exists S_1' \supseteq S_1, S_2' \supseteq S_2.\ (l_1, l_2, S_1', S_2') \in s'\} \\
w.\mathsf{T}.\sqsubseteq_{\text{pub}} & := w.\mathsf{T}.\sqsubseteq
\end{array}
$$

$$
\begin{array}{ll}
w.\mathsf{L}(s_{\mathsf{ref}}, s)(G) & := \{(\mathbf{n}_{l_1}^{\alpha}, n, n) \mid \exists l_2, S_1, S_2.\ (l_1, l_2, S_1, S_2) \in s \wedge n \in S_1\} \\
& \uplus\ \{(\mathbf{n}_{l_1}^{\beta}, n, n) \mid \exists l_2, S_1, S_2.\ (l_1, l_2, S_1, S_2) \in s \wedge n \in S_2\} \\
& \uplus\ \{((\mathsf{unit} \to \mathbf{n}_{l_1}^{\alpha}), {+\!+}l_1, {+\!+}l_2) \mid (l_1, l_2) \in \mathrm{dom}(s)\} \\
& \uplus\ \{((\mathsf{unit} \to \mathbf{n}_{l_1}^{\beta}), {+\!+}l_1, {+\!+}l_2) \mid (l_1, l_2) \in \mathrm{dom}(s)\} \\
& \uplus\ \{((\mathbf{n}_{l_1}^{\alpha} \times \mathbf{n}_{l_1}^{\beta} \to \mathsf{nat}), (\lambda p.\, p.1 = p.2), (\lambda p.\, 0)) \mid \exists l_2.\ (l_1, l_2) \in \mathrm{dom}(s)\}
\end{array}
$$

$$
\begin{array}{ll}
w.\mathsf{H}(s_{\mathsf{ref}}, s)(G) & := \{(h_1, h_2) \mid \\
& \qquad \mathrm{dom}(h_1) = \{l_1 \mid \exists l_2.\ (l_1, l_2) \in \mathrm{dom}(s)\} \wedge \\
& \qquad \mathrm{dom}(h_2) = \{l_2 \mid \exists l_1.\ (l_1, l_2) \in \mathrm{dom}(s)\} \wedge \\
& \qquad \forall (l_1, l_2, S_1, S_2) \in s.\ h_1(l_1) = h_2(l_2) = max(\{0\} \uplus S_1 \uplus S_2)\}
\end{array}
$$

$$
w.\mathsf{N} := \mathcal{N}
$$

Here, ${+\!+}l$ is short for $\lambda\_.\, l := !l + 1; !l$.

Similar to the world construction in Section 3.8.1, states $s \in w.\mathsf{S}$ are functions defined for those locations (l_1, l_2) that, intuitively, were allocated in an instance of e_1 and e_2, respectively. They are mapped to sets S_1 and S_2 of positive integers, representing[5] the current inhabitants of the abstract types α and β, respectively, for that instance. The crucial invariant here is that S_1 and S_2 are always disjoint. The local knowledge $w.\mathsf{L}$ declares e_1's functions equivalent to those of e_2; it also defines the meaning of type $\mathbf{n}_{l_1}^{\alpha}$ as the identity relation restricted to those numbers that inhabit α in the instance pair identified by l_1 (and similarly for $\mathbf{n}_{l_1}^{\beta}$ and β).

According to this interpretation, the transition relation only allows S_1 and S_2 to grow (the distinction between public and private transitions is not needed for this example). Finally, $w.\mathsf{H}$ says that the related heaps at state s are any (h_1, h_2) where h_i contains exactly those locations allocated in instances of e_i, and each such location stores the largest value handed out so far (no matter if at α or β). This latter condition is critical to ensure that S_1 and S_2 stay disjoint in each instance.

Using this world, it is straightforward to finish the proof.

[5] To keep the definitions as simple as possible, the state space includes some states that actual program behaviour cannot result in (but that nevertheless are consistent with the property we want to prove).

3.8.3 World Generator

As explained in Section 3.6.7, and as one may observe from the previous examples, worlds must often describe "n-ary" state spaces, where each state consists of n copies of states drawn from some simpler state space, one copy for each dynamic instance of the object or ADT. Thus, it would be convenient if one were able to reason about program equivalence under the degenerate case of a single copy (*i.e.,* $n = 1$). Fortunately, it is not hard to (*i*) define a *world generator* that, given a single-instance world, automatically performs the multiplexing; and (*ii*) show that consistency of that single-instance world implies equivalence consistency of the multiplexed world.

Here we present the definition and properties of such a world generator. In the next section, we use it to simplify the proof of (a variant of) the example from Section 3.8.1

$$\text{NLWorld} := \{\mathcal{W} \in \text{Names} \to \text{LWorld} \mid \forall \mathcal{N}.\ \mathcal{W}(\mathcal{N}).\mathsf{N} \subseteq \mathcal{N}\}$$

Definition 9 (World generator)**.** We define $\mathbf{G} \in \text{NLWorld} \to \text{NLWorld}$ as follows.

$$\begin{array}{ll}
\mathbf{G}(\mathcal{W})(\mathcal{N}).\mathsf{T}.\mathsf{S} & := \{(s_1, \ldots, s_n) \mid n \in \mathbb{N} \wedge \forall i \in \{1, \ldots, n\}.\ s_i \in \mathcal{W}(\mathcal{N}_i).\mathsf{S}\} \\
\mathbf{G}(\mathcal{W})(\mathcal{N}).\mathsf{T}.\sqsubseteq & := \textit{see below} \\
\mathbf{G}(\mathcal{W})(\mathcal{N}).\mathsf{T}.\sqsubseteq_{\text{pub}} & := \textit{see below} \\
\mathbf{G}(\mathcal{W})(\mathcal{N}).\mathsf{L}(s_{\mathsf{ref}}, (s_1, \ldots, s_n))(R) & := \bigcup_{i \in \{1,\ldots,n\}} \mathcal{W}(\mathcal{N}_i).\mathsf{L}(s_{\mathsf{ref}}, s_i)(R) \\
\mathbf{G}(\mathcal{W})(\mathcal{N}).\mathsf{H}(s_{\mathsf{ref}}, (s_1, \ldots, s_n))(R) & := \otimes_{i \in \{1,\ldots,n\}} \mathcal{W}(\mathcal{N}_i).\mathsf{H}(s_{\mathsf{ref}}, s_i)(R) \\
\mathbf{G}(\mathcal{W})(\mathcal{N}).\mathsf{N} & := \mathcal{N}
\end{array}$$

Here $\{\mathcal{N}_i\}$ is a countably infinite splitting of $\mathcal{N}$ into coutably infinite sets, *i.e.,* $\mathcal{N} = \mathcal{N}_1 \uplus \mathcal{N}_2 \uplus \mathcal{N}_3 \uplus \ldots$ and each $\mathcal{N}_i$ is countably infinite. Many such splittings exists, and it does not matter which one we choose.

The transition relations $\mathbf{G}(\mathcal{W})(\mathcal{N}).\sqsubseteq$ and $\mathbf{G}(\mathcal{W})(\mathcal{N}).\sqsubseteq_{\text{pub}}$ are defined as the least preorders closed under the corresponding following rules.

$$\frac{s'_1 \sqsupseteq_{\text{pub}} s_1 \quad \ldots \quad s'_k \sqsupseteq_{\text{pub}} s_k}{(s'_1, \ldots, s'_k) \sqsupseteq_{\text{pub}} (s_1, \ldots, s_k)} \qquad \frac{}{(s_1, \ldots, s_k, s_{k+1}) \sqsupseteq_{\text{pub}} (s_1, \ldots, s_k)}$$

$$\frac{s'_1 \sqsupseteq s_1 \quad \ldots \quad s'_k \sqsupseteq s_k}{(s'_1, \ldots, s'_k) \sqsupseteq (s_1, \ldots, s_k)} \qquad \frac{s' \sqsupseteq_{\text{pub}} s}{s' \sqsupseteq s}$$

Hence in addition to pointwise transitions, they also allow extensions (moving from a state of arity k to one of arity $k + n$).

For the next lemmas, we define the following notation.

$$G[s_k]_{k \in \{n \setminus i\}} \quad := \quad \lambda s_{\mathsf{ref}}, s'_i.\ G(s_{\mathsf{ref}}, (s_1, \ldots, s_{i-1}, s'_i, s_{i+1}, \ldots, s_n))$$

Lemma 27.

$$\forall G \in \mathsf{GK}(\mathbf{G}(\mathcal{W})(\mathcal{N})\uparrow).\ \forall s_1 \ldots s_{i-1}, s_{i+1} \ldots s_n.\ G[s_k]_{k \in \{n \setminus i\}} \in \mathsf{GK}(\mathcal{W}(\mathcal{N}_i)\uparrow)$$

Proof. The fact that $G[s_k]_{k\in\{n\setminus i\}}$ is monotone w.r.t. $\sqsubseteq$ follows directly from the definition of $\sqsubseteq$ and monotonicity of G.

It remains to show

$$G[s_k]_{k\in\{n\setminus i\}}(s_{\mathsf{ref}}, s_i) \;\supseteq^{\mathcal{N}_i}_{\mathrm{ref}}\; \mathcal{W}(\mathcal{N}_i)\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s_i)(G[s_k]_{k\in\{n\setminus i\}}(s_{\mathsf{ref}}, s_i))$$

for any s_{ref}, s_i, which we do as follows.

$$\begin{array}{lll}
 & G[s_k]_{k\in\{n\setminus i\}}(s_{\mathsf{ref}}, s_i) & \\
= & G(s_{\mathsf{ref}}, s_1, \ldots, s_n) & \\
\supseteq^{\mathcal{N}}_{\mathrm{ref}} & \mathbf{G}(\mathcal{W})(\mathcal{N})\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s_1, \ldots, s_n)(G(s_{\mathsf{ref}}, s_1, \ldots, s_n)) & (G \in \mathsf{GK}(\mathbf{G}(\mathcal{W})(\mathcal{N})\uparrow)) \\
\supseteq^{\mathcal{N}_i}_{\mathrm{ref}} & \mathbf{G}(\mathcal{W})(\mathcal{N})\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s_1, \ldots, s_n)(G(s_{\mathsf{ref}}, s_1, \ldots, s_n)) & (\mathcal{N}_i \subseteq \mathcal{N}) \\
\supseteq^{\mathcal{N}_i}_{\mathrm{ref}} & \mathcal{W}(\mathcal{N}_i)\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s_i)(G(s_{\mathsf{ref}}, s_1, \ldots, s_n)) & \\
= & \mathcal{W}(\mathcal{N}_i)\uparrow.\mathsf{L}(s_{\mathsf{ref}}, s_i)(G[s_k]_{k\in\{n\setminus i\}}(s_{\mathsf{ref}}, s_i)) &
\end{array}$$

The last inequality relies on $\mathcal{W}(\mathcal{N}_i) \in \mathrm{LWorld}$ and the disjointness of $\mathcal{N}_1, \ldots, \mathcal{N}_n$. □

Lemma 28. If $W = \mathbf{G}(\mathcal{W})(\mathcal{N})\uparrow$ and $\forall \mathcal{N}'.\ \mathsf{stable}(\mathcal{W}(\mathcal{N}'))$ and $G \in \mathsf{GK}(W)$, then:

1. $\mathbf{E}_{\mathcal{W}(\mathcal{N}_i)\uparrow}(G[s_k]_{k\in\{n\setminus i\}})(s^0_{\mathsf{ref}}, s^0_i)(s_{\mathsf{ref}}, s_i) \subseteq$
   $\mathbf{E}_W(G)(s^0_{\mathsf{ref}}, s_1, \ldots, s_{i-1}, s^0_i, s_{i+1}, \ldots, s_n)(s_{\mathsf{ref}}, s_1, \ldots, s_n)$
2. $\mathbf{K}_{\mathcal{W}(\mathcal{N}_i)\uparrow}(G[s_k]_{k\in\{n\setminus i\}})(s^0_{\mathsf{ref}}, s^0_i)(s_{\mathsf{ref}}, s_i) \subseteq$
   $\mathbf{K}_W(G)(s^0_{\mathsf{ref}}, s_1, \ldots, s_{i-1}, s^0_i, s_{i+1}, \ldots, s_n)(s_{\mathsf{ref}}, s_1, \ldots, s_n)$

Proof. By coinduction. □

Lemma 29. Suppose $\forall \mathcal{N}.\ \mathsf{stable}(\mathcal{W}(\mathcal{N}))$.

1. $\forall \mathcal{N}.\ \mathsf{stable}(\mathbf{G}(\mathcal{W})(\mathcal{N}))$
2. If $\forall \mathcal{N}.\ \mathsf{consistent}(\mathcal{W}(\mathcal{N})\uparrow)$, then $\forall \mathcal{N}.\ \mathsf{consistent}(\mathbf{G}(\mathcal{W})(\mathcal{N})\uparrow)$.

Lemma 30. $\mathsf{inhabited}(\mathbf{G}(\mathcal{W})(\mathcal{N})\uparrow)$

Proof. It is easy to check that $(\emptyset, \emptyset) \in \mathbf{G}(\mathcal{W})(\mathcal{N})\uparrow.\mathsf{H}(\emptyset, ())(R)$ for any R. □

In the next section, we illustrate the use of $\mathbf{G}$ and these lemmas.

3.8.4 Twin Abstraction, Alternate Proof

Recall the example from Section 3.8.2. We now sketch how to prove it with the help of the world generator from the previous section.

Constructing the world.

$$
\begin{array}{ll}
w.\mathsf{T}.\mathsf{S} & := \{s \in \mathsf{Loc} \times \mathsf{Loc} \xrightarrow{\text{fin}} \mathcal{P}(\mathbb{N}_{>0}) \times \mathcal{P}(\mathbb{N}_{>0}) \mid \\
& \qquad \mathrm{dom}(s) \text{ partial bijection } \wedge \\
& \qquad \forall (l_1, l_2, S_1, S_2) \in s.\ S_1 \cap S_2 = \emptyset\} \\
w.\mathsf{T}.\sqsubseteq & := \{(s, s') \mid \forall (l_1, l_2, S_1, S_2) \in s.\ \exists S_1' \supseteq S_1, S_2' \supseteq S_2.\ (l_1, l_2, S_1', S_2') \in s'\} \\
w.\mathsf{T}.\sqsubseteq_{\text{pub}} & := w.\mathsf{T}.\sqsubseteq
\end{array}
$$

$$
\begin{array}{ll}
w.\mathsf{L}(s_{\mathsf{ref}}, s)(G) & := \{(\mathbf{n}^{\alpha}_{l_1}, n, n) \mid \exists l_2, S_1, S_2.\ (l_1, l_2, S_1, S_2) \in s \wedge n \in S_1\} \\
& \uplus\ \{(\mathbf{n}^{\beta}_{l_1}, n, n) \mid \exists l_2, S_1, S_2.\ (l_1, l_2, S_1, S_2) \in s \wedge n \in S_2\} \\
& \uplus\ \{((\mathsf{unit} \to \mathbf{n}^{\alpha}_{l_1}), {+\!\!+}l_1, {+\!\!+}l_2) \mid (l_1, l_2) \in \mathrm{dom}(s)\} \\
& \uplus\ \{((\mathsf{unit} \to \mathbf{n}^{\beta}_{l_1}), {+\!\!+}l_1, {+\!\!+}l_2) \mid (l_1, l_2) \in \mathrm{dom}(s)\} \\
& \uplus\ \{((\mathbf{n}^{\alpha}_{l_1} \times \mathbf{n}^{\beta}_{l_1} \to \mathsf{nat}), (\lambda p.\, p.1 = p.2), (\lambda p.\, 0)) \mid \exists l_2.\ (l_1, l_2) \in \mathrm{dom}(s)\}
\end{array}
$$

$$
\begin{array}{ll}
w.\mathsf{H}(s_{\mathsf{ref}}, s)(G) & := \{(h_1, h_2) \mid \\
& \qquad \mathrm{dom}(h_1) = \{l_1 \mid \exists l_2.\ (l_1, l_2) \in \mathrm{dom}(s)\} \wedge \\
& \qquad \mathrm{dom}(h_2) = \{l_2 \mid \exists l_1.\ (l_1, l_2) \in \mathrm{dom}(s)\} \wedge \\
& \qquad \forall (l_1, l_2, S_1, S_2) \in s.\ h_1(l_1) = h_2(l_2) = max(\{0\} \uplus S_1 \uplus S_2)\}
\end{array}
$$

$$
\begin{array}{ll}
w.\mathsf{N} & := \mathcal{N}
\end{array}
$$

First, we define $\mathcal{W} \in \mathrm{NLWorld}$ as follows:

$$
\begin{array}{l}
\mathcal{W}(\mathcal{N}').\mathsf{T}.\mathsf{S} := \{(l_1, l_2, S_1, S_2) \in \mathsf{Loc} \times \mathsf{Loc} \times \mathcal{P}(\mathbb{N}_{>0}) \times \mathcal{P}(\mathbb{N}_{>0}) \mid S_1 \cap S_2 = \emptyset\} \\
\mathcal{W}(\mathcal{N}').\mathsf{T}.\sqsubseteq := \{((l_1, l_2, S_1, S_2), (l_1', l_2', S_1', S_2') \mid l_1 = l_1' \wedge l_2 = l_2' \wedge S_1 \subseteq S_1' \wedge S_2 \subseteq S_2'\} \\
\mathcal{W}(\mathcal{N}').\mathsf{T}.\sqsubseteq_{\text{pub}} := \mathcal{W}(\mathcal{N}').\mathsf{T}.\sqsubseteq \\
\mathcal{W}(\mathcal{N}').\mathsf{L}(s_{\mathsf{ref}}, (l_1, l_2, S_1, S_2))(R) := \\
\quad \{(\mathcal{N}'(1), n, n) \mid n \in S_1\} \uplus \{(\mathcal{N}'(2), n, n) \mid n \in S_2\} \uplus \\
\quad \{((\mathsf{unit} \to \mathcal{N}'(1)), {+\!\!+}l_1, {+\!\!+}l_2)\} \uplus \{((\mathsf{unit} \to \mathcal{N}'(2)), {+\!\!+}l_1, {+\!\!+}l_2)\} \uplus \\
\quad \{((\mathcal{N}'(1) \times \mathcal{N}'(2) \to \mathsf{nat}), (\lambda p.\, p.1 = p.2), (\lambda p.\, 0))\} \\
\mathcal{W}(\mathcal{N}').\mathsf{H} := \lambda (l_1, l_2, S_1, S_2), R.\ \{([l_1 \mapsto n], [l_2 \mapsto n]) \mid n = max(\{0\} \uplus S_1 \uplus S_2)\} \\
\mathcal{W}(\mathcal{N}').\mathsf{N} := \{\mathcal{N}'(1), \mathcal{N}'(2)\}
\end{array}
$$

where $\mathcal{N}'(1)$ and $\mathcal{N}'(2)$ denote two distinct elements of $\mathcal{N}'$.

3.8.5 A Free Theorem

The last example demonstrates the treatment of universal types, and the fact that our method may be used to prove at least some simple so-called "free theorems" [90]. Suppose $\vdash p : \forall \alpha.\, \alpha$ and $|p| = v$. We want to prove that $\langle h; v\ [] \rangle \uparrow$ for any h, which implies that there are no closed values of type $\forall \alpha.\, \alpha$.

We start by applying reflexivity to obtain $\vdash v \sim v : \forall \alpha.\, \alpha$. This gives us w with $\vdash v \sim_{w\uparrow} v : \forall \alpha.\, \alpha$ and $w.\mathsf{N} \subseteq \mathsf{TyNam} \setminus \{\mathbf{n}\}$ for some arbitrary $\mathbf{n}$ of our choosing. We

now pick $G := [w\uparrow] \in \mathsf{GK}(w\uparrow)$ (recall Definition 4). From $\vdash v \sim_{w\uparrow} v : \forall\alpha.\,\alpha$ we then get

$$(v, v) \in \mathbf{E}_{w\uparrow}(G)(s_0)(s_0)(\forall\alpha.\,\alpha)$$

where s_0 is the state witnessing $\mathsf{inhabited}(w\uparrow)$. Notice that this can only hold due to the second disjunct in $\mathbf{E}$ because v is a value. Consequently, we get $s \sqsupseteq_{\text{pub}} s_0$ such that:

1. $(v, v) \in \overline{G(s)}(\forall\alpha.\,\alpha) = G(s)(\forall\alpha.\,\alpha)$
2. $(\emptyset, \emptyset) \in w\uparrow.\mathsf{H}(s)(G(s))$

By construction we have $G(s) = w\uparrow.\mathsf{L}(s)(G(s))$. Hence from (1) and $\mathsf{consistent}(w\uparrow)$, we can derive

$$(\mathit{step}(v\ []), \mathit{step}(v\ [])) \in \mathbf{E}_{w\uparrow}(G)(s)(s)(\tau)$$

for any τ. So, in particular, we have:

$$(\mathit{step}(v\ []), \mathit{step}(v\ [])) \in \mathbf{E}_{w\uparrow}(G)(s)(s)(\mathbf{n})$$

In fact, due to the construction of G, Lemma 12 tells us

$$(\mathit{step}(v\ []), \mathit{step}(v\ [])) \in \mathbf{E}^{\boldsymbol{\mathcal{R}}}_{w\uparrow}(G)(s)(s)(\mathbf{n})$$

for $\boldsymbol{\mathcal{R}} = \lambda s.\emptyset$. Together with (2), this allows only two cases for any heap h: either $\langle h; \mathit{step}(v\ [])\rangle$ diverges (then we are done), or it terminates and the resulting values are related by $\overline{G(s')}(\mathbf{n})$ for some $s' \sqsupseteq_{\text{pub}} s$. However, because $W_{\text{ref}}.\mathsf{N} = \emptyset$ and $w.\mathsf{N} \subseteq \mathsf{TyNam} \setminus \{\mathbf{n}\}$, we know $\overline{G(s')}(\mathbf{n}) = w\uparrow.\mathsf{L}(s')(G(s'))(\mathbf{n}) = \emptyset$, which rules out that second case.

3.9 Transitivity

In Section 3.5, we presented parametric bisimulations for λ^{μ}, a very simple fragment of $\mathsf{F}^{\mu!}$ lacking quantified types and state. In particular, we sketched the proof of transitivity in Section 3.5.4.

In this section, we generalize that proof of transitivity to account for the full PB model of the full $\mathsf{F}^{\mu!}$ language, which turns out (unsurprisingly) to be highly non-trivial. We first describe the high-level structure of our extended transitivity proof. The proof divides into two major parts, presented in Sections 3.9.2 and 3.9.3, respectively. It is highly intricate, requiring several tricky auxiliary constructions. In these sections, we highlight the key technical challenges, explain carefully the central constructions and the intuitions behind them, give detailed proofs of the main lemmas, and give proof sketches of other lemmas. The complete proof is of course fully machine-checked in Coq. To give a rough sense of the complexity, it required approximately 3200 lines of Coq, versus 1500 lines to formalize the language, 400 lines to formalize the model, and 2000 lines to prove soundness of the model w.r.t. contextual equivalence.

3.9.1 Structure of the Transitivity Proof

Given that $\Delta; \Gamma \vdash e_1 \sim e_2 : \tau$ and $\Delta; \Gamma \vdash e_2 \sim e_3 : \tau$, we must show $\Delta; \Gamma \vdash e_1 \sim e_3 : \tau$. Unfolding the definition of this goal, we are given a countably infinite set of type names $\mathcal{N}$ and must construct a stable local world w such that (a) $\Delta; \Gamma \vdash e_1 \sim_{w\uparrow} e_3 : \tau$ and (b) $w.\mathsf{N} \subseteq \mathcal{N}$ (*i.e.,* $w.\mathsf{L}$ defines no names outside of $\mathcal{N}$). To do so, we split $\mathcal{N}$ into three disjoint (and also countably infinite) pieces: $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_\exists$. The first two pieces are used to instantiate the assumptions regarding $e_1 \sim e_2$ and $e_2 \sim e_3$, respectively, thus yielding two stable local worlds w_1 and w_2 such that

$$\Delta; \Gamma \vdash e_1 \sim_{w_1\uparrow} e_2 : \tau \tag{3.1}$$

$$\text{and} \quad \Delta; \Gamma \vdash e_2 \sim_{w_2\uparrow} e_3 : \tau \tag{3.2}$$

as well as $w_1.\mathsf{N} \subseteq \mathcal{N}_1$ and $w_2.\mathsf{N} \subseteq \mathcal{N}_2$. Keeping $\mathcal{N}_1$ and $\mathcal{N}_2$ disjoint is a matter of basic hygiene: it ensures that w_1 and w_2, which we will be using in the construction of w, do not step on each other's toes by defining the same type name in incompatible ways. As for the names in $\mathcal{N}_\exists$, we reserve them for a special purpose to be explained later.

At this point, the proof divides into two separate parts. In the first part, we use w_1 and w_2 to directly construct a full world W such that $\Delta; \Gamma \vdash e_1 \sim_W e_3 : \tau$ (and $W.\mathsf{N} \subseteq \mathcal{N}$). While this construction and the proof of the required properties is quite subtle, they are essentially an extension of the transitivity proof for λ^μ that we sketched in detail in Section 3.5.4 and which we present here in much greater detail. The main novelty over that previous proof is that we now (in this first part) deal with quantified types; reference types do not cause much of a problem.

However, the second part of the proof has all to do with references. Specifically, the world W that we create in the first part does not have the required shape of a lifted local world $w\uparrow$. Thus, in the second part, we (i) develop a theory of *weak isomorphisms* between worlds and prove that they preserve term equivalence, and (ii) construct a stable local world w such that $w\uparrow$ is weakly isomorphic to W.

3.9.2 First Part: Constructing the Full World W

3.9.2.1 High-Level Explanation

As mentioned above, this first part of the proof is essentially agnostic as to whether the language/model supports mutable state. To ease the presentation, we therefore gloss over any state-related details at first; we will be more precise in Section 3.9.2.2. We also write W_i as shorthand for $w_i\uparrow$.

We want to construct W such that $\Delta; \Gamma \vdash e_1 \sim_W e_3 : \tau$. The proofs of consistency of W and $\mathbf{E}$-relatedness of closed instances of (e_1, e_3) turn out to be very similar, so let us focus on the latter here. We are given a global knowledge $G \in \mathsf{GK}(W)$ and related substitutions γ_1 and γ_3. These substitutions map each variable bound in Γ to related values $(v_1, v_3) \in \overline{G}(\tau')$. In order to make use of (3.1) and (3.2), we want to be able to: (i) "decompose" G into global knowledges $G_{(1)} \in \mathsf{GK}(W_1)$ and

$G_{(2)} \in \mathsf{GK}(W_2)$ that would be suitable for instantiating (3.1) and (3.2), and (ii) find a mediating substitution γ_2 s.t. for each $x \in \mathrm{dom}(\Gamma)$, it is the case that $\overline{G_{(1)}}$ relates $(\gamma_1 x, \gamma_2 x)$ and $\overline{G_{(2)}}$ relates $(\gamma_2 x, \gamma_3 x)$. Formally, we want:

$$\overline{G}(\tau) \subseteq \overline{G_{(1)}}(\tau_{(1)}) \circ \overline{G_{(2)}}(\tau_{(2)}) \tag{3.3}$$

where $\circ$ is ordinary relational composition. (Pretend *for now* that $\tau_{(i)} = \tau$. We will soon see why that's not good enough.)

If we have this, we can instantiate (3.1) and (3.2), thus obtaining $(\gamma_1 e_1, \gamma_2 e_2) \in \mathbf{E}(G_{(1)})(\tau_{(1)})$ and $(\gamma_2 e_2, \gamma_3 e_3) \in \mathbf{E}(G_{(2)})(\tau_{(2)})$. It thus remains for us to show $(\gamma_1 e_1, \gamma_3 e_3) \in \mathbf{E}(G)(\tau)$. In other words, we must (unsurprisingly) show some kind of transitivity property for $\mathbf{E}$:

$$\mathbf{E}_{W_1}(G_{(1)})(\tau_{(1)}) \circ \mathbf{E}_{W_2}(G_{(2)})(\tau_{(2)}) \subseteq \mathbf{E}_W(G)(\tau) \tag{3.4}$$

Proving this will certainly require us to prove the analogous transitivity property for values, which is the inverse of (3.3):

$$\overline{G}(\tau) \supseteq \overline{G_{(1)}}(\tau_{(1)}) \circ \overline{G_{(2)}}(\tau_{(2)}) \tag{3.5}$$

It is therefore only natural that we will define $W.\mathsf{L}$ to at least include the composition of $W_1.\mathsf{L}$ and $W_2.\mathsf{L}$:

$$W.\mathsf{L}(G)(\tau) \supseteq W_1.\mathsf{L}(G_{(1)})(\tau_{(1)}) \circ W_2.\mathsf{L}(G_{(2)})(\tau_{(2)}) \tag{3.6}$$

In order to see what else we want to put into W, and how to define $G_{(i)}$ and $\tau_{(i)}$, let us consider proving (3.3) and (3.5). For a flexible type τ' the conjunction of (3.3) and (3.5) is:

$$G(\tau') = G_{(1)}(\tau'_{(1)}) \circ G_{(2)}(\tau'_{(2)}).$$

One simple (but inadequate) choice for $G_{(i)}$ would be to define it as the minimal global knowledge in $\mathsf{GK}(W_i)$ (introduced in Definition 4). But there is no reason to believe that for any $(v_1, v_3) \in G(\tau')$, there magically happens to be a "mediating" value v_2 such that $W_1.\mathsf{L}$ relates (v_1, v_2) and $W_2.\mathsf{L}$ relates (v_2, v_3), as required by the $\subseteq$ part of the above equation. To work this magic, we will explicitly *add* such mediating values to $G_{(1)}$ and $G_{(2)}$! Concretely, we define $G_{(1)}$ as the smallest global knowledge in $\mathsf{GK}(W_1)$ that relates $(v_1, \mathbf{I}(\tau, v_1, v_3))$ at $\tau_{(1)}$ whenever G relates (v_1, v_3) at τ, and similarly $G_{(2)}$ as the smallest global knowledge in $\mathsf{GK}(W_2)$ that relates $(\mathbf{I}(\tau, v_1, v_3), v_3)$ at $\tau_{(2)}$ whenever G relates (v_1, v_3) at τ.

Now, what is this magic $\mathbf{I}$? For proving (3.3), it could be anything that maps to CVal. But for (3.5), it is crucial that each mediating value *uniquely* encodes the corresponding value pair (v_1, v_3). We therefore require $\mathbf{I}$ to be *injective*. Since all involved sets are countably infinite, such an encoding function exists and we do not care about the particular choice—except that we will choose its range to be of *rigid* type, specifically nat, for reasons we will explain shortly.

The proofs of (3.3) and (3.5) are by induction on the value closure (recall that it is constructed as a least fixed point). The reason why we must add more to $W.\mathsf{L}$ than just the composition in (3.6), and why $\tau_{(i)}$ cannot just be τ in general, has to do with abstract types. We illustrate the issue here for existential types, but the same problem arises for universals (although in a different place, namely in the proof of (3.4)).

Suppose $\tau_{(i)}$ were the identity and consider (3.5) at some type $\exists\alpha.\,\tau$: We would have to show that if $(\mathsf{pack}\ v_1, \mathsf{pack}\ v_2) \in \overline{G_{(1)}}(\exists\alpha.\,\tau)$ and $(\mathsf{pack}\ v_2, \mathsf{pack}\ v_3) \in \overline{G_{(2)}}(\exists\alpha.\,\tau)$, then $(\mathsf{pack}\ v_1, \mathsf{pack}\ v_3) \in \overline{G}(\exists\alpha.\,\tau)$. Unfolding the value closure, this means that given some τ_1', τ_2' with $(v_1, v_2) \in \overline{G_{(1)}}(\tau[\tau_1'/\alpha])$ and $(v_2, v_3) \in \overline{G_{(2)}}(\tau[\tau_2'/\alpha])$, we must come up with τ' such that $(v_1, v_3) \in \overline{G}(\tau[\tau'/\alpha])$. Now, if the two given representation types happen to be the same ($\tau_1' = \tau_2'$), then we could proceed by just picking $\tau' := \tau_1'$. But of course in general τ_1' and τ_2' will be different!

The intuition behind our solution is quite simple: we pick τ' to be a fresh *type name*, which we use to represent the semantic composition of τ_1' and τ_2'. More concretely, we use a type name $\mathbf{n}$ from $\mathcal{N}_\exists$ (which we reserved for exactly this purpose) to uniquely encode τ_1' and τ_2', then define $\mathbf{n}$'s meaning in W to be precisely $\overline{G_{(1)}}(\tau_1') \circ \overline{G_{(2)}}(\tau_2')$, and finally choose τ' to be $\mathbf{n}$. Since we don't know what τ_1' and τ_2' are, we simply have to encode all pairs of types this way. To pick the names, we use an injective function

$$\mathbf{A} \in \mathsf{CTy} \times \mathsf{CTy} \to \mathcal{N}_\exists$$

which, like $\mathbf{I}$, exists because all involved sets are countably infinite (and, as with $\mathbf{I}$, we do not care about its concrete definition). It should be clear by now what $\tau_{(i)}'$ does and that it is crucial for making the induction go through: it *decodes* τ' by traversing its structure and replacing each type name $\mathbf{n}$ that equals $\mathbf{A}(\tau_1, \tau_2)$—for some τ_1, τ_2—with τ_i.

That's the intuition; the reality is a bit more complex. It turns out that, in order to prove (3.3) and (3.5), the decoding must in fact be *bijective*, but the one sketched above is not injective. For instance, $\mathbf{A}(\mathsf{nat}, \mathsf{nat})$ and nat are obviously distinct types but both decode to nat (by either projection). Fortunately, there is an easy way to obtain the desired bijectivity: we only encode a type pair (τ_1, τ_2) directly as a name $\mathbf{A}(\tau_1, \tau_2)$ if τ_1 and τ_2 are "sufficiently different". If they share some structure, however, we keep the parts that are the same and only apply $\mathbf{A}$ to the parts that are different. To take a simple example: to encode $(\mathsf{nat}, \mathsf{unit})$, we would use the type name $\mathbf{A}(\mathsf{nat}, \mathsf{unit})$, but to encode $(\mathsf{nat} \to \mathsf{nat}, \mathsf{nat} \to \mathsf{unit})$, we would pick $\mathsf{nat} \to \mathbf{A}(\mathsf{nat}, \mathsf{unit})$ instead of $\mathbf{A}(\mathsf{nat} \to \mathsf{nat}, \mathsf{nat} \to \mathsf{unit})$.

Finally, property (3.4) is shown by coinduction. Recalling that $\mathbf{E}$ comprises three cases (DIV, EVAL, CALL), the key here is that the case of $\mathbf{E}$ by which the first and second terms—call them e_1' and e_2'—are related should match the case by which the second and third terms—e_2' and e_3'—are related. In other words, out of the 3×3 possible cases, 6 should never arise. As a representative example, consider the situation where e_1' and e_2' are related because they reduce to related values (case

EVAL), and e_2' and e_3' because they reduce to related calls with related continuations (case CALL). By Lemma 12 we can assume that these calls are to *external* functions related by $G_{(2)}$. But this means that the function called in e_2' must be of the form $\mathbf{I}(\tau, v_1, v_3)$, *i.e.,* not a function at all (recall that we required $\mathbf{I}$ to map to *rigid* values)! Hence, e_2' gets stuck, contradicting the assumption that it reduced to a value.

The remaining possibilities (where the cases of relatedness for (e_1', e_2') and (e_2', e_3') match) are handled as follows:

Case DIV. Then both e_1' and e_3' diverge, so we are done.

Case EVAL. Then e_1', e_2', and e_3' reduce to values v_1, v_2, and v_3, such that $\overline{G_{(1)}}$ relates (v_1, v_2) and $\overline{G_{(2)}}$ relates (v_2, v_3). Thus, by (3.5), $\overline{G}$ relates (v_1, v_3) and we are done.

Case CALL. We know that e_1' and e_2' reduce to related function calls in related continuations, and that the same applies to e_2' and e_3'. Using Lemma 12 as above, we know that all four function calls are actually stuck. Since, by determinacy, e_2' cannot get stuck in two different ways, we have a unique function call and a unique continuation in the middle. So, it suffices to show a corresponding transitivity property for $\mathbf{U}$ and for continuations. The one for continuations follows from (3.3) and the coinductive hypothesis. The one for $\mathbf{U}$ follows from (3.5) and injectivity of $\mathbf{I}$; when reasoning about type instantiations $(f_i\ [])$, we must also make use of our type encoding $\mathbf{A}$—dually to how we handle pack in proving (3.5).

3.9.2.2 The Gory Details

We first formalize the syntactic encoding of types. The previously motivated notion of two types being "sufficiently different" is defined as the negation of *similarity*.

Definition 10. Similarity, written $\approx$, is defined inductively via the following rules:

$$\frac{\mathbf{n} \notin \mathcal{N}_\exists}{\mathbf{n} \approx \mathbf{n}} \qquad \frac{}{\alpha \approx \alpha} \qquad \frac{}{\mathsf{unit} \approx \mathsf{unit}} \qquad \frac{}{\mathsf{nat} \approx \mathsf{nat}} \qquad \frac{\tau \sim \sigma}{\mathsf{ref}\ \tau \approx \mathsf{ref}\ \sigma}$$

$$\frac{\tau \sim \sigma}{\mu\alpha.\,\tau \approx \mu\alpha.\,\sigma} \qquad \frac{\tau \sim \sigma}{\forall\alpha.\,\tau \approx \forall\alpha.\,\sigma} \qquad \frac{\tau \sim \sigma}{\exists\alpha.\,\tau \approx \exists\alpha.\,\sigma}$$

$$\frac{\tau \sim \sigma \quad \tau' \sim \sigma'}{\tau \to \tau' \approx \sigma \to \sigma'} \qquad \frac{\tau \sim \sigma \quad \tau' \sim \sigma'}{\tau \times \tau' \approx \sigma \times \sigma'} \qquad \frac{\tau \sim \sigma \quad \tau' \sim \sigma'}{\tau + \tau' \approx \sigma + \sigma'}$$

$$\frac{\tau, \sigma \in \mathsf{CTy}}{\tau \sim \sigma} \qquad \frac{\tau \approx \sigma}{\tau \sim \sigma}$$

Two closed dissimilar types are encoded as a type name from $\mathcal{N}_\exists$ using a bijection

$$\mathbf{A} \in \{(\tau_1, \tau_2) \in \mathsf{CTy} \times \mathsf{CTy} \mid \tau_1 \not\approx \tau_2\} \to \mathcal{N}_\exists.$$

The encoding of two arbitrary closed types is a natural lifting of $\mathbf{A}$. Instead of defining it explicitly, we find it simpler to define the decoding and then state the existence of a corresponding encoding.

Definition 11. We recursively define *decoding projections* $(-)_{(i)} \in \mathsf{Ty} \to \mathsf{Ty}$ (for $i = 1, 2$) as follows. Note that τ is closed iff $\tau_{(i)}$ is closed.

$$\begin{aligned}
\mathbf{n}_{(i)} &:= \begin{cases} \tau_i & \text{if } \mathbf{n} = \mathbf{A}(\tau_1, \tau_2) \text{ for some } \tau_1, \tau_2 \\ \mathbf{n} & \text{otherwise, } \textit{i.e.,}\ \mathbf{n} \notin \mathcal{N}_\exists \end{cases} \\
\alpha_{(i)} &:= \alpha \\
\mathsf{unit}_{(i)} &:= \mathsf{unit} \\
\mathsf{nat}_{(i)} &:= \mathsf{nat} \\
(\mathsf{ref}\,\tau)_{(i)} &:= \mathsf{ref}\,\tau_{(i)} \\
(\mu\alpha.\,\tau)_{(i)} &:= \mu\alpha.\,\tau_{(i)} \\
(\forall\alpha.\,\tau)_{(i)} &:= \forall\alpha.\,\tau_{(i)} \\
(\exists\alpha.\,\tau)_{(i)} &:= \exists\alpha.\,\tau_{(i)} \\
(\tau \to \tau')_{(i)} &:= \tau_{(i)} \to \tau'_{(i)} \\
(\tau \times \tau')_{(i)} &:= \tau_{(i)} \times \tau'_{(i)} \\
(\tau + \tau')_{(i)} &:= \tau_{(i)} + \tau'_{(i)}
\end{aligned}$$

While we necessarily defined similarity and the decoding projections for arbitrary types (Ty), ultimately we are only interested in closed types (CTy). The encoding of two closed types is now implicit in the surjectivity part of the next lemma.

Lemma 31. $(\lambda\tau.\ \langle\tau_{(1)}, \tau_{(2)}\rangle) \in \mathsf{CTy} \to \mathsf{CTy} \times \mathsf{CTy}$ is bijective.

Proof. Injectivity (generalized to open types) can be easily shown by induction on types using two sub-lemmas:

(a) $\forall\tau.\ \tau_{(1)} \sim \tau_{(2)}$, which is shown by straightforward induction on τ.

(b) $\forall\tau.\ \tau_{(1)} \approx \tau_{(2)} \iff \tau \notin \mathcal{N}_\exists$, which is shown by case analysis on τ using (a).

Surjectivity is generalized as follows:

$$\forall\tau_1, \tau_2.\ \tau_1 \sim \tau_2 \implies \exists\tau.\ \tau_{(1)} = \tau_1 \wedge \tau_{(2)} = \tau_2$$

(Note that if τ_1 and τ_2 are closed, the premise holds trivially.) This property can be proven by induction on τ_1. In each case we ask if $\tau_1 \approx \tau_2$ holds. If it does, we use the inductive hypothesis; otherwise, the premise implies that τ_1, τ_2 are closed and thus we can pick τ to be $\mathbf{A}(\tau_1, \tau_2)$. □

Now, let us define the *decomposition* of value relations. This will eventually be used to decompose a global knowledge for the yet-to-be-defined W into one for W_1 and one for W_2.

Definition 12. Given $R \in \mathsf{VRelF}$, we define $R_{\{i\}} \in \mathsf{VRelF}$ and $R^*_{(i)} \in W_i.\mathsf{T}.\mathsf{S} \to \mathsf{VRelF}$ (for $i = 1, 2$) as follows.

$$\begin{aligned} R_{\{1\}}(\tau_1) &:= \{(v_1, \mathbf{I}(\tau, v_1, v_3)) \mid \exists \tau \in \mathsf{CTyF}^{\setminus\mathcal{N}}_{\setminus\mathrm{ref}}.\ \tau_1 = \tau_{(1)} \wedge (v_1, v_3) \in R(\tau)\} \\ R_{\{2\}}(\tau_2) &:= \{(\mathbf{I}(\tau, v_1, v_3), v_3) \mid \exists \tau \in \mathsf{CTyF}^{\setminus\mathcal{N}}_{\setminus\mathrm{ref}}.\ \tau_2 = \tau_{(2)} \wedge (v_1, v_3) \in R(\tau)\} \\ R^*_{(i)}(s) &:= [W_i.\mathsf{L}(s)]^*_{(R_{\{i\}})} \end{aligned}$$

Here, $\mathsf{CTyF}^{\setminus\mathcal{N}}_{\setminus\mathrm{ref}}$ stands for $\mathsf{CTyF} \setminus (\mathcal{N} \cup \{\mathsf{ref}\,\tau \in \mathsf{CTy}\})$ and $\mathbf{I}$ is an injective function in $\mathsf{CTy} \times \mathsf{CVal} \times \mathsf{CVal} \to \mathsf{CVal}$. By construction, $R^*_{(i)}$ is the least fixed point of $W_i.\mathsf{L}$ that contains $R_{\{i\}}$ (recall Definition 3).

With the help of this, we can now finally construct W.

$$\begin{aligned} W.\mathsf{T} &:= W_1.\mathsf{T} \times W_2.\mathsf{T} \\ W.\mathsf{L}(s_1, s_2)(R)(\tau) &:= \begin{cases} \overline{R^*_{(1)}(s_1)}(\tau_{(1)}) \circ \overline{R^*_{(2)}(s_2)}(\tau_{(2)}) & \text{if } \tau \in \mathcal{N}_\exists \\ W_1.\mathsf{L}(s_1)(R^*_{(1)}(s_1))(\tau_{(1)}) & \text{if } \tau \notin \mathcal{N}_\exists \\ \quad \circ\, W_2.\mathsf{L}(s_2)(R^*_{(2)}(s_2))(\tau_{(2)}) & \end{cases} \\ W.\mathsf{H}(s_1, s_2)(R) &:= W_1.\mathsf{H}(s_1)(R^*_{(1)}(s_1)) \circ W_2.\mathsf{H}(s_2)(R^*_{(2)}(s_2)) \\ W.\mathsf{N} &:= \mathcal{N} \end{aligned}$$

Its well-formedness is easy to check. Note that, although W only actually defines names from $\mathcal{N}_\exists$, we declare that it owns the larger set $\mathcal{N}$ in order to reduce the set of global knowledges that we have to worry about.

Next, some notation for decomposing a global knowledge.

Definition 13. Given $G \in \mathsf{GK}(W)$, we define $G^{s_2}_{(1)} \in W_1.\mathsf{S} \to \mathsf{VRelF}$ for $s_2 \in W_2.\mathsf{S}$, and $G^{s_1}_{(2)} \in W_2.\mathsf{S} \to \mathsf{VRelF}$ for $s_1 \in W_1.\mathsf{S}$.

$$G^{s_2}_{(1)}(s) := G(s, s_2)^*_{(1)}(s) \qquad G^{s_1}_{(2)}(s) := G(s_1, s)^*_{(2)}(s)$$

Note that if the global knowledge argument R in the definition of $W.\mathsf{L}$ and $W.\mathsf{H}$ is of the form $G(s_1, s_2)$ for $G \in \mathsf{GK}(W)$, then the corresponding argument passed to $W_1.\mathsf{L}$ and $W_1.\mathsf{H}$ is exactly $G^{s_2}_{(1)}(s_1)$ (and similarly for W_2). It remains to show that $G^{s_2}_{(1)}(s_1)$ and $G^{s_1}_{(2)}(s_2)$ are in fact valid global knowledges for W_1 and W_2, respectively:

Lemma 32. For any $G \in \mathsf{GK}(W)$:

- $\forall s_2.\ G^{s_2}_{(1)} \in \mathsf{GK}(W_1)$
- $\forall s_1.\ G^{s_1}_{(2)} \in \mathsf{GK}(W_2)$

Proof. We show the first part (the second is analogous). Monotonicity of $G^{s_2}_{(1)}$ is proven by induction (recall the construction) using the fact that G, $(-)_{\{1\}}$ and $W_1.\mathsf{L}$ are monotone. $G^{s_2}_{(1)}(s_1) \supseteq^{W_1.\mathsf{N}}_{\mathrm{ref}} W_1.\mathsf{L}(s_1)(G^{s_2}_{(1)}(s_1))$ follows from $G^{s_2}_{(1)}(s_1)(\tau) = W_1.\mathsf{L}(s_1)(G^{s_2}_{(1)}(s_1))(\tau) \cup G(s_1, s_2)_{\{1\}}(\tau)$. (Note that $G(s_1, s_2)_{\{1\}}(\tau) = \emptyset$ if τ is a reference type or a name in $W_1.\mathsf{N} = \mathcal{N}_1 \subseteq \mathcal{N}$.) □

We now come to the main lemma of this part, namely the conjunction of properties (3.3) and (3.5) from Section 3.9.2.1:

Lemma 33. $\forall G \in \mathsf{GK}(W).\ \forall \tau \in \mathsf{CTy}.\ \forall s_1, s_2.$

$$\overline{G^{s_2}_{(1)}(s_1)}(\tau_{(1)}) \circ \overline{G^{s_1}_{(2)}(s_2)}(\tau_{(2)}) = \overline{G(s_1, s_2)}(\tau)$$

Proof. The $\supseteq$ part is proven by induction on $\overline{G(s_1, s_2)}$ (recall that $\overline{(-)}$ is defined as a least fixed point). The other part is equivalent to $\overline{G^{s_2}_{(1)}(s_1)}(\tau) \subseteq X$, where X is

$$\big\{(v_1, v_2) \mid \forall \sigma, v_3.\ \tau = \sigma_{(1)} \wedge (v_2, v_3) \in \overline{G^{s_1}_{(2)}(s_2)}(\sigma_{(2)}) \implies (v_1, v_3) \in \overline{G(s_1,s_2)}(\sigma)\big\}.$$

This is proven by (generalized) induction on $\overline{G^{s_2}_{(1)}(s_1)}$.

The base cases of both inductive proofs follow from

$$\overline{G^{s_2}_{(1)}(s_1)}(\tau_{(1)}) \circ \overline{G^{s_1}_{(2)}(s_2)}(\tau_{(2)}) = G(s_1, s_2)(\tau)$$

for *flexible* τ (*i.e.,* $\tau \in \mathsf{CTyF}$), which we now show by case analysis on τ. If $\tau \in \mathcal{N}_\exists$, then

$$\begin{array}{rcll} G(s_1, s_2)(\tau) & = & W.\mathsf{L}(s_1, s_2)(G(s_1, s_2))(\tau) & (G \in \mathsf{GK}(W)) \\ & = & \overline{G^{s_2}_{(1)}(s_1)}(\tau_{(1)}) \circ \overline{G^{s_1}_{(2)}(s_2)}(\tau_{(2)}) & (\text{def. of } W.\mathsf{L}) \end{array}$$

and we are done. Otherwise ($\tau \in \mathsf{CTyF} \setminus \mathcal{N}_\exists$), we have:

$$\begin{array}{rl} & \overline{G^{s_2}_{(1)}(s_1)}(\tau_{(1)}) \circ \overline{G^{s_1}_{(2)}(s_2)}(\tau_{(2)}) \\ = & G^{s_2}_{(1)}(s_1)(\tau_{(1)}) \circ G^{s_1}_{(2)}(s_2)(\tau_{(2)}) \\ = & (W_1.\mathsf{L}(s_1)(G^{s_2}_{(1)}(s_1))(\tau_{(1)}) \cup G(s_1, s_2)_{\{1\}}(\tau_{(1)})) \circ \\ & (W_2.\mathsf{L}(s_2)(G^{s_1}_{(2)}(s_2))(\tau_{(2)}) \cup G(s_1, s_2)_{\{2\}}(\tau_{(2)})) \end{array}$$

Now, if τ is a reference type or a name from $\mathcal{N}_1 \uplus \mathcal{N}_2$, we finish by rewriting as follows:

$$\begin{array}{rll} & (W_1.\mathsf{L}(s_1)(G^{s_2}_{(1)}(s_1))(\tau_{(1)}) \cup G(s_1, s_2)_{\{1\}}(\tau_{(1)})) \circ & \\ & (W_2.\mathsf{L}(s_2)(G^{s_1}_{(2)}(s_2))(\tau_{(2)}) \cup G(s_1, s_2)_{\{2\}}(\tau_{(2)})) & \\ = & (W_1.\mathsf{L}(s_1)(G^{s_2}_{(1)}(s_1))(\tau_{(1)}) \cup \emptyset) \circ & \\ & (W_2.\mathsf{L}(s_2)(G^{s_1}_{(2)}(s_2))(\tau_{(2)}) \cup \emptyset) & (\text{def. of } (-)_{\{i\}}) \\ = & W.\mathsf{L}(s_1, s_2)(G(s_1, s_2))(\tau) & (\text{def. of } W.\mathsf{L}) \\ = & G(s_1, s_2)(\tau) & (G \in \mathsf{GK}(W)) \end{array}$$

Otherwise ($\tau \in \mathsf{CTyF}^{\setminus\mathcal{N}}_{\setminus\mathrm{ref}}$), we continue by distributing $\circ$ over $\cup$:

$$\begin{array}{rl} & (W_1.\mathsf{L}(s_1)(G^{s_2}_{(1)}(s_1))(\tau_{(1)}) \cup G(s_1, s_2)_{\{1\}}(\tau_{(1)})) \circ \\ & (W_2.\mathsf{L}(s_2)(G^{s_1}_{(2)}(s_2))(\tau_{(2)}) \cup G(s_1, s_2)_{\{2\}}(\tau_{(2)})) \\ = & (W_1.\mathsf{L}(s_1)(G^{s_2}_{(1)}(s_1))(\tau_{(1)}) \circ W_2.\mathsf{L}(s_2)(G^{s_1}_{(2)}(s_2))(\tau_{(2)})) \cup \\ & (G(s_1, s_2)_{\{1\}}(\tau_{(1)}) \circ G(s_1, s_2)_{\{2\}}(\tau_{(2)})) \cup \\ & (W_1.\mathsf{L}(s_1)(G^{s_2}_{(1)}(s_1))(\tau_{(1)}) \circ G(s_1, s_2)_{\{2\}}(\tau_{(2)})) \cup \\ & (G(s_1, s_2)_{\{1\}}(\tau_{(1)}) \circ W_2.\mathsf{L}(s_2)(G^{s_1}_{(2)}(s_2))(\tau_{(2)})) \end{array}$$

The first disjunct equals $W.\mathsf{L}(s_1, s_2)(G(s_1, s_2))(\tau)$ by construction of W; the second equals $G(s_1, s_2)(\tau)$ by construction of $(-)_{\{i\}}$, injectivity of $\mathbf{I}$ and the injectivity part of Lemma 31; and the third and the fourth are empty by construction of $(-)_{\{i\}}$: thanks to the FixVal and GenVal restriction (cf. Figure 3.12), no rigid value (such as the $\mathbf{I}$-encoded values in $G(s_1, s_2)_{\{i\}}$) can be related by a local knowledge. So, the big union above simply becomes $W.\mathsf{L}(s_1, s_2)(G(s_1, s_2))(\tau) \cup G(s_1, s_2)(\tau)$. Since $G \in \mathsf{GK}(W)$, the second disjunct already contains the first and we are done.

The inductive cases of both inductive proofs boil down to showing that for any $R_1, R_2, R \in \mathsf{VRelF}$ and $S_1, S_2 \in \mathsf{VRel}$ and $\tau \notin \mathsf{CTyF}$ the equation

$$F_{R_1}(S_1)(\tau_{(1)}) \circ F_{R_2}(S_2)(\tau_{(2)}) = F_R(S_1 \bullet S_2)(\tau)$$

holds, where $F_R \in \mathsf{VRel} \to \mathsf{VRel}$ denotes the monotone generating function of $\overline{R}$ (*i.e.,* the function of which $\overline{R}$ is the least fixed-point, and which can be easily inferred from Figure 3.11), and $(S_1 \bullet S_2)(\tau) := S_1(\tau_{(1)}) \circ S_2(\tau_{(2)})$. This is straightforward to show by case analysis on τ. The only really interesting case is for existential types, where (in one direction) we are given two witness types τ_1 and τ_2 and then apply Lemma 31 to find a witness type τ satisfying $\tau_{(1)} = \tau_1$ and $\tau_{(2)} = \tau_2$. □

Finally, we can prove transitive compositionality of $\mathbf{E}$ and then the original goal of this first part.

Lemma 34. $\forall G \in \mathsf{GK}(W).\ \forall \tau \in \mathsf{CTy}.\ \forall s_1, s_2.$

$$\mathbf{E}_{W_1}(G^{s_2}_{(1)})(s_1)(\tau_{(1)}) \circ \mathbf{E}_{W_2}(G^{s_1}_{(2)})(s_2)(\tau_{(2)}) \subseteq \mathbf{E}_W(G)(s_1, s_2)(\tau)$$

Proof. By coinduction, following the sketch in Section 3.9.2.1 (and choosing the middle frame heap, h^{F}_2, to be empty). □

Lemma 35. $\Delta; \Gamma \vdash e_1 \sim_W e_3 : \tau$

Proof. Inhabitation of W follows easily from that of W_1 and W_2 and the construction of $W.\mathsf{H}$. The proofs of consistency and relatedness of (e_1, e_3) by $\mathbf{E}_W$ are very similar and straightforward, using Lemmas 33 and 34. □

3.9.3 Second Part: Constructing the Corresponding Local World w

We now come to the second and final part of our transitivity proof. Conceptually it is quite simple, but the formal details are very subtle. Recall that we basically want to create a world that relates the same things as W from the previous section, but has the shape of a lifted world, *i.e.,* has the form $w\uparrow$ for some local world w. By definition, the state space of a lifted world is of the form $W_{\mathrm{ref}}.\mathsf{S} \times \ldots$, and thus cannot be the same as W's state space $(W_{\mathrm{ref}}.\mathsf{S} \times w_1.\mathsf{S}) \times (W_{\mathrm{ref}}.\mathsf{S} \times w_2.\mathsf{S})$.

Instead of trying to define w directly and then trying to prove that W and $w\uparrow$ give rise to the same equivalences, we take a more principled approach. We first develop a simple notion of *world isomorphism* in order to characterize what it means for two

worlds with different state spaces to "relate the same things". Subsequently, we find an isomorphism between W and a lifted local world. Although somewhat tailored to its application in the transivity proof, the isomorphism theory can be useful in other cases.

3.9.3.1 World Isomorphisms

Roughly, two (full) worlds W_a, W_b are isomorphic iff they declare the same type names, and each state of W_a corresponds to a state of W_b (and vice versa) such that the same values and heaps are related at corresponding states. Different kinds of isomorphism arise depending on what counts as a correspondence. For our purpose, a one-to-one correspondence is too strong. If, say, W_b contains an "inconsistent" state (*i.e.,* a state at which W_b's heap relation is empty), then we should not have to worry about finding a similarly irrelevant state in W_a. So, instead of a full one-to-one correspondence, we use a *partial* one, wherein a state s in one world is permitted to have no correspondent in the other if s is inconsistent. This plays a crucial role in our transitivity proof, as we will see in a moment.

Definition 14. For any $W_a, W_b \in \text{World}$, a function $\phi \in W_a.\mathsf{S} \to \mathcal{P}(W_b.\mathsf{S})$ is a *weak morphism* from W_a to W_b iff:

(1) $\forall s_a, s'_a.\ \forall s_b \in \phi(s_a).\ \forall s'_b \in \phi(s'_a).\ s_a \sqsubseteq s'_a \implies s_b \sqsubseteq s'_b$

(2) $\forall s_a, s'_a.\ \forall s_b \in \phi(s_a).\ \forall s'_b \in \phi(s'_a).\ s_a \sqsubseteq_{\text{pub}} s'_a \implies s_b \sqsubseteq_{\text{pub}} s'_b$

(3) $\forall s_a.\ \forall s_b \in \phi(s_a).\ W_a.\mathsf{L}(s_a) = W_b.\mathsf{L}(s_b)$

(4) $\forall s_a.\ \forall G \in \mathsf{GK}(W_a).\ W_a.\mathsf{H}(s_a)(G(s_a)) \subseteq \bigcup_{s_b \in \phi(s_a)} W_b.\mathsf{H}(s_b)(G(s_a))$

(5) $W_a.\mathsf{N} = W_b.\mathsf{N}$

It is easy to see that worlds and weak morphisms form a category.

Although we actually allow a state from one world to be mapped to a whole set of states from the other world, for the purpose of the transitivity proof this set will always be a singleton or empty.

Definition 15. For any $W_a, W_b \in \text{World}$, functions $\phi \in W_a.\mathsf{S} \to \mathcal{P}(W_b.\mathsf{S})$ and $\psi \in W_b.\mathsf{S} \to \mathcal{P}(W_a.\mathsf{S})$ form a *weak isomorphism*, written $\phi : W_a \cong W_b : \psi$, iff they are weak morphisms and satisfy the following:

- $\forall s_a.\ \forall s_b \in \phi(s_a).\ \forall s'_a \in \psi(s_b).\ s_a \sqsubseteq_{\text{pub}} s'_a$
- $\forall s_b.\ \forall s_a \in \psi(s_b).\ \forall s'_b \in \phi(s_a).\ s_b \sqsubseteq_{\text{pub}} s'_b$

Theorem 7 (Weak isomorphisms preserve equivalences)**.** If $\phi : W_a \cong W_b : \psi$, then for any $\Delta, \Gamma, e_1, e_2, \tau$:

$$\Delta; \Gamma \vdash e_1 \sim_{W_a} e_2 : \tau \iff \Delta; \Gamma \vdash e_1 \sim_{W_b} e_2 : \tau$$

Proof. The proof is both somewhat tricky and tedious, and omitted here. At the heart are the following two global knowledge constructions. Recall that for a monotone function $F \in \mathsf{VRelF} \to \mathsf{VRelF}$ and $R \in \mathsf{VRelF}$, we write $[F]^*_R$ for the least fixed point of the monotone function $F(-) \cup R$.

For $\phi : W_1 \to W_2$ and $G \in \mathsf{GK}(W_2)$:

$$\begin{array}{l} \overleftarrow{G_\phi} \in \mathsf{GK}(W_1) \\ \overleftarrow{G_\phi}(s_1) := [W_1.\mathsf{L}(s_1)]^*_{\bigcup\{G(s_2') \mid \exists s_1' \sqsubseteq s_1.\ s_2' \in \phi(s_1')\}} \end{array}$$

For $\phi : W_1 \to W_2$, $G \in \mathsf{GK}(W_1)$, and $s_1 \in W_1.\mathsf{S}$:

$$\begin{array}{l} \overrightarrow{G^{s_1}_\phi} \in \mathsf{GK}(W_2) \\ \overrightarrow{G^{s_1}_\phi}(s_2) = [W_2.\mathsf{L}(s_2)]^*_{\bigcup\{G(s_1') \mid s_1' \sqsubseteq s_1 \wedge \exists s_2'.\ s_2' \sqsubseteq s_2 \wedge s_2' \in \phi(s_1')\}} \end{array}$$

□

Before moving on, we show three simple examples of weak isomorphisms.

Theorem 8 (The *pruning* isomorphism)**.** Given $W \in$ World, we write $\mathit{prune}(W)$ for the world obtained by removing all invalid states from W's state space, where "invalid" means that no heaps are related at these states. Intuitively, these states simply do not matter, as they can never be meaningfully used in a proof. It is easy to show that W and $\mathit{prune}(W)$ are weakly isomorphic.

Formally, the transformation is defined as follows.

$$\begin{array}{ll} \mathit{prune}(W).\mathsf{T}.\mathsf{S} & := \{s \in W.\mathsf{T}.\mathsf{S} \mid \exists G \in \mathsf{GK}(W).\ \exists (h_1, h_2) \in W.\mathsf{H}(s)(G(s))\} \\ \mathit{prune}(W).\mathsf{T}.\sqsubseteq & := W.\mathsf{T}.\sqsubseteq \\ \mathit{prune}(W).\mathsf{T}.\sqsubseteq_{\text{pub}} & := W.\mathsf{T}.\sqsubseteq_{\text{pub}} \\ \mathit{prune}(W).\mathsf{L} & := W.\mathsf{L} \\ \mathit{prune}(W).\mathsf{H} & := W.\mathsf{H} \\ \mathit{prune}(W).\mathsf{N} & := W.\mathsf{N} \end{array}$$

We have $\phi : W \cong \mathit{prune}(W) : \psi$, where:

$$\begin{array}{l} \phi(s) := \{s \mid \exists G \in \mathsf{GK}(W).\ \exists (h_1, h_2) \in W.\mathsf{H}(s)(G(s))\} \\ \psi(s) := \{s\} \end{array}$$

(Note that $\phi(s)$ is either empty or a singleton set.)

Theorem 9 (The *flattening* isomorphism)**.** Given $W \in$ World, we write $\mathit{flatten}(W)$ for the world obtained by *flattening* W's state space, in the sense that at each new state at most one pair of heaps is related. This is done by pairing each state with

two heaps and defining the heap relation to be empty if these heaps are not related at the state. Formally, flattening is defined as follows.

$$\begin{array}{ll}
\mathit{flatten}(W).\mathsf{T}.\mathsf{S} & := W.\mathsf{T}.\mathsf{S} \times \mathsf{Heap} \times \mathit{Heap} \\
\mathit{flatten}(W).\mathsf{T}.\sqsubseteq & := \{((s', h_1', h_2'), (s, h_1, h_2)) \mid s' \sqsubseteq s\} \\
\mathit{flatten}(W).\mathsf{T}.\sqsubseteq_{\text{pub}} & := \{((s', h_1', h_2'), (s, h_1, h_2)) \mid s' \sqsubseteq_{\text{pub}} s\} \\
\mathit{flatten}(W).\mathsf{L}(s, h_1, h_2) & := W.\mathsf{L}(s) \\
\mathit{flatten}(W).\mathsf{H}(s, h_1, h_2)(R) & := \{(h_1, h_2)\} \cap W.\mathsf{H}(s)(R) \\
\mathit{flatten}(W).\mathsf{N} & := W.\mathsf{N}
\end{array}$$

We have $\phi : W \cong \mathit{flatten}(W) : \psi$, where:

$$\begin{array}{l}
\phi(s) := \{(s, h_1, h_2) \mid h_1, h_2 \in \mathsf{Heap}\} \\
\psi(s, h_1, h_2) := \{s\}
\end{array}$$

Theorem 10 (The *swapping* isomorphism)**.** Given $w \in \text{LWorld}$ with $w.\mathsf{S} = S_1 \times S_2$, we write $\mathit{swap}(w)$ for the local world obtained by *swapping* w's state space. Formally, swapping is defined as follows.

$$\begin{array}{ll}
\mathit{swap}(w).\mathsf{T}.\mathsf{S} & := S_2 \times S_1 \\
\mathit{swap}(w).\mathsf{T}.\sqsubseteq & := \{((s_2', s_1'), (s_2, s_1)) \mid (s_1', s_2') \sqsubseteq (s_1, s_2)\} \\
\mathit{swap}(w).\mathsf{T}.\sqsubseteq_{\text{pub}} & := \{((s_2', s_1'), (s_2, s_1)) \mid (s_1', s_2') \sqsubseteq_{\text{pub}} (s_1, s_2)\} \\
\mathit{swap}(w).\mathsf{L}(s_{\mathsf{ref}}, (s_2, s_1)) & := w.\mathsf{L}(s_{\mathsf{ref}}, (s_1, s_2)) \\
\mathit{swap}(w).\mathsf{H}(s_{\mathsf{ref}}, (s_2, s_1)) & := w.\mathsf{H}(s_{\mathsf{ref}}, (s_1, s_2)) \\
\mathit{swap}(w).\mathsf{N} & := w.\mathsf{N}
\end{array}$$

We have $\phi : w\!\uparrow \cong \mathit{swap}(w)\!\uparrow : \psi$, where:

$$\begin{array}{l}
\phi(s_{\mathsf{ref}}, (s_1, s_2)) := \{(s_{\mathsf{ref}}, (s_2, s_1))\} \\
\psi(s_{\mathsf{ref}}, (s_2, s_1)) := \{(s_{\mathsf{ref}}, (s_1, s_2))\}
\end{array}$$

Given local worlds $w_1, w_2 \in \text{LWorld}$, note that $\mathit{swap}(w_1 \otimes w_2) = w_2 \otimes w_1$. Hence we particularly have $\phi : (w_1 \otimes w_2)\!\uparrow \cong (w_2 \otimes w_1)\!\uparrow : \psi$. This can be used, for instance, to immediately obtain the two symmetric cases of Lemma 9.

Remark. Our notions of morphism and isomorphism are somewhat simple-minded and there ought to be more general versions. In particular, our morphisms by themselves seem not particularly useful. It is not the case, for instance, that a morphism between two worlds implies one direction of Theorem 7. We have played with the idea of morphisms as *lenses*[15] where a morphism from world W_1 to W_2 would mean that W_2 can be viewed as W_1, *e.g.,* by ignoring some of W_2's constraints. Two particulary important instances of such lenses would be the morphisms from $w_i\!\uparrow$ to $(w_1 \otimes w_2)\!\uparrow$, which should then automatically yield Lemma 9. This remains future work, however.

3.9.3.2 Defining w

Recall that lifting a local world means linking it with the shared world W_{ref}, which provides the meaning of reference types. Accordingly, the to-be-constructed local world w's knowledge must not relate anything at reference types, and, in order for $w{\uparrow}$ to be isomorphic to W, must correspond to $W.\mathsf{L}$ at all other types. This is easy to achieve by just choosing w's state space to be the same as W's and then defining $w.\mathsf{L}(s_{\mathsf{ref}})(s)$ to be $W.\mathsf{L}(s)$ for non-reference types. Regarding reference types, we have to satisfy (by the definition of lifting):

$$W_{\text{ref}}.\mathsf{L}(s^{\text{ref}})(R)(\mathsf{ref}\,\tau) = W.\mathsf{L}(s)(R)(\mathsf{ref}\,\tau)$$

This is problematic. Recall that $W.\mathsf{T} = W_1.\mathsf{T} \times W_2.\mathsf{T}$, so s really has the form $((s_1^{\text{ref}}, s_1^{\text{loc}}), (s_2^{\text{ref}}, s_2^{\text{loc}}))$ (we will later omit the inner parentheses for convenience), with $s_1^{\text{ref}}, s_2^{\text{ref}}$ being states of W_{ref}, and $s_1^{\text{loc}}, s_2^{\text{loc}}$ being states of w_1, w_2, respectively. Unfolding the definition of $W.\mathsf{L}$, the above equation is equivalent to

$$\begin{aligned} W_{\text{ref}}.\mathsf{L}(s^{\text{ref}})(R)(\mathsf{ref}\,\tau) = {} & W_{\text{ref}}.\mathsf{L}(s_1^{\text{ref}})(R^*_{(1)}(s_1^{\text{ref}}, s_1^{\text{loc}}))(\mathsf{ref}\,\tau_{(1)}) \circ \\ & W_{\text{ref}}.\mathsf{L}(s_2^{\text{ref}})(R^*_{(2)}(s_2^{\text{ref}}, s_2^{\text{loc}}))(\mathsf{ref}\,\tau_{(2)}), \end{aligned}$$

which in turn simplifies to $s^{\text{ref}} = s_1^{\text{ref}} \bullet s_2^{\text{ref}}$, where

$$s_1^{\text{ref}} \bullet s_2^{\text{ref}} := \{(\tau, \ell_1, \ell_3) \mid \exists \ell_2.\ (\tau_{(1)}, \ell_1, \ell_2) \in s_1^{\text{ref}} \wedge (\tau_{(2)}, \ell_2, \ell_3) \in s_2^{\text{ref}}\}.$$

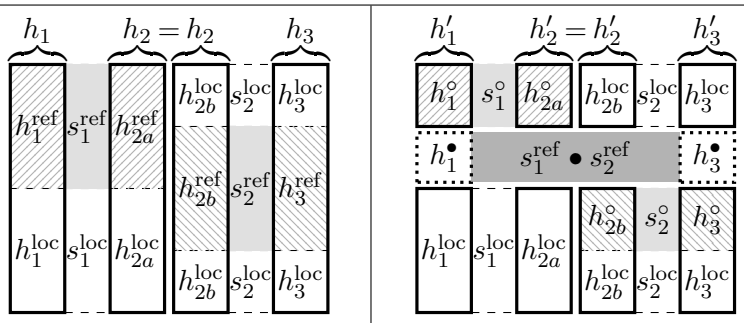
This clearly cannot be true in general as all three states may be arbitrary. Remember, however, that we do not have to worry about inconsistent states! So the solution is easy: in the states where the equation holds—*i.e.,* where s^{ref} and s are *coherent*—we are fine; we just need to make sure that in any other case $w{\uparrow}$'s heap relation is empty. And since w's heap relation may depend on the shared state s_{ref}, this can be easily achieved. Accordingly, the $w{\uparrow}$ state corresponding to s will be $(s_1^{\text{ref}} \bullet s_2^{\text{ref}}, s)$, and the W state corresponding to (s^{ref}, s) will be s—but only if s^{ref} happens to be $s_1^{\text{ref}} \bullet s_2^{\text{ref}}$.

What should $w.\mathsf{H}$ relate when $s^{\text{ref}} = s_1^{\text{ref}} \bullet s_2^{\text{ref}}$ *does* hold? For such states, we want the following equation to be true (ignoring the global knowledge parameter to avoid clutter):

$$w{\uparrow}.\mathsf{H}(s^{\text{ref}}, s) = W.\mathsf{H}(s) \tag{3.7}$$

Let us look at what we know about heaps (h_1, h_3) related by $W.\mathsf{H}(s)$. First, by construction of W, there is some h_2 mediating between $w_1{\uparrow}$ and $w_2{\uparrow}$. By definition of lifting, h_1 and h_2 can be split between $W_{\text{ref}}.\mathsf{H}(s_1^{\text{ref}})$ and $w_1.\mathsf{H}(s_1^{\text{ref}})(s_1^{\text{loc}})$, and similarly h_2 and h_3 can be split between $W_{\text{ref}}.\mathsf{H}(s_2^{\text{ref}})$ and $w_2.\mathsf{H}(s_2^{\text{ref}})(s_2^{\text{loc}})$. Of course, in general the two splits of h_2 may be arbitrarily different. This situation is depicted in the first diagram of Figure 3.18.

Note that $w{\uparrow}.\mathsf{H}(s^{\text{ref}}, s)$ in (3.7) unfolds to $W_{\text{ref}}.\mathsf{H}(s^{\text{ref}}) \otimes w.\mathsf{H}(s^{\text{ref}}, s)$. So basically all we have to do is define $w.\mathsf{H}(s^{\text{ref}}, s)$ to be the "septraction" [89] of $W_{\text{ref}}.\mathsf{H}(s^{\text{ref}})$ from $W.\mathsf{H}(s)$. The way we do this is essentially by describing, in the definition of $w.\mathsf{H}$, the situation from the figure but leaving out the pieces related by $W_{\text{ref}}.\mathsf{H}(s^{\text{ref}})$.



Heap structure enforced by W | Heap structure enforced by $w{\uparrow}$

Figure 3.18: Construction of a stable local world $w \in \text{LWorld}$ such that $w{\uparrow}$ is weakly isomorphic to W.

This is shown in the second diagram of Figure 3.18: $w.\mathsf{H}$ relates heaps h_1', h_3' iff $h_i' = h_i^{\text{loc}} \sqcup h_i^\circ$ ($\neq \bot$), where h_i° is the sub-heap of h_i^{ref} not covered by $s^{\text{ref}} = s_1^{\text{ref}} \bullet s_2^{\text{ref}}$. The missing pieces, $h_1^\bullet$ and $h_3^\bullet$, are then going to be related by $W_{\text{ref}}.\mathsf{H}(s^{\text{ref}})$ when w is lifted.

Formally, w is defined as follows.

$$
\begin{array}{lcl}
w.\mathsf{T} & := & W.\mathsf{T} \\
w.\mathsf{L}(s^{\text{ref}}, s)(R)(\tau) & := & \{(v_1, v_3) \in W.\mathsf{L}(s)(R)(\tau) \mid \nexists \tau'.\ \tau = \mathsf{ref}\ \tau'\} \\
w.\mathsf{H}(s^{\text{ref}}, s)(R) & := & \{(h_1', h_3') \mid \\
 & & \quad \exists s_1^{\text{ref}}, s_1^{\text{loc}}, s_2^{\text{ref}}, s_2^{\text{loc}}, h_1^\circ, h_1^{\text{loc}}, h_{2a}^\circ, h_{2a}^{\text{loc}}, h_{2b}^\circ, h_{2b}^{\text{loc}}, h_3^\circ, h_3^{\text{loc}}. \\
 & & \quad s = (s_1^{\text{ref}}, s_1^{\text{loc}}, s_2^{\text{ref}}, s_2^{\text{loc}}) \wedge s^{\text{ref}} = s_1^{\text{ref}} \bullet s_2^{\text{ref}} \wedge \\
 & & \quad h_1' = h_1^\circ \sqcup h_1^{\text{loc}} \wedge h_{2a}^\circ \sqcup h_{2a}^{\text{loc}} = h_{2b}^\circ \sqcup h_{2b}^{\text{loc}} \wedge h_3' = h_3^\circ \sqcup h_3^{\text{loc}} \wedge \\
 & & \quad \text{dom}(h_{2a}^{\text{loc}}) \cap \text{dom}_{[2]}(s_1^{\text{ref}}) = \text{dom}(h_{2b}^{\text{loc}}) \cap \text{dom}_{[1]}(s_2^{\text{ref}}) = \emptyset \wedge \\
 & & \quad (h_1^\circ, h_{2a}^\circ) \in W_{\text{ref}}.\mathsf{H}(s_1^\circ)(R_{(1)}^*(s_1^{\text{ref}}, s_1^{\text{loc}})) \wedge \\
 & & \quad (h_{2b}^\circ, h_3^\circ) \in W_{\text{ref}}.\mathsf{H}(s_2^\circ)(R_{(2)}^*(s_2^{\text{ref}}, s_2^{\text{loc}})) \wedge \\
 & & \quad (h_1^{\text{loc}}, h_{2a}^{\text{loc}}) \in w_1.\mathsf{H}(s_1^{\text{ref}})(s_1^{\text{loc}})(R_{(1)}^*(s_1^{\text{ref}}, s_1^{\text{loc}})) \wedge \\
 & & \quad (h_{2b}^{\text{loc}}, h_3^{\text{loc}}) \in w_2.\mathsf{H}(s_2^{\text{ref}})(s_2^{\text{loc}})(R_{(2)}^*(s_2^{\text{ref}}, s_2^{\text{loc}}))\} \\
w.\mathsf{N} & := & W.\mathsf{N}
\end{array}
$$

As explained, its heap relation $w.\mathsf{H}$ is empty whenever s^{ref} is not compatible with s_1^{ref} and s_2^{ref}. The sub-heap h_1° (and similarly h_3°) is characterized by saying that it is related by W_{ref} to a sub-heap h_{2a}° of h_{2a}^{ref} at the state obtained by essentially subtracting those parts from s_1^{ref} that are involved in $s_1^{\text{ref}} \bullet s_2^{\text{ref}}$.

3.9.3.3 Showing w's Stability

The well-formedness of w (*i.e.*, $w \in \text{LWorld}$) is fairly easy to check, but proving $\mathsf{stable}(w)$ is tricky (because $w.\mathsf{H}$'s dependency on s_{ref} is tricky)[6]. Recall that stability

[6]In fact, this proof of transitivity was the whole motivation for allowing a local world's heap relation to depend on the shared state in the first place.

is crucial for soundness, as it ensures that a local world's dependency on the shared state is compatible with any changes to that state.

Preliminary to that proof, we define the conversion of a global knowledge for $w\!\uparrow$ to one for W.

Lemma 36. If $s_1^{\mathrm{ref}}, s_2^{\mathrm{ref}} \in W_{\mathrm{ref}}.\mathsf{S}$, then $s_1^{\mathrm{ref}} \bullet s_2^{\mathrm{ref}} \in W_{\mathrm{ref}}.\mathsf{S}$.

Proof. This relies on the injectivity part of Lemma 31. □

Definition 16. For $G \in \mathsf{GK}(w\!\uparrow)$, we define $\overleftarrow{G} \in W.\mathsf{S} \to \mathsf{VRelF}$ as follows:

$$\overleftarrow{G}(s_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, s_2^{\mathrm{ref}}, s_2^{\mathrm{loc}}) := G(s_1^{\mathrm{ref}} \bullet s_2^{\mathrm{ref}}, (s_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, s_2^{\mathrm{ref}}, s_2^{\mathrm{loc}}))$$

Lemma 37. $\forall G \in \mathsf{GK}(w\!\uparrow).\ \overleftarrow{G} \in \mathsf{GK}(W)$

Lemma 38. $\mathsf{stable}(w)$

Proof. Suppose that $G \in \mathsf{GK}(w\!\uparrow)$, $s = (s_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, s_2^{\mathrm{ref}}, s_2^{\mathrm{loc}})$, $\hat{s}^{\mathrm{ref}} \sqsupseteq s^{\mathrm{ref}}$, $h_1' \sqcup \hat{h}_1^\bullet \neq \bot$, $h_3' \sqcup \hat{h}_3^\bullet \neq \bot$,

$$(h_1', h_3') \in w.\mathsf{H}(s^{\mathrm{ref}}, s)(G(s^{\mathrm{ref}}, s)) \tag{3.8}$$

$$\text{and} \quad (\hat{h}_1^\bullet, \hat{h}_3^\bullet) \in W_{\mathrm{ref}}.\mathsf{H}(\hat{s}^{\mathrm{ref}})(G(\hat{s}^{\mathrm{ref}}, s)). \tag{3.9}$$

Our goal is to find $\hat{s} = (\hat{s}_1^{\mathrm{ref}}, \hat{s}_1^{\mathrm{loc}}, \hat{s}_2^{\mathrm{ref}}, \hat{s}_2^{\mathrm{loc}}) \sqsupseteq s$ such that

$$(h_1', h_3') \in w.\mathsf{H}(\hat{s}^{\mathrm{ref}}, \hat{s})(G(\hat{s}^{\mathrm{ref}}, \hat{s})).$$

The main idea is to use $\mathsf{stable}(w_i)$ to obtain $\hat{s}_i^{\mathrm{loc}}$ (for $i = 1, 2$).

In order to do so, we must first construct suitable states and heaps needed for instantiating those stabilities. From (3.8) we know $s^{\mathrm{ref}} = s_1^{\mathrm{ref}} \bullet s_2^{\mathrm{ref}}$ and that h_1', h_3' are structured as depicted in the diagram of Figure 3.19. From $\hat{s}^{\mathrm{ref}} \sqsupseteq s^{\mathrm{ref}}$ we know $\hat{s}^{\mathrm{ref}} = s^{\mathrm{ref}} \uplus s^+$ for some s^+. Thus by (3.9), $\hat{h}_1^\bullet, \hat{h}_3^\bullet$ can be split as $\hat{h}_i^\bullet = h_i^\bullet \sqcup h_i^+$ such that

$$(h_1^\bullet, h_3^\bullet) \in W_{\mathrm{ref}}.\mathsf{H}(s^{\mathrm{ref}})(G(\hat{s}^{\mathrm{ref}}, s)) \tag{3.10}$$

$$\text{and} \quad (h_1^+, h_3^+) \in W_{\mathrm{ref}}.\mathsf{H}(s^+)(G(\hat{s}^{\mathrm{ref}}, s)), \tag{3.11}$$

as depicted in the left part of the diagram in Figure 3.19.

We will now "horizontally" decompose s^+ and s^{ref}, as shown in the right part of Figure 3.19. Since $s_i^{\mathrm{ref}} \sqsupseteq s_i^\circ$, there is $s_i^\bullet$ such that $s_i^{\mathrm{ref}} = s_i^\circ \uplus s_i^\bullet$; consequently we have $\mathrm{dom}_{[2]}(s_1^\bullet) = \mathrm{dom}_{[1]}(s_2^\bullet)$ and $s_1^\bullet \bullet s_2^\bullet = s_1^{\mathrm{ref}} \bullet s_2^{\mathrm{ref}} = s^{\mathrm{ref}}$. To decompose s^+, we choose a set of fresh locations L (of appropriate size) for the middle, *i.e.,* we define s_i^+ such that $s_1^+ \bullet s_2^+ = s^+$ and $\mathrm{dom}_{[2]}(s_1^+) = \mathrm{dom}_{[1]}(s_2^+) = L$. We can then define $\hat{s}_i^{\mathrm{ref}}$ (used to instantiate $\mathsf{stable}(w_i)$) as $\hat{s}_i^{\mathrm{ref}} := s_i^\circ \uplus s_i^\bullet \uplus s_i^+$, so we have $\hat{s}_1^{\mathrm{ref}} \bullet \hat{s}_2^{\mathrm{ref}} = s^{\mathrm{ref}} \uplus s^+ = \hat{s}^{\mathrm{ref}}$.

The mediating heaps h_2^+ (with domain L) and $h_2^\bullet$ are constructed as follows. Since $\hat{s}_i^{\mathrm{ref}} \sqsupseteq s_i^{\mathrm{ref}}$, we know

$$\overleftarrow{G}(\hat{s}_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, \hat{s}_2^{\mathrm{ref}}, s_2^{\mathrm{loc}}) = G(\hat{s}^{\mathrm{ref}}, (\hat{s}_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, \hat{s}_2^{\mathrm{ref}}, s_2^{\mathrm{loc}})) \sqsupseteq G(\hat{s}^{\mathrm{ref}}, s) \tag{3.12}$$

by monotonicity of G. From (3.10), (3.11), (3.12), monotonicity of $W_{\text{ref}}.\mathsf{H}$, and Lemmas 37 and 33, we can then find mediating values to construct $h_2^\bullet$ and h_2^+ satisfying (for $\star \in \{\bullet, +\}$)

$$(h_1^\star, h_2^\star) \in W_{\text{ref}}.\mathsf{H}(s_1^\star)(\overleftarrow{G}_{(1)}^{(\hat{s}_2^{\text{ref}}, s_2^{\text{loc}})}(\hat{s}_1^{\text{ref}}, s_1^{\text{loc}})) \tag{3.13}$$

$$\text{and} \quad (h_2^\star, h_3^\star) \in W_{\text{ref}}.\mathsf{H}(s_2^\star)(\overleftarrow{G}_{(2)}^{(\hat{s}_1^{\text{ref}}, s_1^{\text{loc}})}(\hat{s}_2^{\text{ref}}, s_2^{\text{loc}})). \tag{3.14}$$

We will now prepare to instantiate $\mathsf{stable}(w_i)$, starting with $\mathsf{stable}(w_1)$. First, observe that $\overleftarrow{G}_{(1)}^{(\hat{s}_2^{\text{ref}}, s_2^{\text{loc}})} \in \mathsf{GK}(w_1\uparrow)$, thanks to Lemmas 37 and 32. Next, by monotonicity of G, we have that (for $\star \in \{\hat{s}_1^{\text{ref}}, s_1^{\text{ref}}\}$)

$$\overleftarrow{G}_{(1)}^{(\hat{s}_2^{\text{ref}}, s_2^{\text{loc}})}(\star, s_1^{\text{loc}}) = \overleftarrow{G}(\star, s_1^{\text{loc}}, \hat{s}_2^{\text{ref}}, s_2^{\text{loc}})_{(1)}^* \supseteq G(s^{\text{ref}}, s)_{(1)}^*$$

and thus (3.8), along with monotonicity of $W_{\text{ref}}.\mathsf{H}$ and $w_1.\mathsf{H}$ and the definition of $w.\mathsf{H}$ in Figure 3.18, gives us:

$$(h_1^\circ, h_{2a}^\circ) \in W_{\text{ref}}.\mathsf{H}(s_1^\circ)(\overleftarrow{G}_{(1)}^{(\hat{s}_2^{\text{ref}}, s_2^{\text{loc}})}(\hat{s}_1^{\text{ref}}, s_1^{\text{loc}})) \tag{3.15}$$

$$\text{and} \quad (h_1^{\text{loc}}, h_{2a}^{\text{loc}}) \in w_1.\mathsf{H}(s_1^{\text{ref}}, s_1^{\text{loc}})(\overleftarrow{G}_{(1)}^{(\hat{s}_2^{\text{ref}}, s_2^{\text{loc}})}(s_1^{\text{ref}}, s_1^{\text{loc}})). \tag{3.16}$$

Thus, by (3.13), (3.15), (3.16), and $\hat{s}_1^{\text{ref}} \supseteq s_1^{\text{ref}}$, we can instantiate $\mathsf{stable}(w_1)$, yielding $\hat{s}_1^{\text{loc}} \sqsupseteq s_1^{\text{loc}}$ such that

$$(h_1^{\text{loc}}, h_{2a}^{\text{loc}}) \in w_1.\mathsf{H}(\hat{s}_1^{\text{ref}}, \hat{s}_1^{\text{loc}})(\overleftarrow{G}_{(1)}^{(\hat{s}_2^{\text{ref}}, s_2^{\text{loc}})}(\hat{s}_1^{\text{ref}}, \hat{s}_1^{\text{loc}})). \tag{3.17}$$

In a similar manner, $\mathsf{stable}(w_2)$ yields $\hat{s}_2^{\text{loc}} \sqsupseteq s_2^{\text{loc}}$ such that

$$(h_{2b}^{\text{loc}}, h_3^{\text{loc}}) \in w_2.\mathsf{H}(\hat{s}_2^{\text{ref}}, \hat{s}_2^{\text{loc}})(\overleftarrow{G}_{(2)}^{(\hat{s}_1^{\text{ref}}, s_1^{\text{loc}})}(\hat{s}_2^{\text{ref}}, \hat{s}_2^{\text{loc}})). \tag{3.18}$$

Finally, let $\hat{s} := (\hat{s}_1^{\text{ref}}, \hat{s}_1^{\text{loc}}, \hat{s}_2^{\text{ref}}, \hat{s}_2^{\text{loc}})$. By monotonicity of G, $W_{\text{ref}}.\mathsf{H}$, and $w_i.\mathsf{H}$, and by definition of $w.\mathsf{H}$, we get

$$(h_1', h_3') \in w.\mathsf{H}(\hat{s}^{\text{ref}}, \hat{s})(\overleftarrow{G}(\hat{s})) = w.\mathsf{H}(\hat{s}^{\text{ref}}, \hat{s})(G(\hat{s}^{\text{ref}}, \hat{s}))$$

from (3.8), (3.17), and (3.18), as desired. □

3.9.3.4 Proving W and $w\uparrow$ Isomorphic

We now show that W and $w\uparrow$ are weakly isomorphic. Then we can put all the pieces together, arrive at our goal and thereby finishing the proof of transitivity.

Lemma 39. $\exists \phi, \psi.\ \phi : W \cong w\uparrow : \psi$

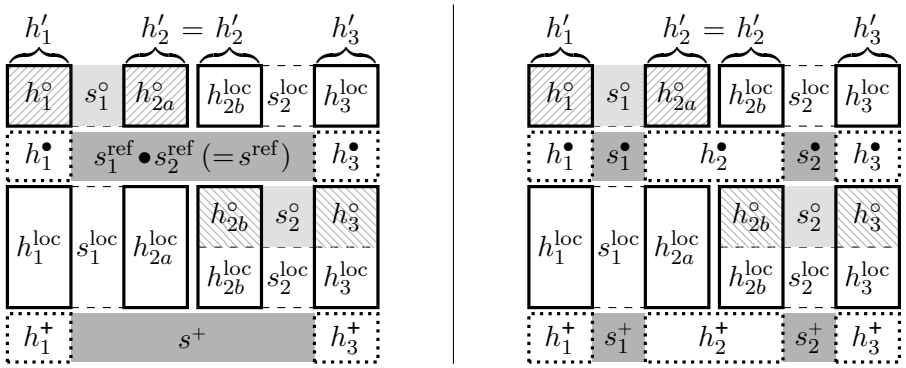


Figure 3.19: "Horizontal" decomposition of $\hat{s}^{\mathrm{ref}}$ in proof of $\mathsf{stable}(w)$.

Proof. We define $\phi \in W.\mathsf{S} \to \mathcal{P}(w{\uparrow}.\mathsf{S})$ and $\psi \in w{\uparrow}.\mathsf{S} \to \mathcal{P}(W.\mathsf{S})$ as discussed earlier.

$$\begin{aligned}\phi(s_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, s_2^{\mathrm{ref}}, s_2^{\mathrm{loc}}) &:= \{(s_1^{\mathrm{ref}} \bullet s_2^{\mathrm{ref}}, (s_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, s_2^{\mathrm{ref}}, s_2^{\mathrm{loc}}))\}\\ \psi(s^{\mathrm{ref}}, (s_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, s_2^{\mathrm{ref}}, s_2^{\mathrm{loc}})) &:= \{(s_1^{\mathrm{ref}}, s_1^{\mathrm{loc}}, s_2^{\mathrm{ref}}, s_2^{\mathrm{loc}}) \mid s^{\mathrm{ref}} = s_1^{\mathrm{ref}} \bullet s_2^{\mathrm{ref}}\}\end{aligned}$$

Showing that ϕ and ψ form a weak isomorphism is mostly straightforward, but for condition (4) of each morphism quite tedious. The key idea behind the proofs of these is the same as that behind Lemma 38: splitting the given heaps as depicted in the diagrams of Figure 3.18. In particular, the proof of morphism condition (4) for ψ is very similar to that of Lemma 38 in the way it uses the construction $\overleftarrow{G}$ and Lemma 37. □

Theorem 11 (Transitivity)**.** $\Delta; \Gamma \vdash e_1 \sim e_3 : \tau$

Proof. We have $\Delta; \Gamma \vdash e_1 \sim_{w\uparrow} e_3 : \tau$ by Lemmas 35 and 39 and Theorem 7. The result then follows from Lemma 38. □

3.10 Stuttering Parametric Bisimulations

PBs, as described earlier, suffer from a limitation: they fail to validate the *eta law* for function values,

$$f{:}(\sigma \to \tau) \vdash f \sim (\lambda x.\, f\ x) : \sigma \to \tau,$$

as well as more complex equivalences (*e.g.,* the *syntactic minimal invariance* property [12]) which depend on it.

In this section, we sketch *stuttering parametric bisimulations* (SPBs), a variant of PBs that overcomes this limitation. The full details are beyong the scope of this thesis and can be found in a technical report [35].

The key behind SPBs is the use of a *logical reduction semantics* that permits finite but unbounded stuttering steps in between actual "physical" steps. Such stuttering steps effectively enable proofs of equivalence of programs to engage in a game of

"hot potato", whereby the burden of proof may be tossed back and forth between different parts of the programs, until *eventually* some part makes a physical step of computation. This technique is inspired by the key idea in *well-founded* [58] and *stuttering* [17] bisimulations, adapted in the cited report for the first time to reasoning about open, higher-order programs.

Interestingly, this mechanism also helps overcome a second limitation, which appears at first glance to be completely unrelated: PBs bake in the assumption that the language in question has a "uniform" reduction semantics, *i.e.,* that the reduction relation is parametric in the evaluation context. This assumption is of course not valid in the presence of control flow constructs like callcc, so one may naturally wonder how to adapt PBs to a language supporting such features.

3.10.1 The Problem with Eta

Let us examine the inherent problem with eta in the PB model, for simplicity in the setting of λ^{μ} (Section 3.5). The eta law for an arbitrary function type $\tau' \to \tau$ corresponds to the following equivalence:

$$f{:}(\tau' \to \tau) \vdash f \sim (\lambda x.\, f\ x) : \tau' \to \tau$$

This equivalence does *not* hold. Here we prove as much for the case of $\tau' = \tau = \mathsf{nat}$.

Proof. By definition the equivalence holds iff there exists a consistent local knowledge L such that for any global knowledge $G \in \mathrm{GK}(L)$ and any related values $(v_1, v_2) \in G(\mathsf{nat} \to \mathsf{nat})$ we have $(v_1, \lambda x.\, v_2\ x) \in \mathbf{E}(G)(\mathsf{nat} \to \mathsf{nat})$. Being values of function type, such v_1 and $\lambda x.\, v_2\ x$ are related by $\mathbf{E}(G)$ iff they are related by G. Thus, in order to disprove the eta law, it suffices to construct a "bad" global knowledge $G^{\lightning} \in \mathsf{GK}(L)$ that relates v_1 and v_2 but does not relate v_1 and $\lambda x.\, v_2\ x$.

This is an easy task. Let $G^{\lightning}$ be the least global knowledge that subsumes L and also relates $\lambda y.\, 0$ and $\lambda y.\, 1$ at $\mathsf{nat} \to \mathsf{nat}$. Then we have:

$$G^{\lightning}(\mathsf{nat} \to \mathsf{nat}) = L(G^{\lightning})(\mathsf{nat} \to \mathsf{nat}) \cup \{(\lambda y.\, 0, \lambda y.\, 1)\}$$

It remains to prove that $G^{\lightning}$ does not relate $\lambda y.\, 0$ and $\lambda x.\, (\lambda y.\, 1)\ x$. Arguing by contradiction, assume it does. Then from $\mathsf{consistent}(L)$ and, say, $(42, 42) \in \overline{G^{\lightning}}(\mathsf{nat})$ we know $(\mathit{step}((\lambda y.\, 0)\ 42), \mathit{step}((\lambda x.\, (\lambda y.\, 1)\ x)\ 42)) \in \mathbf{E}(G^{\lightning})(\mathsf{nat})$, *i.e.,* $(0, (\lambda y.\, 1)\ 42) \in \mathbf{E}(G^{\lightning})(\mathsf{nat})$. Since 0 is a value, this can only mean that $(\lambda y.\, 1)\ 42$ reduces to a related value, *i.e.,* $(0, 1) \in \overline{G^{\lightning}}(\mathsf{nat})$, which by definition of $\overline{G^{\lightning}}$ is false. □

To understand better what is going on here, let us now try to *prove* the eta law and see what goes wrong. As is evident from the reasoning at the beginning of the above disproof, we would have to construct a consistent local knowledge L such that any G subsuming it relates v_1 and $\lambda x.\, v_2\ x$ whenever it relates v_1 and v_2. Since the only leverage we have over G is what we put in L, the only way to force G to relate certain things is to choose L so that it relates them. Luckily, our definition of L

may depend on G as a parameter, so in order to obtain the aforementioned closure property, we can attempt to define the local knowledge such that:

$$L_\eta(R)(\tau' \to \tau) = \{(v_1, \lambda x.\, v_2\ x) \mid (v_1, v_2) \in R(\tau' \to \tau)\}$$

Intuitively, this local knowledge corresponds exactly to what we want to claim: if our context provides us with values v_1 and v_2 that are equivalent at $\tau' \to \tau$, then we are prepared to claim that v_1 and $\lambda x.\, v_2\ x$ are equivalent at the very same type. Unfortunately, this L_η may be inconsistent! Specifically, suppose we are given $G \in \mathrm{GK}(L_\eta)$ and related arguments $(v_1', v_2') \in \overline{G}(\tau')$. For any $(v_1, v_2) \in G(\tau' \to \tau)$, we must show $(\mathit{step}(v_1\ v_1'), \mathit{step}((\lambda x.\, v_2\ x)\ v_2')) \in \mathbf{E}(G)(\tau)$. The trouble is that, while we know that $\mathit{step}((\lambda x.\, v_2\ x)\ v_2') = v_2\ v_2'$, we have no idea what $\mathit{step}(v_1\ v_1')$ is.[7]

The problem here essentially is that the global knowledge G is under no obligation to be sound w.r.t. contextual equivalence. As a result, if we define a local knowledge like L_η that "re-exports" function values (like v_1) obtained from G, there is no way to know whether applications of such values reduce to well-behaved terms.

3.10.2 Guardedness Revisited

Recall that the requirement of "taking a step" in the definition of PB's consistency is crucial in ensuring soundness because it guarantees that coinductive reasoning is suitably *guarded*. As the failed proof attempt above shows, however, the guardedness condition appears to be a little too strict. Note that if we were not forced to take that step, then we could easily finish the proof of $\mathsf{consistent}(L_\eta)$ by appealing to $(v_1\ v_1', v_2\ v_2') \in \mathbf{E}(G)(\tau)$, which follows from $(v_1, v_2) \in G(\tau' \to \tau)$ and $(v_1', v_2') \in \overline{G}(\tau')$ (both given), and $(\bullet, \bullet) \in \mathbf{K}(G)(\tau)(\tau)$, using the third disjunct of $\mathbf{E}$.

Of course, we cannot simply drop the stepping requirement, since this would immediately result in unsoundness—we must have some way of ensuring "productivity" of proofs. What we want to do then, in order to obtain a model that validates the eta law, is to find a slightly weaker guardedness condition (leading to a weaker notion of consistency) that enables the sketched proof of the eta law to go through but is nevertheless strong enough to guarantee soundness of the model.

We achieve this by generalizing the physical notion of "taking a step" to a *logical* one.

3.10.3 Logical Reduction and the Stutter Budget

The idea is very simple. We introduce into the model what we call a *stutter budget* (or just *budget*, for short): two natural numbers, one for each program, that specify how many times one may "stutter"—*i.e.,* avoid taking a reduction step (thus seemingly making no progress)—before eventually taking a step. More precisely, a local knowledge will contain items of the form (n_1, v_1, n_2, v_2) rather than just (v_1, v_2). When proving consistency for such an item, *i.e.,* when showing that the applications

[7] In fact, it may not even exist: for instance, if v_1 is the integer 5, then $\mathit{step}(v_1\ v_1')$ is undefined.

of (v_1, v_2) to related arguments (v_1', v_2') are related, one then has to make a choice for each application before continuing the reasoning: either one reduces it by one physical step (as before), *or* one leaves it untouched but decreases the corresponding budget instead (n_1 for the application of v_1, and n_2 for the application of v_2). When one chooses the latter option, one temporarily shirks one's responsibility to make physical progress, passing the proof burden—or "hot potato" as we called it earlier—to the subgoal of showing that the applications are in the **E** relation. Using the third disjunct (the CALL case), the **E** relation may then do the same thing and pass the hot potato back to the local knowledge. The important thing is that, each time around this seemingly circular proof path, the respective stutter budgets (n_1 and/or n_2) must be decreased, so we know the hot potato game cannot go on forever.

The way we formulate this is that one *is* actually required to perform a reduction step on both sides, as before, but only a *logical* one. This logical reduction relation, operating on an expression and its budget, is defined as follows.

Definition 17 (Logical Reduction)**.**

$$\frac{e \hookrightarrow e'}{n, e \hookrightarrow n', e'} \qquad \frac{n' < n}{n, e \hookrightarrow n', e}$$

That is, a logical step is either a physical step, in which case one may pick an arbitrary budget n' to continue with, or a stutter step, in which case the budget must be decreased.

To get an intuition for why the proposed change to the model is sound, first observe that, since the stutter budget is finite, progress (in the form of a physical step) will eventually be made. Second, note that logical (non-)termination coincides with physical (non-)termination. Thus, logical reduction gives us more flexibility in terms of local reasoning about v_1 and v_2, and this added flexibility is perfectly sound in that it will not enable us to equate terminating and divergent programs.

3.10.4 Eta Revisited

Using the refined model, we can now indeed prove the eta law along the lines of the earlier attempt by picking the following local knowledge:

$$\begin{aligned} L_\eta(R)(\tau) := \{(n_1', v_1, n_2', \lambda x.\, v_2\ x) \mid\ & \exists \tau', \tau''.\ \tau = \tau' \to \tau'' \wedge \\ & \exists n_1, n_2.\ n_1 < n_1' \wedge (n_1, v_1, n_2, v_2) \in R(\tau)\} \end{aligned}$$

Like before, we relate v_1 and $\lambda x.\, v_2\ x$ whenever v_1 and v_2 are related by R. But note the budgets: we choose n_1', the budget at which we "export" v_1 to be *larger* than n_1, the budget at which we import it. (Intuitively, $n_1 := n_1 + 1$ would suffice, but local knowledges in SPBs must be upwards-closed in their budgets.) The budget for $\lambda x.\, v_2\ x$, on the other hand, can be arbitrary. It doesn't matter because an application of that function can always take a physical reduction step (beta reduction).

3.10.5 First-Class Continuations

It turns out that SPBs also enable reasoning about first-class continuations, *e.g.,* in the presence of callcc (recall that λ^{μ} and $\mathsf{F}^{\mu!}$ do not feature control effects).

Basically, in such a setting, one wants the local and global knowledge to also relate continuations (evaluation contexts). Continuations can be thought of as functions, so it makes sense to treat them similarly—in particular to have an analogous concistency condition for the continuations related by the local knowledge. The issue then is that one frequently needs to show consistency of continuations that are constructed on top of continuations related by the global knowledge. A representative example is the following:

$$L(R)(\mathsf{nat}) := \{(K_1[\mathsf{inl}\ \bullet], K_2[\mathsf{inl}\ \bullet]) \mid (K_1, K_2) \in R(\mathsf{nat} + \tau)\}$$

Here, we import continuations K_1 and K_2, and export two continuations that are defined in terms of them. To show consistency, we are given related values v_1 and v_2 and must establish that $(\mathit{step}(K_1[\mathsf{inl}\ v_1]), \mathit{step}(K_2[\mathsf{inl}\ v_2]))$ are related by $\mathbf{E}$. Since the injection of a value into a sum type does not require any computation (*i.e.,* $\mathsf{inl}\ v_i$ does not take a step because it is itself a value), we are stuck.

This problem is obviously very similar to the one that we faced with eta. It should then no longer be a surprise that SPBs solve it as well. Using the stutter budget, we can define a local knowledge like the one above as follows:

$$\begin{aligned} L(R)(\mathsf{nat}) := \{(n_1', K_1[\mathsf{inl}\ \bullet], n_2', K_2[\mathsf{inl}\ \bullet]) \mid \\ \exists n_1 < n_1', n_2 < n_2'.\ (n_1, K_1, n_2, K_2) \in R(\mathsf{nat} + \tau)\} \end{aligned}$$

The consistency of this knowledge is easy to show using logical reduction for both programs.

3.10.6 Comparison to Step-Indexing

There is a superficial sense in which SPB proofs may seem similar to step-indexed KLR (SKLR) proofs, namely that they both involve a certain element of "step-counting". The key difference is *which* steps they are counting: in SKLR proofs one counts physical reduction steps (because the model is built by induction on such steps), whereas in SPB proofs one only counts the stuttering steps (to ensure the absence of infinite stuttering). In essence, SPBs are "non-step"-indexed relations!

Consequently, with SKLRs this step-counting is essential—one cannot ignore physical steps—whereas with SPBs it is often unnecessary. Indeed, when using SPBs to reason about the many examples that PBs already handle, one may usually ignore the extra flexibility provided by SPBs' logical reduction, avoiding step-counting by restricting one's proofs to only rely on physical reduction.

Finally, unlike step-indexing, SPB's stutter budgets cause no problems with transitivity. The technical report [35] contains a detailed proof of transitivity for SPBs. It largely follows the one for PBs but contains some interesting twists.

3.11 Greatest Local Knowledge

Writing down a suitable local knowledge at the beginning of a proof can be quite tedious for complex equivalences (even more so in SPBs). While not really an issue in paper proofs, this quickly becomes very tiresome in formal proofs such as these in our Coq formalization. Fortunately, we can employ *parameterized coinduction* [34, 56] to avoid this issue completely and instead write proofs in an incremental style, where we basically start with a knowledge containing just the functions in question, and extend it (in an implicit fashion) as the proof evolves. Indeed, this is how we prove a big part of soundness in Coq.

In order to get there, we need to express the property of a local knowledge being consistent as that local knowledge being a postfixed point of some monotone function. Then the greatest fixed point of that function is automatically the greatest consistent local knowledge, and so we can use the incremental reasoning principle from parameterized coinduction to do proofs about it. For simplicity, here we do this in the setting of λ^{μ} (Section 3.5). The generalization to $\mathsf{F}^{\mu!}$ is straightforward.

Definition 18. We define the wanted function $\mathfrak{f} \in \mathrm{LK} \overset{\mathrm{mon}}{\to} \mathrm{LK}$.

$$\begin{aligned}\mathfrak{f}(L)(R)(\tau) := \{(v_1, v_2) \mid\ & \forall G \in \mathsf{GK}(L).\ G \supseteq R \implies \\ & \forall \tau'.\ \forall (e_1, e_2) \in \mathbf{U}(\{\tau \mapsto (v_1, v_2)\}, G)(\tau'). \\ & (\mathit{step}(e_1), \mathit{step}(e_2)) \in \mathbf{E}(G)(\tau')\}\end{aligned}$$

Here, $\{\tau \mapsto (v_1, v_2)\}$ is short for the typed value relation $\lambda\tau'.\ \{(v_1, v_2) \mid \tau' = \tau\}$ (containing a single element).

It is easy to verify that $\mathfrak{f}$ is well-defined, *i.e.,* that it always returns a valid local knowledge and is itself monotone. The reason for quantifying over larger G is to ensure monotonicity of the returned local knowledge.

Lemma 40. $L \subseteq \mathfrak{f}(L) \iff \mathrm{consistent}(L)$

Definition 19 (Parameterized greatest consistent local knowledge)**.**

$$\begin{aligned}&\mathfrak{L} \in \mathrm{LK} \to \mathrm{LK} \\ &\mathfrak{L}(L) := \nu X.\ \mathfrak{f}(X \cup L)\end{aligned}$$

Corollary 1 (Greatest consistent local knowledge)**.**

1. $\mathrm{consistent}(\mathfrak{L}(\emptyset))$
2. $\forall L \in \mathrm{LK}.\ \mathrm{consistent}(L) \implies L \subseteq \mathfrak{L}(\emptyset)$

Theorem 12. $\Gamma \vdash e_1 \sim e_2 : \tau \iff \Gamma \vdash e_1 \sim_{\mathfrak{L}(\emptyset)} e_2 : \tau$

Proof. The "if" direction is trivial due to the first part of Corollary 1.

Regarding the "only if" direction: From $\Gamma \vdash e_1 \sim e_2 : \tau$ we know $\Gamma \vdash e_1 \sim_L e_2 : \tau$ with $\mathrm{consistent}(L)$. Hence $L \subseteq \mathfrak{L}(\emptyset)$ by the second part of Corollary 1. It is easy to check that $\Gamma \vdash e_1 \sim_L e_2 : \tau$ implies $\Gamma \vdash e_1 \sim_{L'} e_2 : \tau$ for any $L' \supseteq L$, so in particular for $\mathfrak{L}(\emptyset)$. □

Thanks to the construction of $\mathfrak{L}$ as a greatest parameterized fixed point, we have the expected incremental reasoning principle.

Lemma 41. $L \subseteq \mathfrak{L}(L') \iff L \subseteq \mathfrak{L}(L \cup L')$

Finally, a few words about the generalisation to the $\mathsf{F}^{\mu!}$ setting, where we have not only local knowledges but whole worlds. There we define an operation that *completes* a world by overwriting its local knowledge component with the greatest consistent local knowledge. Consequently, one constructs a world as usual (defines the STS and so on), but can leave its local knowledge empty or include only the initial functions. Then one applies the completion operator to the world and works with the resulting world instead. There is one point to note, though: since there is no consistency condition on values related at type names $\mathbf{n}$, completion does not touch these parts of the local knowledge. Hence the initial local knowledge must already contain any such needed values.

3.12 Comparison To Logical Relations

Let us conclude this chapter with high-level comparison of parametric (bi-)simulations and logical relations.

As we have seen, parametric bisimulations emerge from prior work on KLRs [4, 22] in an attempt to overcome its limitations concerning transitivity. In fact, both PBs and KLRs support the same high-level reasoning principles for higher-order imperative programs; they just do so in technically different ways.

Let us consider the two key principles concerning higher-order functions:

Principle 1 (Showing Behavioral Equivalence): To show that two higher-order functions v_a and v_b **behave** equivalently, it suffices to show that their applications **behave** equivalently when the arguments f_a and f_b passed in are **assumed** equivalent.

Principle 2 (Using Assumed Equivalence): If f_a and f_b are **assumed** equivalent, then $f_a\,\langle\rangle$ and $f_b\,\langle\rangle$ **behave** equivalently.

So what does it mean to "behave equivalently"? This is really the big question, for which KLRs and PBs give different answers. Rather than try to answer it directly, we will instead describe two informal proof principles concerning behavioral equivalence that, at the level of abstraction we are working at here, are supported by both proof methods, and we will finish the proof sketch by just appealing to these proof principles. After that, we will explain how the different proof methods implement these principles.

Note that these proof principles make a distinction between when two functions *behave* equivalently and when they are *assumed* equivalent. We explain the difference between these notions below. Note also that we restricted Principle 2 here to functions with unit argument; this is merely to simplify our informal discussion.

Logical relations. Both KLRs and PBs allow one to turn the above proof sketch into a proper proof. The key difference between them is how they formalize behavioral vs. assumed equivalence.

KLRs formalize this by defining a relation, which says—once and for all—what it means for two expressions to be indistinguishable at a certain type. One then uses this *same* "logical" relation as the definition of *both* behavioral and assumed equivalence. For expressions, the logical relation says that they are equivalent if they either both run forever or they both evaluate to equivalent values. For function values, which both the v's and the f's in our example are, the logical relation says that they are equivalent if they map logically-related arguments to logically-related results. Principles 1 and 2 both fall out of this definition as immediate consequences.

The main difficulty with logical relations is that, by conflating behavioral and assumed equivalence, they introduce an inherent circularity in the construction of the logical relation. In particular, the definition of equivalence of function values refers recursively to itself in a negative position (when quantifying over equivalent arguments). Traditionally, for simpler languages (*e.g.,* without recursive types or higher-order state), this circularity is handled by defining the logical relation by induction on the type structure. For richer languages, such as our source language $\mathcal{S}$, induction on types is no longer sufficient, but step-indexing can be used instead to stratify the construction by the number of steps of computation in the programs being related [4]. This is the approach taken by Hur and Dreyer [33] in their earlier work on compositional compiler correctness. However, it is not known how to prove *transitivity* of logical relations for step-indexed models (at least in a way that is capable of scaling to handle inter-language reasoning, which we need for compiler verification).

Parametric bisimulations. This problem with transitivity was one of the key motivations for *parametric bisimulations (PBs).* Unlike logical relations, PBs treat behavioral and assumed equivalence as distinct concepts. In particular, rather than trying to *define* assumed equivalence, PBs take assumed equivalence as a *parameter* of the model (hence the name "parametric bisimulations"). That is, a PB proof that two expressions are behaviorally equivalent is parameterized by an arbitrary *unknown* relation G (the global knowledge) representing assumed equivalence, and G could relate *any* functions f_a and f_b!

To make use of this unknown G parameter, PBs update the definition of behavioral equivalence accordingly. For function values, one can show them behaviorally equivalent precisely as suggested by Principle 1, *i.e.,* if they map G-related (assumed equivalent) arguments to behaviorally equivalent results. For expressions, behavioral equivalence extends the definition from logical relations with a new possibility, namely that the expressions may call functions f_a and f_b related by G. This amounts to baking Principle 2 directly into the definition of behavioral equivalence. The reason this is necessary—*i.e.,* the reason Principle 2 does not just fall out of the definition otherwise—is that G is a *parameter* of behavioral equivalence. Knowing that f_a and

f_b are assumed equivalent according to G tells us absolutely nothing about them! Consequently, Principle 2 must be explicitly added to the definition of behavioral equivalence as an extra case (the CALL disjunct).

One can understand PBs as defining a "local" notion of behavioral equivalence: two expressions are behaviorally equivalent if they behave the same in their local computations, *ignoring* what happens during calls to (G-related) external functions passed in from the environment. Intuitively, this is perfectly sound: it just means each module in a program is responsible for its own local computations, not the local computations of other modules. Moreover, as we have observed already, it is largely a technical detail: PBs can support the same high-level protocol-based reasoning as KLRs do.

The major benefit of PBs over KLRs is that they avoid the problems with the circularity of KLRs which necessitated step-indexing. In particular, note that by taking Principle 1 as the *definition* of behavioral equivalence for function values, we avoid the negative self-reference that plagues logical relations: the arguments f_a and f_b are simply drawn from the global knowledge G. This avoidance of step-indexing was essential in making it possible to establish that PBs do in fact support transitivity, but the proof of transitivity was far from easy.

Chapter 4

Parametric Inter-Language Simulations

Contents

4.1 Overview

PILS generalize PBs to the inter-language setting of compiler verification. Concretely we are interested in compiling from an ML-like source language $\mathcal{S}$ to a target machine language $\mathcal{T}$. In this section, we give an overview over the results of this chapter and the subsequent chapter.

The main component of our development is a relation between target modules and source modules: $\Gamma \vdash M_{\mathcal{T}} \precsim_{\mathcal{TS}} M_{\mathcal{S}} : \Gamma'$, to be defined later, intuitively states that target module $M_{\mathcal{T}}$ refines source module $M_{\mathcal{S}}$ and that they import the functions listed in Γ and export those in Γ'. The first key result, Theorem 13, applies to whole programs, *i.e.,* well-typed modules that import nothing and export at least a main function (F_{main}) of appropriate type. It states that our relation implies the standard behavioral refinement:

Theorem 13 (Adequacy for whole programs)**.**

$$\frac{\epsilon \vdash M_{\mathcal{T}} \precsim_{\mathcal{TS}} M_{\mathcal{S}} : \Gamma \qquad (F_{\text{main}} : \mathsf{unit} \to \tau) \in \Gamma}{\mathsf{Behav}(M_{\mathcal{T}}) \subseteq \mathsf{Behav}(M_{\mathcal{S}})}$$

$\mathsf{Behav}(-)$ denotes the set of I/O and termination behaviors that a program can have. The theorem implies for instance that, if $M_{\mathcal{S}}$ always successfully terminates, then so does $M_{\mathcal{T}}$ and moreover they produce the same outputs.

If we have a compiler that respects our relation $\precsim_{\mathcal{TS}}$, then Theorem 13 gives us the same result as traditional whole-program compiler verification would. However, our relation also satisfies the following crucial property (omitting a few well-formedness side conditions–see Section 4.7 for the full statement and its proof).

Theorem 14 (Preservation under linking, a.k.a. modularity)**.**

$$\frac{\Gamma \vdash M^1_{\mathcal{T}} \precsim_{\mathcal{TS}} M^1_{\mathcal{S}} : \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash M^2_{\mathcal{T}} \precsim_{\mathcal{TS}} M^2_{\mathcal{S}} : \Gamma_2}{\Gamma \vdash \mathsf{link}(M^1_{\mathcal{T}}, M^2_{\mathcal{T}}) \precsim_{\mathcal{TS}} \mathsf{link}(M^1_{\mathcal{S}}, M^2_{\mathcal{S}}) : \Gamma_1, \Gamma_2}$$

The theorem says that if we link two target modules, each of which is related to a source module, then the resulting target module is related to the linking of those source modules. Notice that, for the linking to make sense, the types of the first module's exported functions (in Γ_1) need to match the second module's assumptions.

Of course, if a program consists of more than two modules, this theorem can be iterated as necessary and once linking results in a whole program, we can apply Theorem 13.

Observe that these properties don't mention any particular compiler but are stated in terms of arbitrary related modules. The missing link is a theorem saying that the desired compilers adhere to our relation. We prove this for *Pilsner* and *Zwickel*, our compilers from $\mathcal{S}$ to $\mathcal{T}$. Their correctness theorems (slightly edited here for presentation) apply to any well-typed source module:

Theorem 22 (Correctness of Pilsner)**.**

$$\frac{\Gamma \vdash M_{\mathcal{S}} : \Gamma'}{\Gamma \vdash \text{Pilsner}(M_{\mathcal{S}}) \precsim_{\mathcal{TS}} M_{\mathcal{S}} : \Gamma'}$$

Theorem 23 (Correctness of Zwickel)**.**

$$\frac{\Gamma \vdash M_{\mathcal{S}} : \Gamma'}{\Gamma \vdash \text{Zwickel}(M_{\mathcal{S}}) \precsim_{\mathcal{TS}} M_{\mathcal{S}} : \Gamma'}$$

While Zwickel carries out a straightforward direct translation from $\mathcal{S}$ to $\mathcal{T}$, Pilsner is more sophisticated: as shown in Figure 1.1, it compiles via an intermediate language $\mathcal{I}$ and performs several optimizations. We will discuss Pilsner and Zwickel in detail in Chapter 5.

These results mean that we can preserve correctness not only by linking, say, Pilsner-produced code with other Pilsner-produced code, but also by linking it with code produced by Zwickel.

Moreover, we would like to stress two important points. PILS were designed with flexibility in mind and make only few assumptions about the translation of source programs, namely details of the calling convention and in-memory representation of values (see the subsequent sections). Consequently:

1. Nothing stops us from proving a theorem analogous to the previous two for *yet another* compiler from $\mathcal{S}$ to $\mathcal{T}$, perhaps even using several different intermediate languages.

2. Nothing stops us from proving the relatedness of a source and target module *by hand*, *e.g.,* when the target module is not the direct result of a compiler run but was manually optimized (see Section 5.13 for an extreme example of this, where the target module overwrites its own code at run time).

Hence, we can also preserve correctness when linking with code that was produced by other compilers or even hand-translated. We only have to ensure that these translations are also correct w.r.t. $\precsim_{\mathcal{TS}}$, such that Theorem 14 applies.

4.1.1 Transitivity

Proving a property like Theorem 22 can require a lot of effort: the more complex the compiler, the more complex its correctness proof. It is thus crucial that a correctness proof can be broken up into several pieces, *e.g.,* one sub-proof per compiler pass. PILS support such a decomposition thanks to a transitivity-like property. In our setting, where Pilsner compiles via one intermediate language $\mathcal{I}$, we can show the following:

Theorem 15 (Transitivity)**.**

$$\frac{|\Gamma| \vdash M_{\mathcal{T}} \precsim_{\mathcal{TI}} M_{\mathcal{I}} : |\Gamma'| \quad |\Gamma| \vdash M_{\mathcal{I}} \precsim^{*}_{\mathcal{II}} M'_{\mathcal{I}} : |\Gamma'| \qquad \Gamma \vdash M'_{\mathcal{I}} \precsim_{\mathcal{IS}} M_{\mathcal{S}} : \Gamma'}{\Gamma \vdash M_{\mathcal{T}} \precsim_{\mathcal{TS}} M_{\mathcal{S}} : \Gamma'}$$

Here, $\precsim_{\mathcal{TI}}$ relates target modules to intermediate modules, $\precsim_{\mathcal{II}}$ relates intermediate modules to intermediate modules, and $\precsim_{\mathcal{IS}}$ relates intermediate modules to source modules. All are very similar to $\precsim_{\mathcal{TS}}$ and support similar reasoning principles. We will say more about them in the following sections; for now suffice it to say that, since $\precsim_{\mathcal{TI}}$ and $\precsim_{\mathcal{II}}$ involve only untyped languages,[1] the relations themselves are "untyped" and we erase the typing annotations in their environments (*e.g.,* written $|\Gamma|$), leaving just a list of function labels. Notice how using the transitive closure of $\precsim_{\mathcal{II}}$ in the second premise of the rule allows us to verify each IL transformation separately.

4.2 Languages

4.2.1 Language-Generic Approach

In order to avoid duplicate work, we define PILS in a language-generic way, *i.e.,* we define similarity $\precsim_{AB}$ for two abstract languages A and B (for some notion of abstract language to be described), and then instantiate it with different language pairs in order to obtain the desired relations. This has two important benefits:

1. Most of the metatheory, which is quite involved, can be established once and for all. This is particularly crucial because PILS were developed from the start in Coq, and over time the definitions—and thus proofs—had to undergo countless changes. This might simply have been infeasible if it weren't for the language-generic setup.

2. One can easily instantiate PILS with a different intermediate language (or several of them) in order to verify a different compiler. Using modularity (Theorem 14), one can then of course safely link $\mathcal{T}$ code produced by this compiler with Pilsner-produced code.

[1] In order to demonstrate that PILS are not inherently tied to typed languages, we consider a type-erasing compiler, not a completely type-directed one.

In (1), we say "most of the metatheory" because transitivity and the parts of modularity and adequacy that deal with details of module loading are actually not proven generically. Ultimately, it would be nice to do so but it would require some effort to properly axiomatize various properties of the abstract language that these proofs rely on. Moreover, it might require a distinction between *intermediate language* and *non-intermediate language*, with slightly different sets of requirements. For now, it is much easier to simply prove the theorems for the concrete instances (of course this involves many generically proven lemmas). The downside of this is that, in order to verify a compiler using different intermediate languages, one needs to reprove the corresponding transitivity property. Adequacy and modularity, on the other hand, do not need to be reproven as they do not involve the intermediate languages.

To instantiate the generic PILS model and obtain one of the desired similarity relations requires us to provide: (i) the pair of concrete languages, and (ii) the *global world* for this pair. The latter can be seen as a predefined protocol (in the sense of Chapter 2) responsible for fixing calling conventions and data representations. We will discuss global worlds further in Section 4.4.

One point that we glossed over so far is that our generic definition is also not entirely generic—as we will see in the next section, it still essentially bakes in our source language's type structure. Consequently, instantiating PILS as-is with a different source language most likely won't make much sense. This is fine, because we focus on a single source language in this work. Extending this to multiple source languages, perhaps even allowing interoperability between them, is clearly important but left to future work.

Another point we glossed over is that we actually define *two* generic models: a typed one and an untyped one. The former is used when the input language is $\mathcal{S}$, the latter is used in all other cases (where no involved language has static types). However, we will continue to refer to them as just "the generic model", because the untyped one is obtained simply by erasing all the type arguments from the typed one.

4.2.2 Language Specification

We now describe the language abstraction in terms of which PILS are defined. In the subsequent sections, we then present the concrete languages under consideration ($\mathcal{S}$, $\mathcal{I}$, $\mathcal{T}$). Common to all languages are a set of events and a countably infinite set of labels:

$$t \in \mathsf{Evt} ::= \iota \mid ?n \mid !n \qquad\qquad F_1, F_2, \ldots \in \mathsf{Lbl}$$

Events are produced by an executing program; they consist of internal computation (ι) and I/O operations (reading or writing a number n, respectively). Labels are used to identify module components; in this work, we consider a simplistic notion of module as the unit of compilation.

Figure 4.1 presents the abstract language in terms of a signature that any concrete language must implement. Keep in mind that we need to account for a very high-

Domains: $\mathbf{Val}, \mathbf{Cont}, \mathbf{Conf}, \mathbf{Mach}, \mathbf{Mod}, \mathbf{Anch}$

Operators and relations:

- $\cdot \quad \in \mathbf{Conf} \to \mathbf{Conf} \to \mathbf{Conf}$
- $\emptyset \quad \in \mathbf{Conf}$
- $\hookrightarrow \quad \in \mathcal{P}(\mathbf{Mach} \times \mathsf{Evt} \times \mathbf{Mach})$
- $\mathbf{real} \quad \in \mathbf{Conf} \to \mathcal{P}(\mathbf{Mach})$
- $\mathbf{cload} \quad \in \mathbf{Mod} \to \mathbf{Anch} \to (\mathsf{Lbl} \times \mathbf{Val})^* \to \mathcal{P}(\mathbf{Conf} \times \mathbf{Conf})$
- $\mathbf{vload} \quad \in \mathbf{Mod} \to \mathbf{Anch} \to (\mathsf{Lbl} \times \mathbf{Val})^* \to \mathsf{Lbl} \to \mathcal{P}(\mathbf{Val})$
- $\mathbf{frame} \quad \in \mathcal{P}(\mathbf{Conf})$
- $\mathbf{core} \quad \in \mathcal{P}(\mathbf{Conf})$
- $\mathbf{halted} := \{m \in \mathbf{Mach} \mid \nexists t, m'.\ m \overset{t}{\hookrightarrow} m'\}$
- $\mathbf{error} \quad := \{m \in \mathbf{Mach} \mid \forall c.\ m \notin \mathbf{real}(c)\}$

Properties:

- $\mathbf{Conf}$ forms a commutative monoid with $\cdot$ and $\emptyset$.
- $\forall m, t, m'.\ m \overset{t}{\hookrightarrow} m' \wedge m' \notin \mathbf{error} \implies m \notin \mathbf{error}$
- $\forall m, t, m'.\ m \notin \mathbf{error} \wedge m \overset{t}{\hookrightarrow} m' \wedge m' \in \mathbf{error} \implies t = \iota$
- $\forall c_1, c_2, c_1', c_2', m.\ c_1 \in \mathbf{core} \wedge c_2 \in \mathbf{core} \wedge m \in \mathbf{real}(c_1 \cdot c_1') \cap \mathbf{real}(c_2 \cdot c_2') \implies c_1 = c_2 \wedge c_1' = c_2'$
- $\emptyset \in \mathbf{frame}$
- $\forall c, c'.\ c \in \mathbf{frame} \wedge c' \in \mathbf{frame} \implies c \cdot c' \in \mathbf{frame}$

Figure 4.1: Language specification

level language ($\mathcal{S}$) on the one extreme and a very low-level language ($\mathcal{T}$) on the other extreme.

A language must come with a set **Val** of *values*, a set **Cont** of *continuations*, a set **Conf** of *configurations*, a set **Mach** of *machines*, a set **Mod** of *modules*, and a set **Anch** of *anchors* (think: load addresses). The operational semantics is given in the form of a transition system ($\hookrightarrow$) of machines, whose transitions are labelled with events t. Configurations can be thought of as partial machines—they play different roles in different contexts (*e.g.,* they might represent just an $\mathcal{S}$ expression, or just an $\mathcal{I}$ heap, or even a full $\mathcal{T}$ machine). If a configuration c is complete, it is *realized* by a set of machines $\mathbf{real}(c)$. (In all our instantiations, this is either empty, meaning the configuration is invalid or incomplete, or it contains exactly one machine.) Configurations must form a commutative monoid with composition $\cdot$ and neutral element $\emptyset$. This monoid is implicitly partial in the sense that a composition may not be realizable. (Having $\cdot$ itself be total is more convenient for mechanization [59]).

We single out two kinds of configurations, described by **frame** and **core**. *Cores* are what PILS's **E** will (primarily) relate—they can be thought of as expressions. *Frames* are configurations that may be contributed by other modules; they generalize the frame heaps that we have seen in Section 3.6. Intuitively, cores and frames are disjoint, although we do not formally require this. Note that there may be configurations that cannot be decomposed into only a core and a frame.

For **core** we require a uniqueness property saying that a realizable configuration can contain only one core. For **frame** we require two very natura properties: the empty configuration must be a frame; and composition of frames must again yield a frame.

We say a machine is *halted*, $m \in \mathbf{halted}$, iff it cannot step any further. It denotes an *error*, $m \in \mathbf{error}$, iff it does not realize any configuration. We require two properties of **error** (*i.e.,* of **real**). The first is that an error "cannot be undone"; apart from this we will actually never care about steps from an erroneous machine. The second says that an error can only be produced silently. It is not needed for the PILS metatheory but, if violated, the model won't validate some basic refinements.

4.2.2.1 Behavior

Based on these constructs, we can define the set of observation traces that machines can produce.

Definition 20 (Observation traces)**.** The set Obs of observation traces is defined coinductively (as the greatest fixed point of a monotone function in a powerset lattice) such that:

$$\mathsf{Obs} = \{\$, \infty\} \uplus (\mathsf{Evt} \setminus \{\iota\} \times \mathsf{Obs})$$

Accordingly, an observation trace o is a sequence of externally visible events (*i.e.,* I/O events) that

1. either goes on infinitely, representing a computation that never stops issuing I/O operations,

2. or ends with a termination marker (\$), representing a computation that produces a finite number of I/O events and then halts,

3. or ends with a divergence marker (∞), representing an execution that produces a finite number of I/O events but never stops performing internal computations.

A machine m's *behavior* is the set of observation traces that m can possibly generate.

Definition 21 (Machine behavior)**.** We define $\mathbf{behav} \in \mathbf{Mach} \to \mathcal{P}(\mathsf{Obs})$ coinductively such that:

$$\begin{aligned}
\mathbf{behav}(m) \;=\; \{o \mid \exists m'.\; & m \overset{\iota}{\hookrightarrow}{}^{*}\; m' \wedge \\
& (\textsc{Err}) \quad m' \in \mathbf{error} \\
\vee\; & (\textsc{Halt}) \quad o = \$ \wedge m' \in \mathbf{halted} \\
\vee\; & (\textsc{Inf}) \quad o = \infty \wedge \exists m''.\; m' \overset{\iota}{\hookrightarrow} m'' \wedge \infty \in \mathbf{behav}(m'') \\
\vee\; & (\textsc{Evt}) \quad \exists t, o', m''.\; o = t, o' \wedge m' \overset{t}{\hookrightarrow} m'' \wedge t \neq \iota \wedge o' \in \mathbf{behav}(m'')\}
\end{aligned}$$

Notice that errors cannot be be observed explicitly. Instead, when an execution produces an error, then from that point on anything can happen, *i.e.,* an arbitrary trace is appended.

4.2.2.2 Modules

Modules, the units of compilation (and linking), can be thought of as collections of labelled values (the *exports*). These values may refer to external functions (the *imports*) by their unique labels.

In the language specification, the module interface consists of two operations, **cload** and **vload**. The former, **cload**, takes an anchor saying "where" the module is to be loaded and values for each of its imports. It then returns the set of configurations—split into a global and a local part—in which the module is considered loaded as part of a complete program (this will become clearer in the next section). Given the same inputs and additionally the label of one of the exported values, **vload** looks up the module's corresponding value.

The reason why the language specification does not talk about a linking operator is that we will not prove modularity at the language-generic level (cf. Section 4.2.1).

4.2.3 Source Language $\mathcal{S}$

4.2.3.1 Syntax and Static Semantics

The source language $\mathcal{S}$ is essentially a straightforward extension of $\mathsf{F}^{\mu!}$ (Section 3.3) with simple modules and numeric I/O. Figure 4.2 presents the additions to syntax and typing, as well as new definitions.

$$
\begin{array}{lll}
p & ::= \ldots \mid F \mid \mathsf{input} \mid \mathsf{output}\ p \\
\Sigma & ::= \epsilon \mid F{:}\sigma, \Sigma \\
M \in \mathsf{Mod} & ::= \cdot \mid F{=}p, M
\end{array}
$$

$\boxed{\Delta; \Gamma; \Sigma \vdash p : \sigma}$

$$
\cdots \quad \frac{\Delta \vdash \Gamma; \Sigma \quad F{:}\sigma \in \Sigma}{\Delta; \Gamma; \Sigma \vdash F : \sigma} \qquad \frac{\Delta \vdash \Gamma; \Sigma}{\Delta; \Gamma; \Sigma \vdash \mathsf{input} : \mathsf{nat}} \qquad \frac{\Delta; \Gamma; \Sigma \vdash p : \mathsf{nat}}{\Delta; \Gamma; \Sigma \vdash \mathsf{output}\ p : \mathsf{unit}}
$$

$\boxed{\Sigma \vdash M : \Sigma'}$

$$
\frac{\forall \sigma \in \Sigma.\ \exists \tau, \tau'.\ \sigma = \tau \to \tau' \vee \sigma = \forall \alpha.\, \tau}{\Sigma \vdash \epsilon : \epsilon}
$$

$$
\frac{\epsilon; \epsilon; \Sigma \vdash p : \sigma \quad \Sigma, F{:}\sigma \vdash M : \Sigma' \quad F \notin \Sigma \quad |p| = v \quad \sigma = \tau \to \tau' \vee \sigma = \forall \alpha.\, \tau}{\Sigma \vdash (F{=}p, M) : (F{:}\sigma, \Sigma')}
$$

$$
\frac{\Sigma \vdash M_1 : \Sigma_1 \quad \Sigma, \Sigma_1 \vdash M_2 : \Sigma_2}{\Sigma \vdash \mathsf{link}(M_1, M_2) : \Sigma_1, \Sigma_2}
$$

Figure 4.2: Syntax and Static Semantics of the Source language $\mathcal{S}$

The syntax is extended with labels (F), a construct for reading input, and one for writing output. The typing judgment now also carries an environment Σ mapping labels to types. Types are the same as in $\mathsf{F}^{\mu!}$. All typing rules from $\mathsf{F}^{\mu!}$ carry over (they simply ignore the new Σ component), and the three new ones are straightforward.

A module M is simply an ordered list of labelled "programs" p. The module typing judgment $\Sigma \vdash M : \Sigma'$ is also presented in Figure 4.2. For now, ignore the third rule, which we discuss in the next section. The typing ensures the following:

- The module has no free variables.
- The module's imports are faithfully described by Σ.
- The module's exports are faithfully described by Σ' as well as uniquelly labelled and disjoint from its imports.
- The module's components may refer to the labels of other components from the same module, but only in a strict left-to-right dependency order. (Notice how, in the premise of the second rule, F moved from the exports to the imports when checking the remaining module.)
- The module's components all are function values (values of function or $\forall$ types).

The last two restrictions are made to keep the module semantics simple and avoid too much distraction. Note, however, that $\mathcal{S}$ supports universal and existential types and thus can in principle be used to code up ML-style modules [70]. Also, PILS themselves do not bake in any restriction on the dependency order and, being a coinductive method, are perfectly compatible with mutual recursion.

$$
\begin{array}{lll}
e & ::= \ldots \mid F \mid \mathsf{input} \mid \mathsf{output}\ e \\
K & ::= \ldots \mid \mathsf{output}\ K \\
h \in \mathsf{Heap} & := (\mathsf{Loc} \overset{\mathrm{fin}}{\rightharpoonup} \mathsf{Val})_\perp \\
\rho \in \mathsf{Env} & := \mathsf{Lbl} \rightharpoonup \mathsf{Val}
\end{array}
$$

$$\boxed{\langle h; \rho; e\rangle \overset{t}{\hookrightarrow} \langle h'; \rho; e'\rangle}$$

$$
\begin{array}{llll}
\langle h; \rho; e\rangle & \overset{\iota}{\hookrightarrow} & \langle h'; \rho; e'\rangle & (\langle h; e\rangle \hookrightarrow \langle h'; e'\rangle) \\
\langle h; \rho; K[F]\rangle & \overset{\iota}{\hookrightarrow} & \langle h; \rho; K[v]\rangle & (v = \rho(F)) \\
\langle h; \rho; K[\mathsf{input}]\rangle & \overset{?n}{\hookrightarrow} & \langle h; \rho; n\rangle & \\
\langle h; \rho; K[\mathsf{output}\ n]\rangle & \overset{!n}{\hookrightarrow} & \langle h; \rho; \langle\rangle\rangle & \\
\langle h; \rho; e\rangle & \overset{\iota}{\hookrightarrow} & \langle \perp; \rho; e\rangle & (e \neq v \text{ and none of the previous rules apply})
\end{array}
$$

Linking and loading:

$$
\begin{array}{l}
\mathsf{link} \in \mathsf{Mod} \times \mathsf{Mod} \to \mathsf{Mod} \\
\mathsf{link}(M_1, M_2) := M_1, M_2 \\
\\
\mathsf{load} \in \mathsf{Mod} \rightharpoonup \mathsf{Heap} \times \mathsf{Env} \times \mathsf{Exp} \\
\mathsf{load}(M) := (\emptyset, M, |p|\ \langle\rangle) \qquad \text{if } M(F_{\mathrm{main}}) = p
\end{array}
$$

Figure 4.3: Dynamic Semantics of Source language $\mathcal{S}$

4.2.3.2 Dynamic Semantics

As shown in Figure 4.3, expressions e are extended in the obvious way, as are evaluation contexts K. (Since we are not interested in $\mathcal{S}$'s contextual equivalence, we no longer care to define arbitrary contexts.) The notion of value is unchanged from that of $\mathsf{F}^{\mu!}$, *i.e.,* labels are not values. Instead, they are looked up in a read-only environment ρ when in evaluation position. The reduction relation is therefore defined between triples of heap, expression, and label environment (rather than just heap and expression like before).

With the goal of a convenient implementation of configuration composition ($\cdot$), we change our notion of heap to include an undefined heap, *i.e.,* we work with $\mathsf{Heap} = (\mathsf{Loc} \rightharpoonup \mathsf{Val})_\perp$ rather than $\mathsf{Loc} \rightharpoonup \mathsf{Val}$. The undefined heap $\perp$ is not to be confused with the empty heap $(\lambda l.\ \perp)$, for which we continue to write $\emptyset$. We overload the $\sqcup$ notation to refer to the obvious merging of two lifted heaps that returns $\perp$ if one of the heaps is $\perp$ (and, as before, when the heaps overlap). Note that the empty heap $\emptyset$ is its neutral element and $\perp$ its absorbing element.

The reduction relation is now also labelled with events $t \in \mathsf{Evt}$. Let us look at the four rules in Figure 4.3. The first rule incorporates any step from $\mathsf{F}^{\mu!}$, labelling it as

internal computation (this includes $\mathsf{F}^{\mu!}$'s non-deterministic allocation). The second rule describes label lookups via ρ. The third and fourth describe numeric input and output, respectively, which are labelled accordingly. Note that the rule for input is (externally) non-deterministic. Finally, whenever none of these rules are applicable and the expression is not a value, the last rule fires. It transitions to a designated error state, where the heap component is invalidated (we will see in a moment how this naturally fits the language specification). For instance, the last rule fires whenever a label is looked up that is not covered by ρ; it also fires whenever (i) the expression is a read or write memory operation but the heap component doesn't contain the location being read or written, respectively, (ii) the expression is an allocation but the heap already contains any possible location, (iii) the expression is any of these but the heap is invalid ($\bot$).

Module linking is simply concatenation (and thus asymmetric). Indeed, the third module typing rule in the previous Figure 4.2, which talks about linking, can be derived from the first two rules. A complete program is a module without imports (be it the result of linking or not) that exports a designated main function (F_{main}) of type $\mathsf{unit} \to \tau$ (for an arbitrary τ). In order to execute it, it has to be load-ed: the initial heap is empty, the (constant) label environment is the module itself, and the initial expression is a call to the main function. Note that since this call is done in the empty evaluation context, the machine will be halted when the function eventually returns (if it returns) thanks to the side condition $e \neq v$ in the last rule. This matters for adequacy, as will be explained in Section 4.7.2.

4.2.3.3 Implementation of the Language Specification

Figure 4.4 presents the way in which we decide $\mathcal{S}$ to implement the abstract language from Section 4.2.1. Values, continuations, and modules are self-explanatory. Since there is no need for anchors in the source language, we set them to be the singleton set *1*. Machines are the triples for which we defined the reduction $\hookrightarrow$, and $\hookrightarrow$ is precisely $\hookrightarrow$. We write $m.\mathrm{hp}$ to project the heap component out of the tuple m.

In order to define configurations, we make use of the following lifting operation.

Definition 22. Given a set X, we write $X_{\bot,\emptyset}$ for the commutative monoid with carrier $X \uplus \{\emptyset, \bot\}$ (*i.e.,* X extended with two new elements) and a composition ($\oplus$) that is defined as follows:

$$\begin{aligned}
&\forall y \in X_{\bot,\emptyset}.\ \emptyset \oplus y = y = y \oplus \emptyset \\
&\forall y \in X_{\bot,\emptyset}.\ \bot \oplus y = \bot = y \oplus \bot \\
&\forall x_1, x_2 \in X.\ x_1 \oplus x_2 = \bot
\end{aligned}$$

Accordingly, $\emptyset$ is neutral, $\bot$ is absorbing, and composition is *exclusive* in that it maps any two original elements to $\bot$.

Configurations are machines where one or more components may be missing or invalid. For a machine m to realize a configuration c, it must match the configuration

$$
\begin{array}{l}
\textbf{Val} \;\; := \mathsf{Val} \\
\textbf{Cont} \;\; ::= \mathsf{Cont} \\
\textbf{Mod} \;\; ::= \mathsf{Mod} \\
\textbf{Anch} \;\; := \mathit{1} \\
\textbf{Mach} \;\; := \mathsf{Heap} \times \mathsf{Env} \times \mathsf{Exp} \\
\textbf{Conf} \;\; := \mathsf{Heap} \times \mathsf{Env}_{\perp,\emptyset} \times \mathsf{Exp}_{\perp,\emptyset} \\
\\
\textbf{Conf} \text{ monoid:} \\
\quad \boldsymbol{\emptyset} := (\emptyset,\emptyset,\emptyset) \qquad (h,\rho,e) \cdot (h',\rho',e') := (h \sqcup h', \rho \cdot \rho', e \cdot e') \\
\\
\hookrightarrow \;:= \hookrightarrow \\
\textbf{real}(c) := \{m \mid m = c \wedge m.\mathrm{hp} \neq \perp \wedge m.\mathrm{hp} \text{ finite}\} \\
\textbf{cload}(M)(\_)(\rho) := \{(c, \boldsymbol{\emptyset}) \mid \exists \rho'.\ c = (\emptyset, \rho \uplus M \uplus \rho', \emptyset)\} \\
\textbf{vload}(M)(\_)(\_)(F) := \{v \mid \exists p.\ v = |p| \wedge (F{=}p) \in M\}
\end{array}
$$

Figure 4.4: Implemenation of Language Specification for $\mathcal{S}$

($m = c$ with the obvious embedding of **Mach** in **Conf**). Moreover, it must carry a valid and finite heap (finiteness guarantees that allocation will succeed). Notice that the definition of **Conf** and its pointwise composition operation ($\cdot$) imply that heaps can successfully be split across several configurations, but label environments and expressions cannot—they must be defined in exactly one component in order for the composition to be realizable.

The implementation of **vload** is straightforward: it simply looks up the given label in the module. Since the anchor argument carries no information, it is ignored (we write _ to indicate this). The imported values are ignored as well since labels are looked up at runtime (this is also why linking in $\mathcal{S}$ is just concatenation). The implementation of **cload** permits any configuration whose heap is empty, whose expression component is missing, and whose environment contains the imported values and the exported values (*i.e.,* the module itself). Note that the environment may contain additional values (ρ'), thus accounting for other modules that M may be linked with.

4.2.4 Intermediate Language $\mathcal{I}$

Pilsner's intermediate language $\mathcal{I}$ is an untyped, or rather, dynamically typed CPS-variant of $\mathcal{S}$, inspired by Kennedy's intermediate language [39]. Its syntax is shown in Figure 4.5, together with a notion of well-formedness carving out the subset of the language that we are interested in (to be discussed in a moment).

$$a \in \mathsf{BExp} ::= \langle\rangle \mid n \mid x.1 \mid x.2 \mid \mathsf{inl}\ x \mid \mathsf{inr}\ x \mid \langle x_1, x_2\rangle \mid x_1 == x_2 \mid x_1 \odot x_2 \mid \mathsf{fix}\, f(y,k).\, e \mid \Lambda k.\, e$$

$$e \in \mathsf{Exp} ::= \mathsf{let}\ y = a\ \mathsf{in}\ e \mid \mathsf{let}\ k = \mathsf{cont}\ y.\, e\ \mathsf{in}\ e \mid y \leftarrow \mathsf{input};\ e \mid \mathsf{output}\ x;\ e \mid y \leftarrow \mathsf{ref}\ x;\ e \mid x_1 := x_2;\ e \mid y \leftarrow !x;\ e \mid \mathsf{ifnz}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{case}\ x\ (y.\ e_1)\ (y.\ e_2) \mid x_1\ x_2\ k \mid x\ []\ k \mid k\ x \mid \mathsf{halt}\, n$$

$$M \in \mathsf{Mod} ::= \cdot \mid F{=}a, M$$
$$Z ::= \epsilon \mid x, Z \mid k, Z$$

$\boxed{Z \vdash a}$

$$\frac{}{Z \vdash \langle\rangle} \qquad \frac{}{Z \vdash n} \qquad \frac{x \in Z}{Z \vdash x.1} \qquad \frac{x \in Z}{Z \vdash x.2} \qquad \frac{x \in Z}{Z \vdash \mathsf{inl}\ x} \qquad \frac{x \in Z}{Z \vdash \mathsf{inr}\ x}$$

$$\frac{x_1, x_2 \in Z}{Z \vdash \langle x_1, x_2\rangle} \qquad \frac{x_1, x_2 \in Z}{Z \vdash x_1 \odot x_2} \qquad \frac{x_1, x_2 \in Z}{Z \vdash x_1 == x_2}$$

$$\frac{Z^\dagger, f, y, k \vdash e}{Z \vdash \mathsf{fix}\, f(y,k).\, e} \qquad \frac{Z^\dagger, k \vdash e}{Z \vdash \Lambda k.\, e}$$

$\boxed{Z \vdash e}$

$$\frac{Z \vdash a \quad Z, y \vdash e}{Z \vdash \mathsf{let}\ y = a\ \mathsf{in}\ e} \qquad \frac{Z, y \vdash e \quad Z, k \vdash e'}{Z \vdash \mathsf{let}\ k = \mathsf{cont}\ y.\, e\ \mathsf{in}\ e'}$$

$$\frac{Z, y \vdash e}{Z \vdash y \leftarrow \mathsf{input};\ e} \qquad \frac{x \in Z \quad Z \vdash e}{Z \vdash \mathsf{output}\ x;\ e}$$

$$\frac{x \in Z \quad Z \vdash e_1 \quad Z \vdash e_2}{Z \vdash \mathsf{ifnz}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2} \qquad \frac{x \in Z \quad Z, y \vdash e_1 \quad Z, y \vdash e_2}{Z \vdash \mathsf{case}\ x\ (y.\ e_1)\ (y.\ e_2)}$$

$$\frac{x_1, x_2, k \in Z}{Z \vdash x_1\ x_2\ k} \qquad \frac{x, k \in Z}{Z \vdash x\ []\ k} \qquad \frac{k, x \in Z}{Z \vdash k\ x}$$

$$\frac{x \in Z \quad Z, y \vdash e}{Z \vdash y \leftarrow \mathsf{ref}\ x;\ e} \qquad \frac{x \in Z \quad Z, y \vdash e}{Z \vdash y \leftarrow !x;\ e} \qquad \frac{x_1, x_2 \in Z \quad Z \vdash e}{Z \vdash x_1 := x_2;\ e}$$

$\boxed{L \vdash M : L'}$

$$\frac{}{L \vdash \epsilon : \epsilon} \qquad \frac{L \vdash a \quad L, F \vdash M : L' \quad F \notin L}{L \vdash (F{=}a, M) : (F, L')}$$

Figure 4.5: Intermediate language $\mathcal{I}$: Syntax and Well-formedness

4.2.4.1 Syntax

Being in continuation-passing style, every subexpression is explicitly named and functions never "return". Concretely, we distinguish between (*i*) *pure expressions* $a \in \mathsf{BExp}$, which are evaluated in let-bindings and always result in a value but never in any side effects, and (*ii*) *control expressions* $e \in \mathsf{Exp}$. Ignoring conditionals, every control expression is essentially a sequence of let-bindings ending in a function or continuation invocation. For instance, $\mathsf{let}\ k = \mathsf{cont}\ y.\, e_1\ \mathsf{in}\ e_2$ defines a new continuation k with argument y and body e_1, and then executes e_2 (which may use k). Here $y \in \mathrm{TVar}$ is term variable, while $k \in \mathrm{KVar}$ is a continuation variable. Any x in the language syntax is a meta-variable standing for either a term variable or a label, *i.e.,* ranges over $\mathrm{TVar} \uplus \mathsf{Lbl}$ but not KVar. Any y is in binding position.

Functions in $\mathcal{I}$ explicitly take a continuation argument k, and that the syntax for function calls accounts for that as well. Jumping to a continuation k, which can either be the one initially given, or a self-defined one, is written $k\ x$, where x is the continuation's argument (in case k is a function's initial continuation, x should be thought of as the function's return value). Note that one cannot evade the initial continuation: defining a new continuation requires a pre-existing continuation for the final control expression. Also note that continuations are second-class citizens in the sense that the syntax does not allow them to be components of pairs, regular arguments to functions, etc.

The other syntactic forms are self-explanatory (but see below for $\mathsf{halt}\, n$).

4.2.4.2 Well-formedness

The well-formedness predicates in Figure 4.5 ensure that programs are well-scoped. For instance, $Z \vdash a$ guarantees that the only labels and free variables that occur in a are (a subset of) those in Z.

More interestingly, however, well-formedness also enforces that continuations are used in an affine fashion [39], *i.e.,* called at most once. Affinity is exploited by Pilsner's code generation pass, as discussed in Section 5.10. Responsible for enforcing affinity are the rules for functions (the one for $\mathsf{fix}\, f(y,k).\, e$ and the one for $\Lambda k.\, e$): when checking the body e in the premise, we not only extend the original environment Z with the function's bindings, but we also *remove all continuations variables* from Z, written $Z^\dagger$. Hence the only continuation initially available to a function is the one it is passed, namely k. This enforces affinity because, roughly, the only way a continuation could be used multiple times is by embedding it into a function closure. Consider the following example, which is ruled out by well-formedness:

$$\begin{array}{l}\mathsf{fix}\, f(y,k).\\ \quad \mathsf{let}\ g = (\mathsf{fix}\, g(\_,k').\, k\ y)\ \mathsf{in}\\ \quad k\ g\end{array}$$

The function f defines a function g that refers to f's continuation k (which well-formedness forbids). It then passes g to k. If k were now to call its argument g, this

would cause a second invocation of k.

Well-formedness for modules is what one would expect.

4.2.4.3 Dynamic Semantics

Figure 4.6 presents $\mathcal{I}$'s dynamic semantics (which does not care about well-formedness). In contrast to the source language, $\mathcal{I}$ employs an environment-based semantics, where continuations and functions evaluate to closures of code and environment. This avoids the need to reason about substitutions when verifying optimizations, which is often a hassle.

Run-time values consist of the unit value, numbers, pairs, left/right tagged values, and closures $\langle \sigma; a \rangle$. A closure's environment component σ is a mapping from variables (including labels) to values[2].

In order to give the semantics of let-bindings, we first define a straightforward evaluation function that evaluates a pure expression a in an environment σ, written $[\![a]\!]_\sigma$. It is total on inputs whose free variables are covered by σ. The small-step reduction relation is fairly straightforward as well. As one can easily see, the pure expression component a of a closure is always either a function or continuation expression. For instance, we execute a continuation definition $\mathsf{let}\ k = \mathsf{cont}\ y.\, e_1\ \mathsf{in}\ e_2$ by creating a new closure consisting of the current environment σ as well as the continuation's code $\mathsf{cont}\ y.\, e_1$, and add this value to the environment under name k. We would then proceed to execute e_2 in this extended environment.

The $\mathsf{halt}\, n$ expression form is merely a technical device that we use in the transitivity proof (Section 4.8). The compiler is oblivious to it. Since $\mathcal{I}$ is an intermediate language, used internally by Pilsner, adding such a dummy expression is unproblematic. Note that reduction is defined to be stuck (**halted**) for such an expression. It is even ruled out by the well-formedness relation above.

4.2.4.4 Implementation of the Language Specification

Figure 4.7 shows how we implement the language specification with $\mathcal{I}$. Modules, anchors, configurations, etc. are fairly similar to those in the source language. We define **Cont** simply as Val because continuations are already values in the language.

Since $\mathcal{I}$'s semantics is closure-based, expressions and their run-time environment (σ) are tightly coupled, and we do not need to allow configurations containing an environment but no expression, or vice versa. Hence, in the implementation of **Conf**, we lift Env and Exp not individually but together as $(\mathsf{Env} \times \mathsf{Exp})_{\perp,\emptyset}$.

For the same reason, the implementation of **cload** is trivial: because function values already contain their environment, there is nothing else to load, so both the global and local configuration are empty. The implementation of **vload**, on the other hand, is consequently more complicated than before. Defined by recursion on the structure of the given module, it looks up the function expression a associated

[2]The operational semantics of $\mathcal{I}$ treats term variables, continuation variables, and labels all in a uniform way.

$l \in \mathsf{Loc}$
$v \in \mathsf{Val} \quad ::= \langle\rangle \mid n \mid l \mid \langle v_1, v_2\rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \langle \sigma; a\rangle \mid \langle \sigma; \mathsf{cont}\ y.\, e\rangle$
$h \in \mathsf{Heap} := (\mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val})_\perp$
$\sigma \in \mathsf{Env} \quad := \mathsf{Lbl} \uplus \mathsf{TVar} \uplus \mathsf{KVar} \rightharpoonup \mathsf{Val}$

$\boxed{[\![a]\!]_\sigma \in \mathsf{Val}_\perp}$

$$
\begin{aligned}
[\![\langle\rangle]\!]_\sigma &:= \langle\rangle \\
[\![n]\!]_\sigma &:= n \\
[\![\langle x_1, x_2\rangle]\!]_\sigma &:= \langle v_1, v_2\rangle && (\sigma(x_1) = v_1 \wedge \sigma(x_2) = v_2) \\
[\![x.1]\!]_\sigma &:= v_1 && (\sigma(x) = \langle v_1, v_2\rangle) \\
[\![x.2]\!]_\sigma &:= v_2 && (\sigma(x) = \langle v_1, v_2\rangle) \\
[\![\mathsf{inl}\ x]\!]_\sigma &:= \mathsf{inl}\ v && (\sigma(x) = v) \\
[\![\mathsf{inr}\ x]\!]_\sigma &:= \mathsf{inr}\ v && (\sigma(x) = v) \\
[\![\mathsf{fix}\, f(y,k).\, e]\!]_\sigma &:= \langle \sigma; \mathsf{fix}\, f(y,k).\, e\rangle \\
[\![\Lambda k.\, e]\!]_\sigma &:= \langle \sigma; \Lambda k.\, e\rangle \\
[\![x_1 \odot x_2]\!]_\sigma &:= [\![\odot]\!]\,(n_1, n_2) && (\sigma(x_1) = n_1 \wedge \sigma(x_2) = n_2) \\
[\![x_1 == x_2]\!]_\sigma &:= 1 && (\sigma(x_1) = l = \sigma(x_2)) \\
[\![x_1 == x_2]\!]_\sigma &:= 0 && (\sigma(x_1) = l_1 \neq l_2 = \sigma(x_2))
\end{aligned}
$$

$\boxed{\langle h; \sigma, e\rangle \overset{t}{\hookrightarrow} \langle h'; \sigma', e'\rangle}$

$$
\begin{aligned}
\langle h; \sigma, \mathsf{let}\ y = a\ \mathsf{in}\ e\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma[y{\mapsto}v], e\rangle && ([\![a]\!]_\sigma = v) \\
\langle h; \sigma, \mathsf{let}\ k = \mathsf{cont}\ y.\, e_1\ \mathsf{in}\ e_2\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma[k{\mapsto}\langle\sigma; \mathsf{cont}\ y.\, e_1\rangle], e_2\rangle \\
\langle h; \sigma, (y \leftarrow \mathsf{input};\ e)\rangle &\overset{?n}{\hookrightarrow} \langle h; \sigma[y{\mapsto}n], e\rangle \\
\langle h; \sigma, (\mathsf{output}\ x;\ e)\rangle &\overset{!n}{\hookrightarrow} \langle h; \sigma, e\rangle && (\sigma(x) = n) \\
\langle h; \sigma, \mathsf{ifnz}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma, e_2\rangle && (\sigma(x) = 0) \\
\langle h; \sigma, \mathsf{ifnz}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma, e_1\rangle && (\sigma(x) = n \neq 0) \\
\langle h; \sigma, \mathsf{case}\ x\ (y.\ e_1)\ (y.\ e_2)\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma[y{\mapsto}v], e_1\rangle && (\sigma(x) = \mathsf{inl}\ v) \\
\langle h; \sigma, \mathsf{case}\ x\ (y.\ e_1)\ (y.\ e_2)\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma[y{\mapsto}v], e_2\rangle && (\sigma(x) = \mathsf{inr}\ v) \\
\langle h; \sigma, k\ x\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma'[y{\mapsto}v], e\rangle && (\sigma(k) = \langle\sigma'; \mathsf{cont}\ y.\, e\rangle) \\
& && (\sigma(x) = v) \\
\langle h; \sigma, x_1\ x_2\ k\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma'[f, y, k'{\mapsto}v_1, v_2, v], e\rangle \\
& && (\sigma(x_1), \sigma(x_2), \sigma(k) = v_1, v_2, v) \\
& && (v_1 = \langle\sigma'; \mathsf{fix}\, f(y,k').\, e\rangle) \\
\langle h; \sigma, x\ []\ k\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma'[k'{\mapsto}v], e\rangle && (\sigma(x) = \langle\sigma'; \Lambda k'.\, e\rangle) \\
& && (\sigma(k) = v) \\
\langle h; \sigma, (y \leftarrow \mathsf{ref}\ x;\ e)\rangle &\overset{\iota}{\hookrightarrow} \langle h \sqcup [l \mapsto v]; \sigma[y{\mapsto}l], e\rangle && (\sigma(x) = v) \\
& && (h \sqcup [l \mapsto v] \neq \perp) \\
\langle h; \sigma, (y \leftarrow !x;\ e)\rangle &\overset{\iota}{\hookrightarrow} \langle h; \sigma[y{\mapsto}v], e\rangle && (\sigma(x) = l) \\
& && (h(l) = v) \\
\langle h \sqcup [l \mapsto v]; \sigma, (x_1 := x_2;\ e)\rangle &\overset{\iota}{\hookrightarrow} \langle h \sqcup [l \mapsto v']; \sigma, e\rangle && (\sigma(x_1) = l) \\
& && (\sigma(x_2) = v') \\
\langle h; \sigma, e\rangle &\overset{\iota}{\hookrightarrow} \langle \perp; \sigma, e\rangle && (e \neq \mathsf{halt}\, n) \\
& && (\text{no other rule applicable})
\end{aligned}
$$

Figure 4.6: Intermediate language $\mathcal{I}$: Dynamics

$$
\begin{array}{lll}
\textbf{Val} & := & \mathsf{Val} \\
\textbf{Cont} & ::= & \mathsf{Val} \\
\textbf{Mod} & ::= & \mathsf{Mod} \\
\textbf{Anch} & := & \mathit{1} \\
\textbf{Mach} & := & \mathsf{Heap} \times (\mathsf{Env} \times \mathsf{Exp}) \\
\textbf{Conf} & := & \mathsf{Heap} \times (\mathsf{Env} \times \mathsf{Exp})_{\bot,\emptyset}
\end{array}
$$

Conf monoid:
$\emptyset := (\emptyset, \emptyset) \qquad (h, x) \cdot (h', x') := (h \sqcup h', x \oplus x')$

$\hookrightarrow := \hookrightarrow$
$\mathbf{real}(c) := \{m \mid m = c \wedge m.\text{hp} \neq \bot \wedge m.\text{hp finite}\}$

$$
\begin{array}{lcll}
\mathbf{vload}(\epsilon)(\_)(\rho)(F) & := & \emptyset & \\
\mathbf{vload}(F'{=}a, M)(\_)(\rho)(F) & := & \{\langle \rho; a \rangle\} & (F' = F) \\
\mathbf{vload}(F'{=}a, M)(\_)(\rho)(F) & := & \mathbf{vload}(M)(\_)(\rho, (F', \langle \rho; a \rangle))(F) & (F' \neq F) \\
 & & & \\
\mathbf{cload}(M)(\_)(\rho) & := & \{(\emptyset, \emptyset)\} &
\end{array}
$$

Figure 4.7: Intermediate language $\mathcal{I}$: implementation of the language specification

with the given label F and constructs a suitable environment for it. This environment is the given one (σ) extended with all the values previously defined in the module, which are computed recursively.

4.2.5 Target Language $\mathcal{T}$

As shown in Figure 4.8, our target language $\mathcal{T}$ is an idealized machine language featuring instructions for arithmetic, control flow (*e.g.,* conditional jump), memory access, and numeric I/O. Some of them support multiple addressing modes. For instance, if operand o is $\langle r_1 \pm n \rangle$, then the instruction sto o r_2 stores the contents of register r_2 on the stack at the address contained in register r_1, offset by $\pm n$. If $o = [r_1 \pm n]$, then it stores it on the heap instead. The lpc instruction loads the current program counter into the given register.

Code is encoded as data (using an injective function $\mathbb{E}$) and can thus be modified using the standard store instruction (sto).

The alloc instruction allocates a chunk of heap memory of the given size. We assume the choice of memory to be deterministic (but otherwise arbitrary).

$\mathcal{T}$ is idealized also in the sense that machine words are unbounded natural numbers and stack and heap are unbounded as well. The set of registers, though, is fixed (their names, by the way, are merely suggestive and the language semantics does not differentiate them).

$$
\begin{array}{lcl}
r \in \mathsf{Reg} & ::= & \mathsf{sp} \mid \mathsf{clo} \mid \mathsf{arg} \mid \mathsf{env} \mid \mathsf{ret} \mid \mathsf{aux}_1 \mid \mathsf{aux}_2 \mid \mathsf{aux}_3 \\
o \in \mathsf{Opr} & ::= & n \mid r \mid \langle r \pm n \rangle \mid [r \pm n] \\
z \in \mathsf{Instr} & ::= & \mathsf{jmp}\ o \mid \mathsf{jnz}\ r\ o \mid \mathsf{ld}\ r\ o \mid \mathsf{sto}\ o\ r \mid \mathsf{lpc}\ r \mid \mathsf{bop}\ \odot\ r\ o_1\ o_2 \mid \\
 & & \mathsf{input}\ r \mid \mathsf{output}\ r \mid \mathsf{alloc}\ r_1\ r_2
\end{array}
$$

$$
\mathbb{E} \in \mathsf{Instr} \to \mathsf{Word} \qquad \text{(injective)}
$$

$$
\begin{array}{lcl}
\mathsf{Word} & := & \mathbb{N} \\
R \in \mathsf{RegFile} & := & \mathsf{Reg} \to \mathsf{Word} \\
q \in \mathsf{Stack} & := & (\mathsf{Word} \rightharpoonup \mathsf{Word})_\perp \\
h \in \mathsf{Heap} & := & (\mathsf{Word} \rightharpoonup \mathsf{Word})_\perp
\end{array}
$$

$$
\begin{array}{lcll}
[\![-]\!]\,(-,-,-) & \in & \mathsf{Opr} \to \mathsf{RegFile} \times \mathsf{Stack} \times \mathsf{Heap} \rightharpoonup \mathsf{Word} & \\
[\![n]\!]\,(R,q,h) & := & n & \\
[\![r]\!]\,(R,q,h) & := & R(r) & \\
[\![\langle r \pm n \rangle]\!]\,(R,q,h) & := & w & (q(R(r) \pm n) = w) \\
[\![[r \pm n]]\!]\,(R,q,h) & := & w & (h(R(r) \pm n) = w)
\end{array}
$$

$$
\begin{array}{l}
\mathit{store} \in \mathsf{Word} \times \mathsf{RegFile} \times \mathsf{Stack} \times \mathsf{Heap} \to \mathsf{Word} \to \mathsf{Opr} \rightharpoonup \mathsf{Word} \times \mathsf{RegFile} \times \mathsf{Stack} \times \mathsf{Heap} \\
\mathit{store}(n,R,q,h)(w)(r) := (n+1, R[r \mapsto w], q, h) \\
\mathit{store}(n,R,q,h)(w)(\langle r \pm n' \rangle) := (n+1, R, q \sqcup [R(r) \pm n' \mapsto w], h) \\
\mathit{store}(n,R,q,h)(w)([r \pm n']) := (n+1, R, q, h \sqcup [R(r) \pm n' \mapsto w])
\end{array}
$$

$$
\begin{array}{ll}
(n,R,q,h) \overset{t}{\hookrightarrow} (n',R',q',h') & (n \neq 0 \text{ and matching entry in table}) \\
(n,R,q,h) \overset{\iota}{\hookrightarrow} (n,R,q,\perp) & (n \neq 0 \text{ and no matching entry in table})
\end{array}
$$

Instruction z	Condition	Reduction
$\mathsf{jmp}\ o$	$[\![o]\!]\,(R,q,h) = n'$	$\overset{\iota}{\hookrightarrow} (n', R, q, h)$
$\mathsf{jnz}\ r\ o$	$R(r) = 0$	$\overset{\iota}{\hookrightarrow} (n+1, R, q, h)$
$\mathsf{jnz}\ r\ o$	$R(r) \neq 0 \wedge [\![o]\!]\,(R,q,h) = n'$	$\overset{\iota}{\hookrightarrow} (n', R, q, h)$
$\mathsf{ld}\ r\ o$	$[\![o]\!]\,(R,q,h) = w$	$\overset{\iota}{\hookrightarrow} (n+1, R[r \mapsto w], q, h)$
$\mathsf{sto}\ o\ r$	$\mathit{store}(n,R,q,h)(R(r))(o) = \Omega$	$\overset{\iota}{\hookrightarrow} \Omega$
$\mathsf{lpc}\ r$		$\overset{\iota}{\hookrightarrow} (n+1, R[r \mapsto n], q, h)$
$\mathsf{bop}\ \odot\ r\ o_1\ o_2$	$[\![o_1]\!]\,(R,q,h) = n_1 \wedge [\![o_2]\!]\,(R,q,h) = n_2$	
	$\wedge\ \mathit{store}(n,R,q,h)([\![\odot]\!]\,(n_1,n_2))(r) = \Omega$	$\overset{\iota}{\hookrightarrow} \Omega$
$\mathsf{alloc}\ r_1\ r_2$	$\mathit{alloc}(h, R(r_2)) = w$	$\overset{\iota}{\hookrightarrow} (n+1, R[r_1 \mapsto w], q,$ $h \sqcup [w \longmapsto 0^{R(r_2)}])$
$\mathsf{input}\ r$		$\overset{?i}{\hookrightarrow} (n+1, R[r \mapsto i], q, h)$
$\mathsf{output}\ r$		$\overset{!R(r)}{\hookrightarrow} (n+1, R, q, h)$

Figure 4.8: Target language $\mathcal{T}$

The reduction relation is defined between quadruples consisting of a program counter n, a register file R, a stack q, and a heap h. The program counter always refers to the heap.

The table in Figure 4.8 lists the regular reduction rules and is to be interpreted as follows: (n, R, q, h) steps according to the "Reduction" column *if* $h(n) = \mathbb{E}(z)$ (*i.e.,* the current instruction is z) and the condition in the second column holds. For instance, if $h(n)$ encodes the load instruction $\mathsf{ld}\ r\ o$ and $[\![o]\!]\,(R, q, h) = w$, then $(n, R, q, h) \overset{\iota}{\hookrightarrow} (n+1, R[r \mapsto w], q, h)$. This rule makes use of the auxiliary function $[\![-]\!]$ for evaluating the instruction's operand o, resulting in a word w. Its definition, also given in the figure, is straightforward. A second auxiliary function, *store*, is used to compute the result of a memory write (it is undefined for a numeric operand). Finally, if none of the rules from the table apply, then an error rule can fire, analogous to $\mathcal{S}$ and $\mathcal{I}$. This is the case, for instance, if the program counter does not point to a valid instruction, or if a store instruction tries to access an invalid memory address. The exception is when the program counter is 0, which we take to be the system continuation used in loading a whole program (see below). For adequacy (Section 4.7.2) we want such machines to be stuck, *i.e.,* not take any step at all, not even an error step.

4.2.5.1 Stack

Stack and heap are separate simply because both are unbounded in size. There are actually no special instructions for operating the stack, just the standard load and store instructions (ld and sto) in combination with the stack addressing operand $\langle r \pm n \rangle$. We do, however, adopt the convention that the sp register functions as the stack pointer (high-water mark).

We define the set of stacks analogous to that of heaps such that it includes $\perp$. Regarding the operational semantics, this is merely for the sake of uniformity: in fact, we only ever care about stacks that are *fully* defined, *i.e.,* store a value for every address and thus are infinite. This makes sense because the way to "allocate" stack cells is by bumping up the stack pointer and not via a designated instruction whose reduction rule could extend the stack accordingly.

4.2.5.2 Calling Convention

In order to support linking of machine language modules (to be defined in a moment)—no matter their origin—all modules have to agree upon a calling convention. We therefore dictate the following contract for calls to imported functions:

1. Write the function value into register clo.
2. Write the argument value into register arg.
3. Write the return address into register ret.

4. Jump to $[\mathsf{clo} + 0]$, i.e., to wherever the value in the heap at address clo points to.

If and when control eventually reaches the return address, the following must hold[3]:

5. The function's result resides in register arg.

6. Registers env and sp have been preserved, i.e., these are callee-save registers while the rest are caller-save.

7. The contents of the stack (up to the stack pointer) have been preserved as well.

(For simplicity, we will also follow this convention for intra-module calls, but this is an implementation detail of our compilers.)

4.2.5.3 Modules, Loading, and Linking

We now come to Figure 4.9. An $\mathcal{S}$ or $\mathcal{I}$ module will be translated to a *group* in $\mathcal{T}$. A group g consists of two components: a *code pointer table* and a *data block*. The table associates each exported function (label) with the index in the data block where its code starts. The data block is just a sequence of words.

If we define $\mathcal{T}$ modules to be such groups, then linking and relocation would become tricky. Instead, we define $\mathcal{T}$ modules to be *collections* of groups. The idea is that the code in each group can assume that its imported functions will at run time be located in memory right before the group's data block (this will be ensured by the loader), so the code can address them relative to the current program counter. This way, linking can be defined simply as concatenation (think: taking the union) and there is no need for maintaining information about—or performing operations related to—relocation.

Loading requires some work, of course. We first define a recursive auxiliary function $load'$. Here and elsewhere in the figure, some constructions use meta-level standard operations from functional programming, such as *enumerate*, which turns a list a, b, c into a list of pairs $(1, a), (2, b), (3, c)$. $load'$ constructs the initial heap (split into a global and local part, so that it can be reused for the implementation of **cload**) and also returns a mapping from the exported labels to their function values. The argument n denotes the address at which to load the module. In the recursive use of $load'$, this address is increased by $n_u + n_c + n_c + n_d$, the amount of heap cells used for the current group g. The last n_d of these cells simply contain g's data block (see the second part in the definition of h_2). The preceding cells form the *group header*.

The first n_u cells of the group header (n to $n + n_u - 1$) contain the imported "function values" for g (in the terminology of the calling convention above), *i.e.,* they enable code in g to use its imports. Note that these function values are not the code pointers but *point* to the code pointers. This is to enable a uniform compilation of

[3] A module may assume these for its imports and must guarantee them for its exports.

module-level and internal functions, the latter of which cannot simply be represented by code pointers because they generally also need to carry an environment.

It remains to explain the $n_c + n_c$ cells in between. Recall that in the source language's module typing rules, when checking a function, there is no formal distinction between those labels that are imports of the module and those that are defined by the module itself ("left" of the function in question) and thus may be referenced by the function. It is also convenient not to make such a distinction when implementing a compiler from $\mathcal{S}$ to $\mathcal{T}$. For this reason, the loader prepares for each group g a homogeneous environment not only consisting of the group's imports but also of its exports, to be referenced by the group's code. We call this the group's *label environment*. We already discussed the first part of it, namely the first n_u cells of h_2. Cells $n + n_u$ to $n + n_u + n_c - 1$ make up the second part: they contain the exported function values.

What are these values? They must be the addresses of heap cells containing the corresponding code pointers. These cells need to be placed somewhere, so the loader simply places them right after the label environment and right before the code. This means that cells $n + n_u$ to $n + n_u + n_c - 1$ contain the addresses $n + n_u + n_c$ to $n + n_u + n_c + n_c - 1$ (see h_1), and cells $n + n_u + n_c$ to $n + n_u + n_c + n_c - 1$ contain the code pointers listed in the group's table c (after shifting them by the appropriate offset). The reason for putting the exported function values into the global heap (h_1) and all the rest into the local heap (h_2) will become clear in Section 4.4. For now, this separation should be ignored.

Figure 4.10 sums up the heap layout created by *load′*. The dashed chunk is not included in the global heap created by *load′* but will be added to it in the definition of **cload** in the next section.

We define a basic well-formedness predicate on target modules, $\vdash M : L$. Any module produced by our compilers will satisfy this predicate. It states that L lists the labels that M exports (in the order in which they appear in M's groups). Morever, the predicate ensures that the code pointer tables are valid: if (a, b) is a group in M and its table a says that n is the code pointer for export F, then n is a valid index into the data b.

Defining the actual *load* function is now easy. $load(M)$ uses *load′* with address 0 (any would work) and empty import list (since M is assumed to be complete) to compute heaps h_1, h_2, and export mapping x. It then returns a machine whose heap is the union of h_1 and h_2 (note that these are always valid disjoint heaps); whose stack is filled with zeroes; whose register file is zeroed (R_0) except that clo stores w, the value of the program's main function; and whose program counter is n, the value stored at w (*i.e.,* the main function's code pointer). Notice that this simulates a function call to the main function. Also note that since sp is 0, the stack is *logically* empty. Requiring the stack to *physically* contain only zeroes is an arbitrary choice.

Note that the definition shown in the figure assumes that M provides a main function, *i.e.,* $x(F_{\text{main}}) \neq \bot$, and that its value is a valid heap address, *i.e.,* $h_1(w) \neq \bot$. When that is not the case, then $load(M)$ is not defined either, *i.e.,* $load(M) = \bot$.

$$
\begin{array}{l}
g \in \mathsf{Group} := (\mathsf{Lbl} \times \mathsf{Word})^* \times \mathsf{Word}^* \\
M \in \mathsf{Mod} := \mathsf{Group}^* \\
\\
\mathsf{link} \in \mathsf{Mod} \times \mathsf{Mod} \to \mathsf{Mod} \\
\mathsf{link}(M_1, M_2) := M_1, M_2 \\
\\
\mathit{load}' \in \mathsf{Mod} \to \mathsf{Word} \to \mathsf{Word}^* \to \mathsf{Heap} \times \mathsf{Heap} \times (\mathsf{Lbl} \times \mathsf{Word})^* \\
\mathit{load}'(\epsilon)(n)(u) \quad := (\emptyset, \emptyset, \epsilon) \\
\mathit{load}'(g, M)(n)(u) := (h_1 \sqcup h_1', h_2 \sqcup h_2', (x, x')) \\
\quad \text{where } (c, d) := g \\
\qquad\quad n_\rho := \mathit{length}(\rho) \\
\qquad\quad n_c := \mathit{length}(c) \\
\qquad\quad n_d := \mathit{length}(d) \\
\qquad\quad x := \mathit{enumerate}\ (\mathit{map}\ \mathit{fst}\ c)\ (n + n_\rho + n_c) \\
\qquad\quad h_1 := [n + n_u + n_c \longmapsto \mathit{map}\ (\lambda(F, w).n + n_u + n_c + n_c + w)\ c] \\
\qquad\quad h_2 := [n \longmapsto u, \mathit{map}\ \mathit{snd}\ x] \sqcup [n + n_u + n_c + n_c \longmapsto d] \\
\qquad\quad (h_1', h_2', x') := \mathit{load}'(M)(n + n_u + n_c + n_c + n_d)(u, \mathit{map}\ \mathit{snd}\ x) \\
\\
\mathit{load} \in \mathsf{Mod} \rightharpoonup \mathsf{Word} \times \mathsf{RegFile} \times \mathsf{Stack} \times \mathsf{Heap} \\
\mathit{load}(M) := (n, R_0[\mathsf{clo} \mapsto w], (\lambda\_.0), h_1 \sqcup h_2) \\
\quad \text{where } (h_1, h_2, x) := \mathit{load}'(M)(0)(\epsilon) \\
\qquad\quad w := x(F_{\text{main}}) \\
\qquad\quad n := h_1(w) \\
\\
\boxed{\vdash M : L} := \\
\qquad L = \mathit{flatten}\,(\mathit{map}\,(\mathit{map}\,\mathit{fst} \circ \mathit{fst})\,M) \wedge \\
\qquad \forall (a, b) \in M.\ \forall (F, n) \in a.\ n < \mathit{length}(b)
\end{array}
$$

Figure 4.9: Target language $\mathcal{T}$: Modules

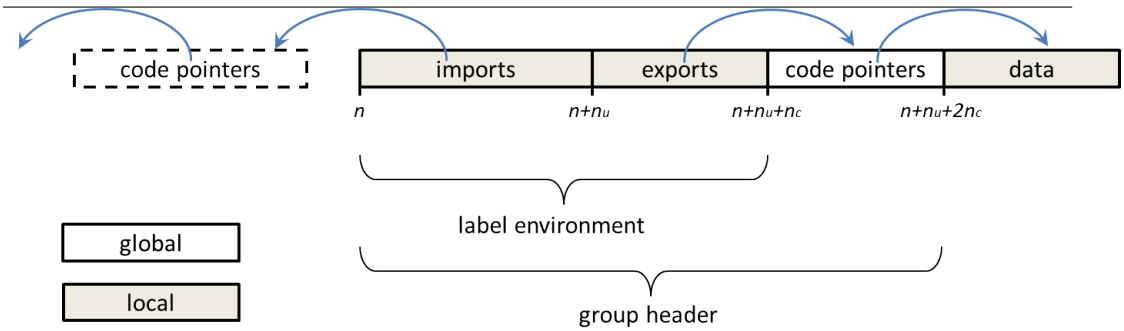


Figure 4.10: Heap layout as set up by load'.

Example To illustrate the heap construction by *load*, let us look at an example. Suppose we have two modules M_a and M_b, each consisting of a single group (*e.g.,* as procuded by one of our compilers):

$$\begin{array}{lcl} M_a & = & g_a \\ g_a & = & ((F_f, 0), (F_g, 20), d_a) \end{array}$$

$$\begin{array}{lcl} M_b & = & g_b \\ g_b & = & ((F_{\text{main}}, 0), d_b) \end{array}$$

Module M_a exports two functions F_f and F_g, whose code starts at offsets 0 and 20 in d_a, respectively. Module M_b exports one function (the "main" function) F_{main}, whose code starts at offset 0 in d_b. We are interested in the complete program

$$M = \mathsf{link}(M_a, M_b) = g_a, g_b$$

obtained by linking M_a and M_b, so the assumption is that M_a has no imports, while M_b's imports are M_a's exports.

The result of $load'(M)(0)(\epsilon)$ is (h_1, h_2, x), which looks as follows:

$$\begin{array}{l} h_1 = h_1^a \sqcup h_1^b \\ h_2 = h_2^a \sqcup h_2^b \\ x = (F_f, 2), (F_g, 3), (F_{\text{main}}, n_a + 7) \end{array}$$

$$\begin{array}{l} h_1^a = [2 \longmapsto 4, 24] \\ h_1^b = [n_a + 7 \mapsto n_a + 8] \end{array}$$

$$\begin{array}{l} h_2^a = [0 \longmapsto 2, 3] \sqcup [4 \longmapsto d_a] \\ h_2^b = [n_a + 4 \longmapsto 2, 3, n_a + 7] \sqcup [n_a + 8 \mapsto d_b] \end{array}$$

$$n_a = \mathit{length}(d_a)$$

For instance, M_b's only label environment, used by the code in d_b to look up functions, starts at address $n_a + 4$ and takes up three cells. (Since M_b is the "last" module, this environment's contents correspond to the export map x.) So if F_{main} wants to call F_g, then F_{main}'s code in d_b will read the heap at address $n_a + 5$ and find the number 3. Indeed, 3 is F_g's function value: the heap at address 3 stores 24, which is F_g's code pointer because d_a starts at address 4.

Note that F_g's function value, 3, is not known at M_b's compile time. On the other hand, d_b knows where to look for F_g because it knows the size of its own environment and thus can access $n_a + 5$ relative to its starting address $(n_a + 8)$.

4.2.5.4 Implementation of the Language Specification

Let us now show how we implement the language specification with $\mathcal{T}$. Values are words (numbers, heap addresses, stack addresses), anchors are words (heap ad-

$$
\begin{array}{lcl}
\textbf{Val} & := & \mathsf{Word} \\
\textbf{Anch} & := & \mathsf{Word} \\
\textbf{Mod} & ::= & \mathsf{Mod} \\
\textbf{Cont} & := & \mathsf{Word} \\
\textbf{Mach} & := & \mathsf{Word} \times \mathsf{RegFile} \times \mathsf{Stack} \times \mathsf{Heap} \\
\textbf{Conf} & := & \mathsf{Word}_{\perp,\emptyset} \times \mathsf{RegFile}_{\perp,\emptyset} \times \mathsf{Stack} \times \mathsf{Heap}
\end{array}
$$

Conf monoid:
$\emptyset := (\emptyset, \emptyset, \emptyset, \emptyset) \qquad (n, R, q, h) \cdot (n', R', q', h') := (n \oplus n', R \oplus R', q \sqcup q', h \sqcup h')$

$\hookrightarrow := \hookrightarrow$
$\mathbf{real}(c) := \{m \mid m = c \wedge m.\mathrm{hp} \neq \perp \wedge m.\mathrm{hp}\ \text{finite} \wedge m.\mathrm{st} \neq \perp\}$

$\mathbf{vload}(M)(n)(\rho)(F) := \{v \mid \exists h_1, h_2, x.$
$\quad load'(M)(n)(map\ snd\ \rho) = (h_1, h_2, x) \wedge x(F) = v\}$

$\mathbf{cload}(M)(n)(\rho) := \{(c_1, c_2) \mid \exists h_1, h_2, x, R, h.$
$\quad load'(M)(n)(map\ snd\ \rho) = (h_1, h_2, x) \wedge \mathrm{dom}(h) = \{w \mid \exists F.\ (F, w) \in \rho\} \wedge$
$\quad h \sqcup h_1 \sqcup h_2 \neq \perp \wedge R(\mathsf{sp}) = 0 \wedge c_1 = (\emptyset, R, (\lambda\_.0), h \sqcup h_1) \wedge c_2 = (\emptyset, \emptyset, \emptyset, h_2)\}$

Figure 4.11: Target language $\mathcal{T}$: Implementation of the Language Specification

dresses), and continuations are words (code pointers, *i.e.,* heap addresses). The configuration monoid and **real** are defined mostly analogously to the previous languages (heap and stack can be split, the rest cannot).

Both the implementation of **vload** and that of **cload** make use of $load'$[4]. **vload** simply looks up the given label F in the generated export map x. **cload** relies on the other two components instead, the global heap h_1 and the local heap h_2. It allows for any global configuration c_1 whose program counter is missing; whose register file is defined and stores 0 in the stack pointer register sp; whose stack is constant zero (matching *load*); and whose heap is h_1 plus some disjoint h. Intuitively, h contains the imported function values (this will become clearer once we see how **cload** is used). Accordingly, **cload** requires that h's domain corresponds to the import map ρ; it must also be disjoint from h_2, but apart from that, h can be arbitrary. The local configuration, c_2, is restricted to be empty except for its heap component, which must be exactly h_2.

[4]Note that $load'$ does not care if the module is complete or not.

4.3 Worlds

Like in PBs (Chapter 3), worlds play a central role in PILS. Before, we distinguished between local worlds and full worlds. A local world was constructed on a per-proof basis, and turned into a full world by linking it with a predefined shared world. This basic story is still true for PILS. However, we have to generalize the notion of shared world somewhat. Instead of only governing global references, the shared world's purpose is now also to formalize the calling convention and data representation that all modules have to agree upon. Moreover, since we are dealing with multiple languages, instead of having a single shared world, we will have one for each language pair of interest: we will define the global worlds $\Omega_{\mathcal{TI}}$, $\Omega_{\mathcal{II}}$, $\Omega_{\mathcal{IS}}$, and $\Omega_{\mathcal{TS}}$.

As previously mentioned, we are going to define PILS generically, *i.e.,* parameterized by two languages A and B and a global world for them, and we are going to have two versions of this generic definition: one where the relations are type-indexed, and one where they are not. We present both at the same time (in several figures to come) by showing the typed version but highlighting (in color) all the type-related components that simply need to be erased in order to obtain the untyped version.

The structure of worlds is shown in Figure 4.12, following some preliminary definitions. First, the set of types used by the typed versions of the model is the same as for PBs, because $\mathcal{S}$'s type language is the same as $\mathsf{F}^{\mu!}$'s. Recall that we extend this language with type names $\mathbf{n}$. The set CTyF of (closed) flexible types is slightly different from before: only function types and type names are considered flexible, reference types are not. In PBs, the treatment of reference types as flexible was convenient, but did not make so much sense conceptually since local worlds could not influence their interpretation. The generalization of PBs to PILS naturally leads to a cleaner treatment of reference types.

Global worlds $\Omega \in \mathrm{GWorld}_{A,B}$ generalize PBs's shared world W_{ref} (from Section 3.6.3.1). They consist of a transitition system T, a *configuration relation* C, and a *query component* Q. The notion of transition systems is the same as before (TrSys was defined in Section 3.6.1). Configuration relations generalize PBs's heap relations. The query component is completely new; it provides several methods for "querying" the global state (see below). We write $A.\mathbf{Val}$ to denote language A's implementation of the abstract notion **Val**, and so on.

Full worlds $W \in \mathrm{World}_{A,B}$ look like extensions of global worlds. They additionally contain an *algebraic well-founded ordered set* O, which will be used to facilitate *stuttering* in proofs (Section 4.5.1). For now, it can be safely ignored.

In the typed setting only, full worlds also contain a *type name interpretation* N. This component provides a set NS of owned type names and a relational interpretation NR of these names (in PBs, this was part of the local knowledge). Local worlds $w \in \mathrm{LWorld}^T_{A,B}$ are like full worlds except that they don't contain the global query handlers and that their configuration relation must only relate frames. As before, local worlds can depend on the state of the global world. We will see the lifting of local worlds to full worlds in a moment.

$$
\begin{array}{lcl}
\mathbf{n} \in \mathsf{TyNam} & & \\
\sigma \in \mathsf{Ty} & ::= & \alpha \mid \mathbf{n} \mid \mathsf{unit} \mid \mathsf{nat} \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid \sigma_1 \to \sigma_2 \mid \forall\alpha.\,\sigma \mid \exists\alpha.\,\sigma \mid \\
& & \mu\alpha.\,\sigma \mid \mathsf{ref}\,\sigma \\
\\
\mathsf{CTy} & ::= & \{\,\sigma \in \mathsf{Ty} \mid \mathrm{fv}(\sigma) = \emptyset\,\} \\
\mathsf{CTyF} & := & \{(\tau_1 \to \tau_2) \in \mathsf{CTy}\} \cup \{(\forall\alpha.\,\tau) \in \mathsf{CTy}\} \cup \{\mathbf{n} \in \mathsf{CTy}\} \\
\\
\mathsf{VRelF}_{A,B} & := & \mathsf{CTyF} \to \mathcal{P}(A.\mathbf{Val} \times B.\mathbf{Val}) \\
\mathsf{VRel}_{A,B} & := & \mathsf{CTy} \to \mathcal{P}(A.\mathbf{Val} \times B.\mathbf{Val}) \\
\mathsf{CRel}_{A,B} & := & \mathcal{P}(A.\mathbf{Conf} \times B.\mathbf{Conf}) \\
\mathsf{MRel}_{A,B} & := & \mathcal{P}(A.\mathbf{Mach} \times B.\mathbf{Mach}) \\
\mathsf{ERel}_{A,B} & := & \mathsf{CTy} \to \mathsf{CRel}_{A,B} \\
\mathsf{KRel}_{A,B} & := & \mathsf{CTy} \to \mathsf{CTy} \to \mathcal{P}(A.\mathsf{Cont} \times B.\mathsf{Cont}) \\
\\
\text{Queries } (\mathbf{v}, \mathbf{v}' \in L.\mathbf{Val} \text{ and } \mathbf{k} \in L.\mathbf{Cont}): & & \\
vq \in \mathsf{VQry}_L & ::= & \mathsf{unit} \mid \mathsf{nat}\,n \mid \mathsf{pair}\,\mathbf{v}\,\mathbf{v}' \mid \mathsf{inl}\,\mathbf{v} \mid \mathsf{inr}\,\mathbf{v} \mid \mathsf{roll}\,\mathbf{v} \mid \mathsf{pack}\,\mathbf{v} \mid \mathsf{fix} \mid \mathsf{gen} \mid \\
& & \mathsf{goodfix} \mid \mathsf{goodgen} \\
cq \in \mathsf{CQry}_L & ::= & \mathsf{app}\,\mathbf{v}\,\mathbf{v}'\,\mathbf{k} \mid \mathsf{inst}\,\mathbf{v}\,\mathbf{k} \mid \mathsf{ret}\,\mathbf{v}\,\mathbf{k} \\
\\
\mathsf{QH}^T_{A,B} & := & \{\,(\mathsf{vqh_a} \in T.\mathsf{S} \overset{\text{mon}}{\to} \mathsf{VQry}_A \to \mathcal{P}(A.\mathbf{Val}) \\
& & \quad ,\mathsf{vqh_b} \in T.\mathsf{S} \overset{\text{mon}}{\to} \mathsf{VQry}_B \to \mathcal{P}(B.\mathbf{Val}) \\
& & \quad ,\mathsf{cqh_a} \in T.\mathsf{S} \to \mathsf{CQry}_A \to \mathcal{P}(A.\mathbf{Conf}) \\
& & \quad ,\mathsf{cqh_b} \in T.\mathsf{S} \to \mathsf{CQry}_B \to \mathcal{P}(B.\mathbf{Conf}) \\
& & \quad ,\mathsf{rqh} \in T.\mathsf{S} \overset{\text{mon}}{\to} \mathsf{VRel}_{A,B})\,\} \\
\\
\mathsf{CR}^T_{A,B} & := & (T.\mathsf{S} \to \mathsf{VRelF}_{A,B}) \overset{\text{mon}}{\to} T.\mathsf{S} \to \mathsf{CRel}_{A,B} \\
\mathsf{LCR}^T_{A,B} & := & (T.\mathsf{S} \to \mathsf{VRelF}_{A,B}) \overset{\text{mon}}{\to} T.\mathsf{S} \to \mathcal{P}(A.\mathbf{frame} \times B.\mathbf{frame}) \\
\\
\mathsf{NI}^T_{A,B} & := & \{\,(\mathsf{NS} \in \mathcal{P}(\mathsf{TyNam}) \\
& & \quad ,\mathsf{NR} \in (T.\mathsf{S} \to \mathsf{VRelF}_{A,B}) \overset{\text{mon}}{\to} T.\mathsf{S} \overset{\text{mon}}{\to} \mathsf{TyNam} \to \mathcal{P}(A.\mathbf{Val} \times B.\mathbf{Val})) \mid \\
& & \qquad \forall G, s.\ \forall \mathbf{n} \notin \mathsf{NS}.\ \mathsf{NR}(G)(s)(\mathbf{n}) = \emptyset\,\} \\
\\
\mathsf{GWorld}_{A,B} & := & \{\,(\mathsf{T} \in \mathsf{TrSys}, \mathsf{C} \in \mathsf{CR}^{\mathsf{T}}_{A,B}, \mathsf{Q} \in \mathsf{QH}^{\mathsf{T}}_{A,B})\,\} \\
\mathsf{LWorld}^T_{A,B} & := & \{\,(\mathsf{T} \in \mathsf{TrSys}, \mathsf{C} \in \mathsf{LCR}^{T\times\mathsf{T}}_{A,B}, \mathsf{N} \in \mathsf{NI}^{T\times\mathsf{T}}_{A,B}, \mathsf{O} \in \mathsf{AWFO})\,\} \\
\mathsf{World}_{A,B} & := & \{\,(\mathsf{T} \in \mathsf{TrSys}, \mathsf{C} \in \mathsf{CR}^{\mathsf{T}}_{A,B}, \mathsf{Q} \in \mathsf{QH}^{\mathsf{T}}_{A,B}, \mathsf{N} \in \mathsf{NI}^{\mathsf{T}}_{A,B}, \mathsf{O} \in \mathsf{AWFO})\,\}
\end{array}
$$

Figure 4.12: Worlds

At this point, the reader is probably wondering where the local knowledge component has gone (apart from the interpretion of type names), which played such a crucial role in PBs. We have seen in Section 3.11 how one can in princple always pick the local knowledge to be the greatest consistent relation. Here we go one step further by getting rid of the local knowledge component in worlds altogether. We will explain this in detail in Section 4.5.

4.3.1 Queries

Let us talk about a (global) world's query component. It provides five query methods, as shown in the definition of $\mathsf{QH}^T_{A,B}$ in Figure 4.12. Part of their purpose is to act as the interface to access information stored in the global world's state. We will see later how they are used exactly but we want to give some intuition here.

Configuration Queries Recall that PBs's RET case asserts that the two computations have finished and returned similar values (*e.g.,* Section 3.5). In our source language $\mathcal{S}$, termination is a syntactic property and is trivial to check. But what does it mean for a low-level $\mathcal{T}$ computation to be "finished"?

In order to define PILS in a language-generic fashion, we require the global world to provide a way of answering such a question. This is done in the form of two query handlers cqh_a (for the "output" language A) and cqh_b (for the input language B). There are three queries $cq \in \mathsf{CQry}_L$ concerning the configurations of a language L. The query $\mathsf{app}\, f\, \mathbf{v}\, \mathbf{k}$ asks for the configurations that currently represent a call of function f with argument $\mathbf{v}$ and continuation $\mathbf{k}$; $\mathsf{inst}\, g\, \mathbf{k}$ asks for the configurations that represent an instantiation of generalization g with continuation $\mathbf{k}$; and $\mathsf{ret}\, \mathbf{v}\, \mathbf{k}$ asks for those that represent a return of value $\mathbf{v}$ to continuation $\mathbf{k}$. The reason why we care about continuations here as well is that in low-level languages such as $\mathcal{T}$, we cannot tell whether a computation has returned without knowing its initial continuation (think: return address).

In effect, this means that the global world's implementation of cqh_a and cqh_b determines a major aspect of the calling conventions.

Value Queries Recall that PBs's RET and STEP/CALL cases require that certain values (*e.g.,* the returned results) are related by the value closure of the global knowledge. The value closure is a straightforward lifting of a relation from functions to other value forms, *e.g.,* by saying that two pairs are related iff their first projections are related (recursively) and their second projections as well.

In order to define the value closure operation generically in the PILS setting, we first need to have a uniform way of determining how $\mathcal{S}$'s value forms are represented by the two languages A and B. This is easy for our source and intermediate language: given a value, it is trivial to tell whether it represents, say, a certain pair. On the machine language side, though, values are just machine words, and whether or not

$$\langle\!\langle - \rangle\!\rangle^{(-)} \in \mathsf{VRelF}_{A,B} \to T.\mathsf{S} \to \mathsf{VRel}_{A,B} \qquad (\text{implicit: } A, B;\ T \in \mathsf{TrSys};\ Q \in \mathsf{QH}^T_{A,B})$$

$$\begin{aligned}
\langle\!\langle R \rangle\!\rangle^s := {} & \{\, (\tau \to \tau', \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in R \mid \mathbf{v}_\mathsf{a} \in Q.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{fix}) \wedge \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{fix}) \,\} \\
& \cup \{\, (\forall\alpha.\,\tau, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in R \mid \mathbf{v}_\mathsf{a} \in Q.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{gen}) \wedge \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{gen}) \,\} \\
& \cup \{\, (\mathbf{n}, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in R \mid \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{name}) \,\} \\
& \cup \{\, (\mathsf{unit}, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \mid \mathbf{v}_\mathsf{a} \in Q.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{unit}) \wedge \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{unit}) \,\} \\
& \cup \{\, (\mathsf{nat}, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \mid \exists n.\ \mathbf{v}_\mathsf{a} \in Q.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{nat}\, n) \wedge \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{nat}\, n) \,\} \\
& \cup \{\, (\tau_1 \times \tau_2, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \mid \exists \mathbf{v}^1_\mathsf{a}, \mathbf{v}^2_\mathsf{a}, \mathbf{v}^1_\mathsf{b}, \mathbf{v}^2_\mathsf{b}.\ (\mathbf{v}^1_\mathsf{a}, \mathbf{v}^1_\mathsf{b}) \in \langle\!\langle R \rangle\!\rangle^s(\tau_1) \wedge (\mathbf{v}^2_\mathsf{a}, \mathbf{v}^2_\mathsf{b}) \in \langle\!\langle R \rangle\!\rangle^s(\tau_2) \wedge \\
& \qquad\qquad \mathbf{v}_\mathsf{a} \in Q.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{pair}\, \mathbf{v}^1_\mathsf{a}\, \mathbf{v}^2_\mathsf{a}) \wedge \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{pair}\, \mathbf{v}^1_\mathsf{b}\, \mathbf{v}^2_\mathsf{b}) \,\} \\
& \cup \{\, (\tau_1 + \tau_2, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \mid \exists \mathbf{v}^1_\mathsf{a}, \mathbf{v}^1_\mathsf{b}.\ (\mathbf{v}^1_\mathsf{a}, \mathbf{v}^1_\mathsf{b}) \in \langle\!\langle R \rangle\!\rangle^s(\tau_1) \wedge \\
& \qquad\qquad \mathbf{v}_\mathsf{a} \in Q.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{inl}\, \mathbf{v}^1_\mathsf{a}) \wedge \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{inl}\, \mathbf{v}^1_\mathsf{b}) \,\} \\
& \cup \{\, (\tau_1 + \tau_2, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \mid \exists \mathbf{v}^2_\mathsf{a}, \mathbf{v}^2_\mathsf{b}.\ (\mathbf{v}^2_\mathsf{a}, \mathbf{v}^2_\mathsf{b}) \in \langle\!\langle R \rangle\!\rangle^s(\tau_2) \wedge \\
& \qquad\qquad \mathbf{v}_\mathsf{a} \in Q.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{inr}\, \mathbf{v}^2_\mathsf{a}) \wedge \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{inr}\, \mathbf{v}^2_\mathsf{b}) \,\} \\
& \cup \{\, (\mu\alpha.\,\tau, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \mid \exists \mathbf{v}'_\mathsf{a}, \mathbf{v}'_\mathsf{b}.\ (\mathbf{v}'_\mathsf{a}, \mathbf{v}'_\mathsf{b}) \in \langle\!\langle R \rangle\!\rangle^s(\tau[\mu\alpha.\,\tau/\alpha]) \wedge \\
& \qquad\qquad \mathbf{v}_\mathsf{a} \in Q.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{roll}\, \mathbf{v}'_\mathsf{a}) \wedge \mathbf{v}_\mathsf{b} \in Q.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{roll}\, \mathbf{v}'_\mathsf{b}) \,\} \\
& \cup \{\, (\mathsf{ref}\, \tau, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \mid (\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in Q.\mathsf{rqh}(s)(\tau) \,\}
\end{aligned}$$

$$\overline{G}(s) := \langle\!\langle G(s) \rangle\!\rangle^s$$

Figure 4.13: Value Closure.

a given word represents a pair typically very much depends on the memory in which it is considered (and similarly for other forms).

We therefore require that the global world provide a mechanism to answer questions such as: does value $\mathbf{v}$ currently represent a pair of values $\mathbf{v}_1$ and $\mathbf{v}_2$? This mechanism takes the form of two value query handlers vqh_a and vqh_b, very much like the configuration query handlers that we have just seen. They take a state and a value query $vq \in \mathsf{VQry}_L$ (for $L \in \{A, B\}$), and return a set of values. The above example translates into $\mathbf{v} \in \mathsf{vqh}_\mathsf{a}(s)(\mathsf{pair}\, \mathbf{v}_1\, \mathbf{v}_2)$, for instance. Crucially, these handlers must be monotone in their state argument w.r.t. $\sqsubseteq$, reflecting the immutability of values (*e.g.,* "once a pair, always a pair").

For global reference values, the global world also needs to keep track of which A locations correspond to which B locations (in the two modules one reasons about). Recall the global world W_{ref} from PBs, where a state s_{ref} was a partial bijection between heap locations. In PILS, a global world must provide rqh for querying this correspondence. (Essentially, each of the global worlds that we will define, will still carry such a partial bijection in its state, and rqh is simply the interface to it.) Actually, since we will never be interested in asking whether some value is a reference without also asking about its counterpart, the language of value queries (VQry_L) mentioned above does not include a construct for references.

4.3.2 Value Closure

With the help of these three query handlers, the generalization of PBs's value closure operator is fairly straightforward. It is shown in Figure 4.13[5] as $\langle\!\langle - \rangle\!\rangle^{(-)}$. Like before, it is defined as a least fixed point. Unlike before, however, it now takes the world's transition system T and query method collection Q (both implicitly) as well as the current state s (explicitly) as arguments, so that it can invoke Q's query methods on s to ensure that the values have the right "shape". Note that references are dealt with using $Q.\mathsf{rqh}$, and all the rest using $Q.\mathsf{vqh}_\mathsf{a}$ and $Q.\mathsf{vqh}_\mathsf{b}$. There are a few points to note.

- Admittedly, the name "value closure" is no longer justified since $\langle\!\langle R \rangle\!\rangle^s$ is not always a superset of R. We nevertheless keep the name for the sake of consistency.

  Related to this, we will see in the later parts of this section that PILS are set up such that whenever we take related values from the global knowledge, we do so via its value closure—never directly from the unfiltered global knowledge (in contrast to PBs).

- The reason why the operator does not simply take the whole global world as implicit argument is that in the definition of our particular global worlds (Section 4.4), we already want to make use of the value closure operator—hence there would be a cycle.

- As a convenient shorthand, we write $\overline{G}(s)$ for $\langle\!\langle G(s) \rangle\!\rangle^s$, *i.e.,* the value closure over $G(s)$ relative to state s.

- The asymmetry in the case of type names seems odd: we require $\mathbf{v}_\mathsf{b}$ to satisfy the name query as expected, but we require nothing of $\mathbf{v}_\mathsf{a}$. What is name anyway?

  Like in PBs, type names allow us to attach relational interpretations to abstract types, and it is important that these relations can be essentially arbitrary. In our PILS transivitity proof, it turns out that it is also important that the *untyped* value closure has a "type name" case, namely when showing something along the lines of $\overline{G}(\mathbf{n}) \subseteq \overline{G_1} \circ \overline{G_2}(\mathbf{n})$ (see Lemma 64 in Section 4.8). There we must be able to somehow decompose the relational interpretation of $\mathbf{n}$. However, we cannot simply add an unrestricted case of the form

  $$\cup \{ (\mathbf{n}, \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in R \}$$

  to the value closure, because then inverting $\overline{G}$, in order to reason about programs that use related values, would force us to deal with the ever-present possibility of essentially not knowing anything at all about these values.

[5] Here and in other places, we sometimes write $W.\mathsf{S}$ short for $W.\mathsf{T}.\mathsf{S}$, and similar for other components when there is no ambiguity.

To avoid this hopeless situation, we introduced the name query and require in the value closure's type name case that the left-side values satisfy it. Moreover, we defined $\mathcal{I}$'s value query handler such that it only admits a very particular class of values (cf. Figure 4.18). These values suffice for the purposes of our transitivity proof, and, crucially, they cannot ever be succesfully used: any attempt to deconstruct them results in an error (this implies that they are distinct from all other value forms occurring in the value closure). This knowledge makes it easy to deal with the artifical type name case in proofs.

- The reader may also wonder about the two value queries that are not used by the value closure operator: goodfix and goodgen. These are the PILS verions of PBs's FixVal and GenVal. They will be imposed on what the PILS substitute for PBs's local knowledge is. Like before, their purpose is to ensure that certain relational compositions that come up in the proof of transitivity are empty. This will become clear in Section 4.8.2.

4.3.3 Lifting and Separating Conjunction of Local Worlds

Figure 4.14 shows the PILS versions of lifting and separating conjunction of local worlds (the PB versions were defined in Section 3.6.4). All definitions in this figure take as implicit arguments two languages A and B; some also implicitly take the global world $\Omega \in \mathrm{GWorld}_{A,B}$ or just a transition system T. We will make sure that it is clear from the context how these implicit arguments are instantiated when using the operators later on.

The first construct in the figure is the separating conjunction $\otimes$, generalizing $\otimes$ (Figure 3.14) from heap relations to configuration relations.

Next is the lifting operator. It composes its argument w with the global world Ω much like in PBs. Note how the resulting full world $w{\uparrow}$ forwards any queries to Ω. Also note that, when accessing Ω's configuration relation $\Omega.\mathsf{C}$, we need to pass it a global knowledge indexed by $\Omega.\mathsf{T}$ states, but the one we are given (G) is indexed by $w{\uparrow}.\mathsf{T}$ states; thus, instead of passing G, we pass $G(-, s)$. (This does not apply to $w.\mathsf{C}$, because it uses the product transition system anyways.)

The product $\otimes$ of local worlds is defined iff their name sets are disjoint, like for PBs. The definition is straightforward except for one point: both in the construction of the product's configuration relation and its name interpretation, we are given a G indexed by $T \times (w_1 \otimes w_2).\mathsf{T}$ states, but need to pass one indexed by $T \times w_1.\mathsf{T}$ states to w_1, and one indexed by $T \times w_2.\mathsf{T}$ states to w_2. We would like to pass $\lambda(s'_{\mathrm{g}}, s'_1).\ G(s'_{\mathrm{g}}, (s'_1, s_2))$ to w_1 (and similarly for w_2), but this is not guaranteed to be monotone in s_2. Consequently, we wouldn't be able to show the required monotonicity of the composed NR. For this reason, we first monotonize the global knowledge constructions, written $G^{s_2}_{\langle 1\rangle}$ and $G^{s_1}_{\langle 2\rangle}$. This monotization is harmless because ultimately we are only interested in monotone G anyways.

$$(-) \otimes (-) \in \mathsf{CRel}_{A,B} \to \mathsf{CRel}_{A,B} \to \mathsf{CRel}_{A,B}$$
$$R \otimes R' := \{\, (c_1 \cdot c_1', c_2 \cdot c_2') \mid (c_1, c_2) \in R \wedge (c_1', c_2') \in R' \,\}$$

$$(-)\uparrow \in \mathrm{LWorld}_{A,B}^{\Omega.\mathsf{T}} \to \mathrm{World}_{A,B}$$
$$\begin{array}{lcl}
w\uparrow.\mathsf{T} & := & \Omega.\mathsf{T} \times w.\mathsf{T} \\
w\uparrow.\mathsf{Q}.\mathsf{vqh}_\mathsf{a}(s_\mathrm{g}, s) & := & \Omega.\mathsf{Q}.\mathsf{vqh}_\mathsf{a}(s_\mathrm{g}) \\
w\uparrow.\mathsf{Q}.\mathsf{vqh}_\mathsf{b}(s_\mathrm{g}, s) & := & \Omega.\mathsf{Q}.\mathsf{vqh}_\mathsf{b}(s_\mathrm{g}) \\
w\uparrow.\mathsf{Q}.\mathsf{cqh}_\mathsf{a}(s_\mathrm{g}, s) & := & \Omega.\mathsf{Q}.\mathsf{cqh}_\mathsf{a}(s_\mathrm{g}) \\
w\uparrow.\mathsf{Q}.\mathsf{cqh}_\mathsf{b}(s_\mathrm{g}, s) & := & \Omega.\mathsf{Q}.\mathsf{cqh}_\mathsf{b}(s_\mathrm{g}) \\
w\uparrow.\mathsf{Q}.\mathsf{rqh}(s_\mathrm{g}, s) & := & \Omega.\mathsf{Q}.\mathsf{rqh}(s_\mathrm{g}) \\
w\uparrow.\mathsf{C}(G)(s_\mathrm{g}, s) & := & \Omega.\mathsf{C}(G(-, s))(s_\mathrm{g}) \otimes w.\mathsf{C}(G)(s_\mathrm{g}, s) \\
w\uparrow.\mathsf{N} & := & w.\mathsf{N} \\
w\uparrow.\mathsf{O} & := & w.\mathsf{O}
\end{array}$$

$$(-)_{\langle 1 \rangle}^{(-)} \in ((\Omega.\mathsf{T} \times (T_1 \times T_2)).\mathsf{S} \to \mathsf{VRelF}_{A,B}) \to T_2.\mathsf{S} \to (\Omega.\mathsf{T} \times T_1).\mathsf{S} \to \mathsf{VRelF}_{A,B}$$
$$G_{\langle 1 \rangle}^{s_2}(s_\mathrm{g}, s_1) := \bigcup \{\, G(s_\mathrm{g}, (s_1, s_2')) \mid s_2' \sqsubseteq s_2 \,\}$$

$$(-)_{\langle 2 \rangle}^{(-)} \in ((\Omega.\mathsf{T} \times (T_1 \times T_2)).\mathsf{S} \to \mathsf{VRelF}_{A,B}) \to T_1.\mathsf{S} \to (\Omega.\mathsf{T} \times T_2).\mathsf{S} \to \mathsf{VRelF}_{A,B}$$
$$G_{\langle 2 \rangle}^{s_1}(s_\mathrm{g}, s_2) := \bigcup \{\, G(s_\mathrm{g}, (s_1', s_2)) \mid s_1' \sqsubseteq s_1 \,\}$$

$$(-) \otimes (-) \in \mathrm{LWorld}_{A,B}^{T} \to \mathrm{LWorld}_{A,B}^{T} \rightharpoonup \mathrm{LWorld}_{A,B}^{T}$$
$$\begin{array}{lcl}
(w_1 \otimes w_2).\mathsf{T} & := & w_1.\mathsf{T} \times w_2.\mathsf{T} \\
(w_1 \otimes w_2).\mathsf{C}(G)(s_\mathrm{g}, (s_1, s_2)) & := & \Omega.\mathsf{C}(G_{\langle 1 \rangle}^{s_2})(s_\mathrm{g}, s_1) \otimes w.\mathsf{C}(G_{\langle 2 \rangle}^{s_1})(s_\mathrm{g}, s_2) \\
(w_1 \otimes w_2).\mathsf{N}.\mathsf{NS} & := & w_1.\mathsf{N}.\mathsf{NS} \uplus w_2.\mathsf{N}.\mathsf{NS} \\
(w_1 \otimes w_2).\mathsf{N}.\mathsf{NR}(G)(s_\mathrm{g}, (s_1, s_2)) & := & w_1.\mathsf{N}.\mathsf{NR}(G_{\langle 1 \rangle}^{s_2})(s_\mathrm{g}, s_1) \cup w_2.\mathsf{N}.\mathsf{NR}(G_{\langle 2 \rangle}^{s_1})(s_\mathrm{g}, s_2) \\
(w_1 \otimes w_2).\mathsf{O} & := & w_1.\mathsf{O} \times w_2.\mathsf{O}
\end{array}$$

Figure 4.14: Lifting and separating conjunction of local worlds.

$$
\begin{array}{lcl}
\Omega^{AB}.\mathsf{T} & := & \mathsf{T}^{A} \times \mathsf{T}^{AB}_{\mathsf{ref}} \times \mathsf{T}^{B} \\
\Omega^{AB}.\mathsf{Q}.\mathsf{vqh}_{\mathsf{a}}(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}) & := & \mathsf{vqh}^{A}(s_{\mathsf{a}}) \\
\Omega^{AB}.\mathsf{Q}.\mathsf{vqh}_{\mathsf{b}}(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}) & := & \mathsf{vqh}^{B}(s_{\mathsf{b}}) \\
\Omega^{AB}.\mathsf{Q}.\mathsf{cqh}_{\mathsf{a}}(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}) & := & \mathsf{cqh}^{A}(s_{\mathsf{a}}) \\
\Omega^{AB}.\mathsf{Q}.\mathsf{cqh}_{\mathsf{b}}(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}) & := & \mathsf{cqh}^{B}(s_{\mathsf{b}}) \\
\Omega^{AB}.\mathsf{Q}.\mathsf{rqh}(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}) & := & \mathsf{rqh}^{AB}(s_{\mathsf{ref}}) \\
\Omega^{AB}.\mathsf{C}(G)(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}) & := & (\mathsf{P}^{A}(s_{\mathsf{a}}) \times \{B.\emptyset\}) \otimes \mathsf{C}^{AB}_{\mathsf{ref}}(G)(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}) \otimes (\{A.\emptyset\} \times \mathsf{P}^{B}(s_{\mathsf{b}}))
\end{array}
$$

Figure 4.15: Schematic Definition of Concrete Global Worlds.

4.4 Concrete Global Worlds

As we are going to see in Section 4.5, PILS define at the top-level a generic *module similarity*, written $\precsim_{\Omega}$. This basically generalizes PBs open program equivalence (here we consider modules because they are the objects that we compile and link). This relation is relative to a global world $\Omega \in \mathrm{GWorld}_{A,B}$ for the language pair A, B, with A intuitively being the language *to* which is being translated and B the one *from* which is being translated. Eventually we will instantiate this relation with four global worlds in order to obtain the four models of interest. In this section, we define these four global worlds:

- $\Omega_{\mathcal{TS}} \in \mathrm{GWorld}_{\mathcal{T},\mathcal{S}}$ (typed)
- $\Omega_{\mathcal{TI}} \in \mathrm{GWorld}_{\mathcal{T},\mathcal{I}}$ (untyped)
- $\Omega_{\mathcal{II}} \in \mathrm{GWorld}_{\mathcal{I},\mathcal{I}}$ (untyped)
- $\Omega_{\mathcal{IS}} \in \mathrm{GWorld}_{\mathcal{I},\mathcal{S}}$ (typed)

Some details in this section may be hard to understand without familiarity with the actual PILS relations that will be discussed in Section 4.5. It can therefore be helpful to revisit this section at a later point. (We did consider a different order of presentation but prefer the current one.)

To avoid duplicate work, we define all four global worlds following the schema shown in Figure 4.15 . Accordingly, each global world consists of three parts: one solely regarding the output language A, one solely regarding the input language B, and one regarding both.

4.4.1 Global References

The last of these deals with global references. It consists of a transition system $\mathsf{T}^{AB}_{\mathsf{ref}}$, a configuration relation $\mathsf{C}^{AB}_{\mathrm{ref}}$ and a reference query handler rqh^{AB}, the implementation of $\Omega^{AB}.\mathsf{Q}.\mathsf{rqh}$. These are defined in Figure 4.16 and closely follow the construction of W_{ref} in PBs: states are partial bijections on locations that can grow over time,

and the configuration relation $\mathsf{C}_{\text{ref}}^{AB}$ ensures that these locations are allocated in the heaps and store related values. Reference queries are resolved by rqh^{AB} via simple membership tests on the state.

$\mathsf{T}_{\mathsf{ref}}^{AB}$ comes in a typed and an untyped version (recall that the untyped one is obtained by erasing the type-related components and conditions, which are highlighted in color). For $\Omega_{\mathcal{TS}}$ and $\Omega_{\mathcal{IS}}$, we pick the typed $\mathsf{T}_{\mathsf{ref}}^{\mathcal{TS}}$ and $\mathsf{T}_{\mathsf{ref}}^{\mathcal{IS}}$, respectively, since they involve the typed source language. For $\Omega_{\mathcal{TI}}$ and $\Omega_{\mathcal{II}}$, we pick the untyped $\mathsf{T}_{\mathsf{ref}}^{\mathcal{TI}}$ and $\mathsf{T}_{\mathsf{ref}}^{\mathcal{II}}$, respectively. (This is reflected in the concrete definitions of the different C_{ref} implementations, namely in the quantification over elements of s_{ref}). We pick the versions of rqh^{AB} analogously.

4.4.2 Unary Parts

The unary part of Ω^{AB} concerning language A consists of a transition system T^A, query handlers vqh^A and cqh^A (implementing $\Omega^{AB}.\mathsf{vqh}_{\mathsf{a}}$ and $\Omega^{AB}.\mathsf{cqh}_{\mathsf{a}}$, respectively), and a configuration predicate P^A. We will discuss these in a moment. The unary part concerning language B is analogous. The overall configuration relation $\Omega^{AB}.\mathsf{C}$ then is the separating conjunction of three parts: $\mathsf{C}_{\text{ref}}^{AB}$; the lifting of P^A by pairing each element with an empty B configuration; and the analogous lifting of P^B. It remains to define T^L, vqh^L and cqh^L, for $L \in \{\mathcal{S}, \mathcal{I}, \mathcal{T}\}$.

4.4.2.1 Global World Components Regarding $\mathcal{S}$

Figure 4.17 shows these constructions for the source language $\mathcal{S}$. States are the runtime environments used by the operational semantics (mapping labels to values, see Section 4.2.3). It makes sense that the global world "owns" the environment because it is shared by all modules constituting the program. In reflectance of the semantics, the transition relation for these states is trivial: environments are not allowed to change in any way, not even to grow.

The value and configuration query handlers are purely syntactic (they do not care about the state) and mostly straightforward. For instance, value v represents a pair of v_1 and v_2 iff it is *the* pair value $\langle v_1, v_2 \rangle$. Note the following points:

- There are no restrictions on inhabitants of type names (we allow the full set of values).
- Similarly for goodfix and goodgen, which only matter for the intermediate language.
- We find it convenient to restrict the configurations accepted by $\mathsf{cqh}_{\mathcal{S}}$ to cores (Section 4.2.2). We will do so also for the other languages.
- Our characterization of application (and instantiation) is actually "one step ahead": what we characterize is not the beta redex but rather the result of performing the beta reduction. This is done for consistency with how we set things up for the machine language in Section 4.4.2.3 below.

$$
\begin{array}{lcl}
\mathsf{T}^{AB}_{\mathsf{ref}}.\mathsf{S} & := & \{\, s_{\mathsf{ref}} \in \mathcal{P}_{\mathrm{fin}}(\mathsf{CTy} \times \mathsf{Loc}_A \times \mathsf{Loc}_B) \mid \forall(\tau, l_1, l_2) \in s_{\mathsf{ref}}.\ \forall(\tau', l_1', l_2') \in s_{\mathsf{ref}}. \\
 & & \quad (l_1 = l_1' \implies \tau = \tau' \wedge l_2 = l_2') \wedge (l_2 = l_2' \implies \tau = \tau' \wedge l_1 = l_1') \,\} \\
\mathsf{T}^{AB}_{\mathsf{ref}}.\sqsubseteq & := & \subseteq \\
\mathsf{T}^{AB}_{\mathsf{ref}}.\sqsubseteq_{\mathrm{pub}} & := & \subseteq
\end{array}
$$

$$
\mathsf{rqh}^{AB}(s_{\mathsf{ref}})(\tau) := \{\, (l_{\mathsf{a}}, l_{\mathsf{b}}) \mid (\tau, l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\}
$$

$$
\mathsf{C}^{AB}_{\mathsf{ref}}(G)(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}) := C^{AB}_{\mathsf{ref}}(s_{\mathsf{ref}})(\overline{G}(s_{\mathsf{a}}, s_{\mathsf{ref}}, s_{\mathsf{b}}))
$$

$$
\begin{array}{lcl}
C^{\mathcal{TS}}_{\mathsf{ref}}(s_{\mathsf{ref}})(R) & := & \{\, ((\emptyset, \emptyset, \emptyset, h_{\mathsf{a}}), (h_{\mathsf{b}}, \emptyset, \emptyset)) \mid h_{\mathsf{a}} \neq \bot \wedge h_{\mathsf{b}} \neq \bot \wedge \\
 & & \quad \mathrm{dom}(h_{\mathsf{a}}) = \{\, l_{\mathsf{a}} \mid \exists \tau, l_{\mathsf{b}}.\ (\tau, l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\} \wedge \\
 & & \quad \mathrm{dom}(h_{\mathsf{b}}) = \{\, l_{\mathsf{b}} \mid \exists \tau, l_{\mathsf{a}}.\ (\tau, l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\} \wedge \\
 & & \quad \forall(\tau, l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}}.\ (h_{\mathsf{a}}(l_{\mathsf{a}}), h_{\mathsf{b}}(l_{\mathsf{b}})) \in R(s)(\tau) \,\}
\end{array}
$$

$$
\begin{array}{lcl}
C^{\mathcal{TI}}_{\mathsf{ref}}(s_{\mathsf{ref}})(R) & := & \{\, ((\emptyset, \emptyset, \emptyset, h_{\mathsf{a}}), (h_{\mathsf{b}}, \emptyset)) \mid h_{\mathsf{a}} \neq \bot \wedge h_{\mathsf{b}} \neq \bot \wedge \\
 & & \quad \mathrm{dom}(h_{\mathsf{a}}) = \{\, l_{\mathsf{a}} \mid \exists l_{\mathsf{b}}.\ (l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\} \wedge \\
 & & \quad \mathrm{dom}(h_{\mathsf{b}}) = \{\, l_{\mathsf{b}} \mid \exists l_{\mathsf{a}}.\ (l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\} \wedge \\
 & & \quad \forall(l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}}.\ (h_{\mathsf{a}}(l_{\mathsf{a}}), h_{\mathsf{b}}(l_{\mathsf{b}})) \in R(s) \,\}
\end{array}
$$

$$
\begin{array}{lcl}
C^{\mathcal{II}}_{\mathsf{ref}}(s_{\mathsf{ref}})(R) & := & \{\, ((h_{\mathsf{a}}, \emptyset), (h_{\mathsf{b}}, \emptyset)) \mid h_{\mathsf{a}} \neq \bot \wedge h_{\mathsf{b}} \neq \bot \wedge \\
 & & \quad \mathrm{dom}(h_{\mathsf{a}}) = \{\, l_{\mathsf{a}} \mid \exists l_{\mathsf{b}}.\ (l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\} \wedge \\
 & & \quad \mathrm{dom}(h_{\mathsf{b}}) = \{\, l_{\mathsf{b}} \mid \exists l_{\mathsf{a}}.\ (l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\} \wedge \\
 & & \quad \forall(l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}}.\ (h_{\mathsf{a}}(l_{\mathsf{a}}), h_{\mathsf{b}}(l_{\mathsf{b}})) \in R(s) \,\}
\end{array}
$$

$$
\begin{array}{lcl}
C^{\mathcal{IS}}_{\mathsf{ref}}(s_{\mathsf{ref}})(R) & := & \{\, ((h_{\mathsf{a}}, \emptyset), (h_{\mathsf{b}}, \emptyset, \emptyset)) \mid h_{\mathsf{a}} \neq \bot \wedge h_{\mathsf{b}} \neq \bot \wedge \\
 & & \quad \mathrm{dom}(h_{\mathsf{a}}) = \{\, l_{\mathsf{a}} \mid \exists \tau, l_{\mathsf{b}}.\ (\tau, l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\} \wedge \\
 & & \quad \mathrm{dom}(h_{\mathsf{b}}) = \{\, l_{\mathsf{b}} \mid \exists \tau, l_{\mathsf{a}}.\ (\tau, l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}} \,\} \wedge \\
 & & \quad \forall(\tau, l_{\mathsf{a}}, l_{\mathsf{b}}) \in s_{\mathsf{ref}}.\ (h_{\mathsf{a}}(l_{\mathsf{a}}), h_{\mathsf{b}}(l_{\mathsf{b}})) \in R(s)(\tau) \,\}
\end{array}
$$

Figure 4.16: Global World Components Related to Global References.

$$
\begin{array}{lcl}
\mathsf{T}^{\mathcal{S}}.\mathsf{S} & := & \mathsf{Env} \\
\mathsf{T}^{\mathcal{S}}.\sqsubseteq & := & (=) \\
\mathsf{T}^{\mathcal{S}}.\sqsubseteq_{\mathrm{pub}} & := & (=)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{unit}) & := & \{\, \langle\rangle \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{nat}\,\mathbf{v}) & := & \{\, n \mid n = \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{pair}\,\mathbf{v}\,\mathbf{v}') & := & \{\, \langle \mathbf{v}, \mathbf{v}' \rangle \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{inl}\,\mathbf{v}) & := & \{\, \mathsf{inl}\ \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{inr}\,\mathbf{v}) & := & \{\, \mathsf{inr}\ \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{pack}\,\mathbf{v}) & := & \{\, \mathsf{pack}\ \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{roll}\,\mathbf{v}) & := & \{\, \mathsf{roll}\ \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{fix}) & := & \{\, \mathsf{fix}\, f(x).\, e \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{gen}) & := & \{\, \Lambda.\, e \,\} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{name}) & := & \mathbf{Val} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{goodfix}) & := & \mathbf{Val} \\
\mathsf{vqh}_{\mathcal{S}}(\_)(\mathsf{goodgen}) & := & \mathbf{Val}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{cqh}_{\mathcal{S}}(\_)(\mathsf{app}\,\mathbf{v}\,\mathbf{v}'\,\mathbf{k}) & := & \{\, (\emptyset, \emptyset, \mathbf{k}[e[\mathbf{v}/f][\mathbf{v}'/x]]) \mid \mathbf{v} = \mathsf{fix}\, f(x).\, e \,\} \\
\mathsf{cqh}_{\mathcal{S}}(\_)(\mathsf{inst}\,\mathbf{v}\,\mathbf{k}) & := & \{\, (\emptyset, \emptyset, \mathbf{k}[e]) \mid \mathbf{v} = \Lambda.\, e \,\} \\
\mathsf{cqh}_{\mathcal{S}}(\_)(\mathsf{ret}\,\mathbf{v}\,\mathbf{k}) & := & \{\, (\emptyset, \emptyset, \mathbf{k}[\mathbf{v}]) \,\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{P}^{\mathcal{S}}(s) & \in & \mathcal{P}(\mathbf{Conf}) \\
\mathsf{P}^{\mathcal{S}}(s) & := & \{\, (\emptyset, s, \emptyset) \,\}
\end{array}
$$

Figure 4.17: Global World Components Concerning Language $\mathcal{S}$.

$$
\begin{array}{lcl}
\mathsf{T}^{\mathcal{I}}.\mathsf{S} & := & \mathit{1} \\
\mathsf{T}^{\mathcal{I}}.\sqsubseteq & := & \mathit{1} \times \mathit{1} \\
\mathsf{T}^{\mathcal{I}}.\sqsubseteq_{\mathrm{pub}} & := & \mathit{1} \times \mathit{1}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{unit}) & := & \{\, \langle\rangle \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{nat}\,\mathbf{v}) & := & \{\, n \mid n = \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{pair}\,\mathbf{v}\,\mathbf{v}') & := & \{\, \langle \mathbf{v}, \mathbf{v}' \rangle \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{inl}\,\mathbf{v}) & := & \{\, \mathsf{inl}\ \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{inr}\,\mathbf{v}) & := & \{\, \mathsf{inr}\ \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{pack}\,\mathbf{v}) & := & \{\, \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{roll}\,\mathbf{v}) & := & \{\, \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{fix}) & := & \{\, \langle \sigma; \mathsf{fix}\, f(y,k).\, e \rangle \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{gen}) & := & \{\, \langle \sigma; \Lambda.\, e \rangle \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{name}) & := & \{\, \langle \sigma; n \rangle \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{goodfix}) & := & \{\, \langle \sigma; \mathsf{fix}\, f(y,k).\, e \rangle \mid \forall n.\ e \neq \mathsf{halt}\, n \,\} \\
\mathsf{vqh}_{\mathcal{I}}(\_)(\mathsf{goodgen}) & := & \{\, \langle \sigma; \Lambda k.\, e \rangle \mid \forall n.\ e \neq \mathsf{halt}\, n \,\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{cqh}_{\mathcal{I}}(\_)(\mathsf{app}\,\mathbf{v}\,\mathbf{v}'\,\mathbf{k}) & := & \{\, (\emptyset, (\sigma[f{\mapsto}\mathbf{v}, y{\mapsto}\mathbf{v}', k{\mapsto}\mathbf{k}], e)) \mid \mathbf{v} = \langle \sigma; \mathsf{fix}\, f(y,k).\, e \rangle \,\} \\
\mathsf{cqh}_{\mathcal{I}}(\_)(\mathsf{inst}\,\mathbf{v}\,\mathbf{k}) & := & \{\, (\emptyset, (\sigma[k{\mapsto}\mathbf{k}], e)) \mid \mathbf{v} = \langle \sigma; \Lambda k.\, e \rangle \,\} \\
\mathsf{cqh}_{\mathcal{I}}(\_)(\mathsf{ret}\,\mathbf{v}\,\mathbf{k}) & := & \{\, (\emptyset, (\sigma, k\ x)) \mid \sigma(k) = \mathbf{k} \wedge \sigma(x) = \mathbf{v} \,\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{P}^{\mathcal{I}}(s) & \in & \mathcal{P}(\mathbf{Conf}) \\
\mathsf{P}^{\mathcal{I}}(s) & := & \{\, \emptyset \,\}
\end{array}
$$

Figure 4.18: Global World Components Concerning Language $\mathcal{I}$.

The associated configuration predicate $\mathsf{cpred}^{\mathcal{S}}$ simply requires that the configuration consists solely of the state's environment.

4.4.2.2 Global World Components Regarding $\mathcal{I}$

Figure 4.18 shows the constructions for the intermediate language $\mathcal{I}$. In constrast to $\mathcal{S}$, the run-time environment in $\mathcal{I}$ is not shared by all modules in a program but is local to a computation. In fact, each function value, being a closure, comes with its own environment. Therefore, there is no need to keep track of anything purely $\mathcal{I}$-specific in the global world and we can pick the trivial transition system (with a singleton state space).

Like for the source language, the value and configuration query handlers are purely syntacic and mostly straightforward. Note the following points:

- Being untyped, $\mathcal{I}$ does not provide a "pack" or "roll" construct and thus there are no "packed" or "rolled" values. We decide to represent such $\mathcal{S}$ values the

same as the underlying values. This decision is arbitrary but natural.

- We restrict inhabitants of type names to be numbers.
- We require that the body of "good" functions is not a $\mathsf{halt}\, n$ expression. We will use such "bad" functions when crafting intermediate values in the proof of transitivity.

4.4.2.3 Global World Components Regarding $\mathcal{T}$

The constructions for the machine language $\mathcal{T}$, shown in Figure 4.19, are more interesting. Recall that $\mathcal{T}$ configurations consist of (possibly) a program counter, (possibly) a register file, a stack and a heap. We set things up such that the global world always contributes the register file, the free part of the stack, and the part of the heap used for allocating values. Formally, the state space $\mathsf{T}^{\mathcal{T}}.\mathsf{S}$ ranges over pairs of register file and *value registry*. Let us first focus on the register file.

Registers. The predicate $\mathsf{P}^{\mathcal{T}}$ only allows configurations whose register file is precisely the one in the current state. The full transition relation allows the contents of the register file to change arbitrarily, because any code may write to a register. The public transition relation, however, requires that the callee-saved registers (*i.e.,* sp and env) retain their value. This corresponds to what the calling convention prescribes for the end-to-end behavior of functions.

Stack. Recall that sp is intended to function as the stack pointer. Hence the stack portion above and including address $R(\mathsf{sp})$ is considered free. Accordingly, $\mathsf{P}^{\mathcal{T}}$ only allows configurations whose stack is exactly this infinite, cofinite, and continuous chunk (its contents do not matter). This formally ties the stack to register sp. Moreover, it rules out that any particular module puts constraints on that part. Note that this, together with the public transition relation, implies that PILS do not expect functions to preserve the unused part of the stack.

Value registry. The value registry's purpose is to keep track of allocated values, primarily such that the value query handler can do its job. As the figure shows, it is a mapping from *value descriptions* to sets of values, and $\mathsf{vqh}_{\mathcal{T}}$ consults this mapping to find out which pairs, sums, and functions are currently known. Other value forms don't require an entry in the registry (for instance, numbers are unboxed and simply represent themselves). We collapse fix-points and generalizations as they are represented the same way in $\mathcal{T}$.

This representation, and that of the other value forms, manifests itself in the configuration predicate $\mathsf{cpred}^{\mathcal{T}}$, which connects the registry to the memory (and thus to reality). It dictates that in order for a value v to represent a pair of v_1 and v_2 ($v \in s.\rho(\mathsf{pair}_{\mathcal{T}}\, v_1\, v_2)$), v must be a heap pointer to v_1, with v_2 being stored in the adjacent cell. Similarly, sum values are represented by a heap pointer to a tag (zero for

$$
\begin{array}{lcl}
d \in \mathsf{ValDescr} & ::= & \mathsf{pair}_{\mathcal{T}}\,\mathbf{v}\,\mathbf{v}' \mid \mathsf{inl}_{\mathcal{T}}\,\mathbf{v} \mid \mathsf{inr}_{\mathcal{T}}\,\mathbf{v} \mid \mathsf{fun}_{\mathcal{T}}\,\mathbf{v} \qquad (\mathbf{v} \in \mathbf{Val}) \\
\rho \in \mathsf{Registry} & := & \mathsf{ValDescr} \to \mathcal{P}(\mathbf{Val})
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{T}^{\mathcal{T}}.\mathsf{S} & := & \mathsf{RegFile} \times \mathsf{Registry} \\
\mathsf{T}^{\mathcal{T}}.\sqsubseteq & := & \{\, ((R,\kappa),(R',\kappa')) \mid \kappa \subseteq \kappa' \,\} \\
\mathsf{T}^{\mathcal{T}}.\sqsubseteq_{\mathrm{pub}} & := & \{\, ((R,\kappa),(R',\kappa')) \mid \kappa \subseteq \kappa' \wedge R(\mathsf{sp}) = R'(\mathsf{sp}) \wedge R(\mathsf{env}) = R'(\mathsf{env}) \,\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{vqh}_{\mathcal{T}}(\_)(\mathsf{unit}) & := & \mathbf{Val} \\
\mathsf{vqh}_{\mathcal{T}}(\_)(\mathsf{nat}\,\mathbf{v}) & := & \{\, \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{T}}(s)(\mathsf{pair}\,\mathbf{v}\,\mathbf{v}') & := & s.\rho(\mathsf{pair}_{\mathcal{T}}\,\mathbf{v}\,\mathbf{v}') \\
\mathsf{vqh}_{\mathcal{T}}(s)(\mathsf{inl}\,\mathbf{v}) & := & s.\rho(\mathsf{inl}_{\mathcal{T}}\,\mathbf{v}) \\
\mathsf{vqh}_{\mathcal{T}}(s)(\mathsf{inr}\,\mathbf{v}) & := & s.\rho(\mathsf{inr}_{\mathcal{T}}\,\mathbf{v}) \\
\mathsf{vqh}_{\mathcal{T}}(\_)(\mathsf{roll}\,\mathbf{v}) & := & \{\, \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{T}}(\_)(\mathsf{pack}\,\mathbf{v}) & := & \{\, \mathbf{v} \,\} \\
\mathsf{vqh}_{\mathcal{T}}(s)(\mathsf{fix}) & := & \{\, \mathbf{v} \mid \exists \mathbf{v}'.\ \mathbf{v} \in s.\rho(\mathsf{fun}_{\mathcal{T}}\,\mathbf{v}') \,\} \\
\mathsf{vqh}_{\mathcal{T}}(s)(\mathsf{gen}) & := & \{\, \mathbf{v} \mid \exists \mathbf{v}'.\ \mathbf{v} \in s.\rho(\mathsf{fun}_{\mathcal{T}}\,\mathbf{v}') \,\} \\
\mathsf{vqh}_{\mathcal{T}}(\_)(\mathsf{name}) & := & \mathbf{Val} \\
\mathsf{vqh}_{\mathcal{T}}(\_)(\mathsf{goodfix}) & := & \mathbf{Val} \\
\mathsf{vqh}_{\mathcal{T}}(\_)(\mathsf{goodgen}) & := & \mathbf{Val}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{cqh}_{\mathcal{T}}(s)(\mathsf{app}\,\mathbf{v}\,\mathbf{v}'\,\mathbf{k}) & := & \{\, (n,\emptyset,\emptyset,\emptyset) \mid \mathbf{v} = s.R(\mathsf{clo}) \wedge \mathbf{v}' = s.R(\mathsf{arg}) \wedge \\
 & & \qquad \mathbf{k} = s.R(\mathsf{ret}) \wedge \mathbf{v} \in s.\rho(\mathsf{fun}_{\mathcal{T}}\,n) \,\} \\
\mathsf{cqh}_{\mathcal{T}}(s)(\mathsf{inst}\,\mathbf{v}\,\mathbf{k}) & := & \{\, (n,\emptyset,\emptyset,\emptyset) \mid \mathbf{v} = s.R(\mathsf{clo}) \wedge \\
 & & \qquad \mathbf{k} = s.R(\mathsf{ret}) \wedge n \in s.\rho(\mathsf{fun}\,\mathbf{v}) \,\} \\
\mathsf{cqh}_{\mathcal{T}}(s)(\mathsf{ret}\,\mathbf{v}\,\mathbf{k}) & := & \{\, (\mathbf{k},\emptyset,\emptyset,\emptyset) \mid \mathbf{v} = s.R(\mathsf{arg}) \,\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{P}^{\mathcal{T}} \in \mathsf{T}^{\mathcal{T}}.\mathsf{S} \to \mathcal{P}(\mathbf{Conf}) \\
\mathsf{P}^{\mathcal{T}}(R,\kappa) := \{\, (\emptyset,R,q,h) \mid q \neq \bot \wedge \mathrm{dom}(q) = \{n \mid n \geq R(\mathsf{sp})\} \wedge \forall \mathbf{v},\mathbf{v}',\mathbf{v}''. \\
\qquad (\mathbf{v} \in \kappa(\mathsf{pair}_{\mathcal{T}}\,\mathbf{v}'\,\mathbf{v}'') \implies h(\mathbf{v}) = \mathbf{v}' \wedge h(\mathbf{v}+1) = \mathbf{v}'') \wedge \\
\qquad (\mathbf{v} \in \kappa(\mathsf{inl}_{\mathcal{T}}\,\mathbf{v}') \implies h(\mathbf{v}) = 0 \wedge h(\mathbf{v}+1) = \mathbf{v}') \wedge \\
\qquad (\mathbf{v} \in \kappa(\mathsf{inr}_{\mathcal{T}}\,\mathbf{v}') \implies \exists n \neq 0.\ h(\mathbf{v}) = n \wedge h(\mathbf{v}+1) = \mathbf{v}') \wedge \\
\qquad (\mathbf{v} \in \kappa(\mathsf{fun}_{\mathcal{T}}\,\mathbf{v}') \implies h(\mathbf{v}) = \mathbf{v}') \,\}
\end{array}
$$

Figure 4.19: Global World Components Concerning Language $\mathcal{T}$.

"left", non-zero for "right"), with the data being in the adjacent cell. Function values are just heap addresses (intuitively: pointers to closures). The contents stored at such an address is the code pointer, which is part of the value description: $s.\rho(\mathsf{fun}_{\mathcal{T}}\,\mathbf{v}')$ is the set of functions with code pointer $\mathbf{v}'$. From the point of view of the global world, only this heap cell matters. Code and environment, possibly stored in the consecutive cells, are subject to a local world.

Both public and full transition relation allow the registry to grow over time, but forbid any other modification. This ensures that any registered value stays valid forever, and never changes intrinsically[6]. Of course, the (required) monotonicity of $\mathsf{vqh}_{\mathcal{T}}$ relies on this.

Configuration queries. The registry also plays a role in $\mathsf{cqh}_{\mathcal{T}}$. How should the app and inst queries be implemented for $\mathcal{T}$, *i.e.,* what shall count as function calls? Following Hur and Dreyer [32], we choose to be very liberal in what we accept here. For instance, we do not want to require that the program counter points to a particular "call" instruction (in fact, we don't even assume such an instruction in our language). To be as flexible as possible, we define an application as the (relevant part of the) machine state *after* the function call happened. That is, we require that the program counter points to the first instruction of the function's code and that the appropriate registers are filled with return address and argument value (in the case of a fix-point application), as dictated by the calling convention. We also require that the function is registered as such. This is important because otherwise, when reasoning about a function call, one may not know whether reading the code pointer (before jumping) succeeds.

The $\mathsf{ret}\,\mathbf{v}\,\mathbf{k}$ query is easier to answer: the configuration's program counter must be exactly the continuation $\mathbf{k}$ and register arg must contain value $\mathbf{v}$.

4.5 Simulations

The definitions of the various PILS simulation relations and auxiliary constructions are shown in Figures 4.20 to 4.22. (As before, the parts unique to the typed model are highlighted.) Let us first give a brief overview.

Most notions and concepts from PBs still exist in PILS. For some of the constructions, the main difference in generalizing to the inter-language setting is simply that they are now abstracted over the details of the particular languages being related, with the help of language specification and worlds. That said, there are significant changes as well. Recall that (i) we want PILS to be asymmetric (because so is refinement), (ii) we are getting rid of an explicit local knowledge component, (iii) we want to avoid having two sources of coinduction (as were present in PBs), and (iv) we both construct a typed and an untyped version.

[6] Benign changes to the concrete representation are allowed. For instance, the tag of an "inr" value may change from 1 to 2, because both are non-zero.

$$\mathrm{ES}^W_{A,B} := W.\mathsf{O} \to A.\mathbf{Cont} \times B.\mathbf{Cont} \to (W.\mathsf{S} \to \mathsf{VRelF}_{A,B}) \to W.\mathsf{S} \to W.\mathsf{S} \to (A.\mathbf{Conf} \times B.\mathbf{Conf})_\bot \to \mathsf{ERel}_{A,B}$$
$$\mathrm{KS}^W_{A,B} := W.\mathsf{O} \to A.\mathbf{Cont} \times B.\mathbf{Cont} \to (W.\mathsf{S} \to \mathsf{VRelF}_{A,B}) \to W.\mathsf{S} \to W.\mathsf{S} \to \mathsf{KRel}_{A,B}$$

$$\mathbf{E}_W \in \mathrm{ES}^W_{A,B}$$
$$\mathbf{E}_W(i)(\mathbf{k}^0_\mathsf{a}, \mathbf{k}^0_\mathsf{b})(G)(s_0)(s)(\phi)(\tau) = \{\, (e_\mathsf{a}, e_\mathsf{b}) \mid G \in \mathbf{GK}_W \implies$$
$$\forall c_\mathsf{a}, c_\mathsf{b}.\ \forall \eta_\mathsf{a} \in \mathbf{frame}.\ \forall \eta_\mathsf{b} \in \mathbf{frame}.$$
$$\forall (m_\mathsf{a}, m_\mathsf{b}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s)(\phi)(e_\mathsf{a}, e_\mathsf{b})(c_\mathsf{a}, c_\mathsf{b})(\eta_\mathsf{a}, \eta_\mathsf{b}).$$
$$(\textsc{err})\ \ \exists m'_\mathsf{b}.\ m_\mathsf{b} \overset{\iota}{\hookrightarrow}{}^* m'_\mathsf{b} \wedge m'_\mathsf{b} \in \mathbf{error}$$
$$\vee\ (\textsc{ret})\ \ \exists m'_\mathsf{b}, s', \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}, e'_\mathsf{a}, e'_\mathsf{b}, c'_\mathsf{a}, c'_\mathsf{b}.\ m_\mathsf{b} \overset{\iota}{\hookrightarrow}{}^* m'_\mathsf{b} \wedge s' \sqsupseteq s \wedge s' \sqsupseteq_{\mathrm{pub}} s_0 \wedge$$
$$(m_\mathsf{a}, m'_\mathsf{b}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s')(\bot)(e'_\mathsf{a}, e'_\mathsf{b})(c'_\mathsf{a}, c'_\mathsf{b})(\eta_\mathsf{a}, \eta_\mathsf{b}) \wedge (\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in \overline{G}(s')(\tau) \wedge$$
$$(e'_\mathsf{a}, e'_\mathsf{b}) \in W.\mathsf{cqh}_\mathsf{a}(s')(\mathsf{ret}\, \mathbf{v}_\mathsf{a}\, \mathbf{k}^0_\mathsf{a}) \times W.\mathsf{cqh}_\mathsf{b}(s')(\mathsf{ret}\, \mathbf{v}_\mathsf{b}\, \mathbf{k}^0_\mathsf{b})$$
$$\vee\ (\textsc{step})\ m_\mathsf{a} \notin \mathbf{halted} \wedge \forall t, m'_\mathsf{a}.\ m_\mathsf{a} \overset{t}{\hookrightarrow} m'_\mathsf{a} \implies \exists i', e'_\mathsf{a}, e'_\mathsf{b}, c'_\mathsf{a}, c'_\mathsf{b}, m'_\mathsf{b}, m''_\mathsf{b}, s', \phi'.$$
$$m_\mathsf{b} \overset{\iota}{\hookrightarrow}{}^* m'_\mathsf{b} \wedge s' \sqsupseteq s \wedge (m'_\mathsf{a}, m''_\mathsf{b}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s')(\phi')(e'_\mathsf{a}, e'_\mathsf{b})(c'_\mathsf{a}, c'_\mathsf{b})(\eta_\mathsf{a}, \eta_\mathsf{b}) \wedge$$
$$(\textsc{rec})\ \ (m'_\mathsf{b} \overset{t}{\hookrightarrow} m''_\mathsf{b} \vee m'_\mathsf{b} = m''_\mathsf{b} \wedge t = \iota \wedge i' < i) \wedge \phi' \neq \bot \wedge$$
$$(e'_\mathsf{a}, e'_\mathsf{b}) \in \mathbf{E}_W(i')(\mathbf{k}^0_\mathsf{a}, \mathbf{k}^0_\mathsf{b})(G)(s_0)(s')(\phi')(\tau)$$
$$\vee\ (\textsc{call})\ m'_\mathsf{b} \overset{t}{\hookrightarrow} m''_\mathsf{b} \wedge \phi' = \bot \wedge$$
$$(e'_\mathsf{a}, e'_\mathsf{b}) \in \mathbf{U}(s')(G(s'))(G(s'))(\mathbf{K}_W(i')(\mathbf{k}^0_\mathsf{a}, \mathbf{k}^0_\mathsf{b})(G)(s_0)(s'))(\tau)\, \}$$

$$\mathbf{K}_W \in \mathrm{KS}^W_{A,B}$$
$$\mathbf{K}_W(i)(\mathbf{k}^0_\mathsf{a}, \mathbf{k}^0_\mathsf{b})(G)(s_0)(s)(\tau')(\tau) = \{\, (\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b}) \mid \forall G' \supseteq G.\ \forall s' \sqsupseteq_{\mathrm{pub}} s.\ \forall (v_\mathsf{a}, v_\mathsf{b}) \in \overline{G'}(s')(\tau').$$
$$W.\mathsf{cqh}_\mathsf{a}(s')(\mathsf{ret}\, v_\mathsf{a}\, \mathbf{k}_\mathsf{a}) \times W.\mathsf{cqh}_\mathsf{b}(s')(\mathsf{ret}\, v_\mathsf{b}\, \mathbf{k}_\mathsf{b}) \subseteq \mathbf{E}_W(i)(\mathbf{k}^0_\mathsf{a}, \mathbf{k}^0_\mathsf{b})(G')(s_0)(s')(\bot)(\tau)\, \}$$

$$\mathbf{F}_W \in (W.\mathsf{S} \to \mathsf{VRelF}_{A,B}) \to W.\mathsf{S} \to \mathsf{VRelF}_{A,B}$$
$$\mathbf{F}_W(G)(s)(\tau) := \{\, (\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in \mathsf{GoodFuns}(s)(\tau) \mid \exists i.\ \forall \mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b}.\ \forall G' \supseteq G.\ \forall s' \sqsupseteq s.$$
$$\mathbf{U}(s')(sng(\tau)(\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}))(G'(s'))(sng(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})) \subseteq \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G')(s')(s')(\bot)\, \}$$

$$\mathbf{GK}_W \in \mathcal{P}(W.\mathsf{S} \to \mathsf{VRelF}_{A,B})$$
$$\mathbf{GK}_W := \{\, G \mid \mathbf{F}_W(G) \subseteq G \wedge (\forall s, s'.\ s' \sqsupseteq s \implies G(s') \supseteq G(s)) \wedge$$
$$\forall s.\ \forall \mathbf{n} \in W.\mathsf{NS}.\ G(s)(\mathbf{n}) = W.\mathsf{NR}(G)(s)(\mathbf{n})\, \}$$

$$\Gamma \vdash M_\mathsf{a} \precsim_\Omega M_\mathsf{b} : \Gamma' :=$$
$$\forall \mathcal{N} \in \mathcal{P}(\mathsf{TyNam}).\ \mathcal{N}\ \text{countably infinite} \implies \exists w \in \mathrm{LWorld}^{\Omega.\mathsf{T}}_{A,B}.$$
$$\forall \gamma_\mathsf{a}, \gamma_\mathsf{b}.\ \mathrm{dom}(\gamma_\mathsf{a}) = \mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma_\mathsf{b}) \implies \forall \Psi_\mathsf{a}, \Psi_\mathsf{b}.$$
$$\forall (h^g_\mathsf{a}, h^l_\mathsf{a}) \in \mathbf{cload}(M_\mathsf{a})(\Psi_\mathsf{a})(\gamma_\mathsf{a}).\ \forall (h^g_\mathsf{b}, h^l_\mathsf{b}) \in \mathbf{cload}(M_\mathsf{b})(\Psi_\mathsf{b})(\gamma_\mathsf{b}).\ \exists s_0.$$
$$w.\mathsf{NS} \subseteq \mathcal{N} \wedge$$
$$w \in \mathsf{stable}(\Omega) \wedge$$
$$(\forall G \in \mathbf{GK}_{w\uparrow}.\ (h^g_\mathsf{a}, h^g_\mathsf{b}) \in \Omega.\mathbf{C}(G(-, snd\, s_0))(fst\, s_0)) \wedge$$
$$(\forall G \in \mathbf{GK}_{w\uparrow}.\ (h^l_\mathsf{a}, h^l_\mathsf{b}) \in w.\mathbf{C}(G)(s_0)) \wedge$$
$$\Omega.\mathsf{rqh}(fst\, s_0) = \emptyset \wedge$$
$$\forall f' {:}\tau' \in \Gamma'.\ \exists (v_\mathsf{a}, v_\mathsf{b}) \in \mathbf{vload}(M_\mathsf{a})(\Psi_\mathsf{a})(\gamma_\mathsf{a})(f') \times \mathbf{vload}(M_\mathsf{b})(\Psi_\mathsf{b})(\gamma_\mathsf{b})(f').$$
$$\forall s \sqsupseteq s_0.\ \forall G \in \mathbf{GK}_{w\uparrow}.\ (s \in \mathsf{sat}(G) \wedge \forall f {:}\tau \in \Gamma.\ (\gamma_\mathsf{a} f, \gamma_\mathsf{b} f) \in \overline{G}(s)(\tau)) \implies$$
$$(v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s)(\tau')$$

Figure 4.20: Key components of PILS

At the top-level, PILS define *module similarity*, written $\Gamma \vdash M_\mathsf{a} \precsim_\Omega M_\mathsf{b} : \Gamma'$. This basically generalizes PBs open program equivalence (here we consider modules because they are the objects that we compile and link). This relation is relative to a global world $\Omega \in \mathrm{GWorld}_{A,B}$ for the language pair A, B, with A intuitively being the language *to* which is being translated and B the one *from* which is being translated. Γ and Γ' list the imported and exported labels, respectively (each label is paired with its type in the typed version). At the end, we instantiate this relation with four global worlds in order to obtain the four models of interest: we write $\precsim_{\mathcal{TS}}$ for the typed model $\precsim_{\Omega_{\mathcal{TS}}}$, we write $\precsim_{\mathcal{TI}}$ for the untyped model $\precsim_{\Omega_{\mathcal{TI}}}$, and so on.

The E relation. The main ingredient of the top-level relation is **E**, the generalization of PBs's expression equivalence, which retains the familiar distinction of three cases.

In our generic setting there is no notion of expressions—only configurations—but we find it helpful to still refer to the configurations related by **E** as such. Indeed, as we will see, for the source language $\mathcal{S}$ these configurations will usually just be expressions (empty heap and no environment). For $\mathcal{T}$, on the other hand, they will usually just be program counters (empty heap, empty stack, no register file). These configurations will be completed with the help of configurations provided by the world, as discussed in a moment.

Let us come to the formal definition of **E**, shown in Figure 4.20. At a high level, it relates two "expressions" e_a (the "target" of a transformation) and e_b (the "source" of a transformation) iff one of three cases holds:

(ERR) e_b can silently (*i.e.,* without I/O) produce an error.

(RET) e_a is finished, and e_b can silently finish returning a related value.

(STEP) e_a can take a step and e_b can match it (perhaps after some internal computation). "Match" means that both steps produce the same event and that the remaining computations either (REC) are again related by **E**, or (CALL) are about to call related "external" functions, *i.e.,* functions related by the global knowledge G. This "external call" case is the characteristic feature of PBs, as discussed at length in Chapter 3. In the concrete PILS instances we care about, the event t is in the CALL case guaranteed to always be ι.

We now go into details.

Asymmetric small-step formulation. In contrast to the symmetric big-step formulation of **E** in PBs, PILS employ an asymmetric small-step formulation (ASF). Notice how **E**'s STEP case asks us to consider each possible step of the "target" program e_a in turn (when using REC repeatedly), each time asking us to match it with steps of the "source" program e_b. Besides being seemingly necessary to properly deal with events (here: I/O), such an ASF is also important in the context of compiler

verification because it gives the compiler the flexibility to remove erroneous behaviors of the source program and resolve some of its nondeterminism.

In a naive ASF, each step of the target must be matched with one or more steps of the source, thus forcing the source to take at least as many steps as the target. Of course, this is overly restrictive and our actual definition allows *stuttering*, *i.e.,* it allows the source to not take any steps at all. In order for this to be sound, however, some other form of progress must occur; otherwise one could trivially show any divergent target program related to any source program. We follow the standard approach to stuttering by indexing the $\mathbf{E}$ relation with an element i of a well-founded partially ordered set $(W.\mathsf{O})$. We postpone a discussion of the details to Section 4.5.1.

Notice that the check that $m_\mathsf{a} \notin \textbf{halted}$, *i.e.,* that m_a can take a step, is necessary to keep the STEP case from being trivially satisfied whenever the program cannot continue.

From configurations to machines and back. In order to talk about the execution of e_a and e_b, we first need to "complete" these configurations and convert them into physical machines. These completions should not be completely arbitrary; they should adhere to the world's constraints at the current state s. Hence we first quantify over c_a and c_b, representing the portion of the machine state constrained by the world W (an implicit argument), and require that they are indeed related by $W.\mathbf{C}(G)(s)$. This is done with the help of the auxiliary construct cfg from Figure 4.21. Its ϕ argument, which is also an argument to $\mathbf{E}$, plays an important role, but we postpone its explanation. For now let us assume that it is always $\bot$ (even in $\mathbf{E}$'s REC case), so that we don't need to worry about the MORE disjunct in cfg.

Then, given such c_a and c_b, we attach these to e_a and e_b, together with arbitrary frame configurations η_a and η_b representing the rest of the running program state. Finally, we only consider machines m_a and m_b that realize these composed configurations[7].

The two other occurrences of cfg, namely in STEP and RET, are proof obligations. For instance, STEP requires us to show that, after the step(s), each resulting machine can again be decomposed into a (possibly new) expression e'_a, a (possibly new) configuration c'_a, and the *original* frame configuration η_a (similarly for the B-side), since we should not have touched the frame's private memory. Moreover, c'_a and c'_b must again satisfy W's constraints, but we may advance s to a future state s' in order to achieve that.

Configuration queries. In the RET case of $\mathbf{E}$, we generalize PB's "terminating with a value" to "returning a value to the initial continuation", where the initial continuations $\mathbf{k}^0_a, \mathbf{k}^0_b$ are given as additional arguments to $\mathbf{E}$. As expected, we do so by appeal to the world's configuration query handlers $W.\mathsf{cqh}_\mathsf{a}$ and $W.\mathsf{cqh}_\mathsf{b}$. Similarly,

[7]Note that, in our three concrete languages, a machine realizing a configuration is identical to the configuration (modulo the obvious embedding).

$$\mathsf{cfg} \in \mathsf{CR}^T_{A,B} \to (T.\mathsf{S} \to \mathsf{VRelF}_{A,B}) \to T.\mathsf{S} \to (A.\mathbf{Cont} \times B.\mathbf{Cont})_\bot \to$$
$$A.\mathbf{Conf} \times B.\mathbf{Conf} \to A.\mathbf{Conf} \times B.\mathbf{Conf} \to A.\mathbf{Conf} \times B.\mathbf{Conf} \to \mathsf{MRel}_{A,B}$$
$$\mathsf{cfg}(C)(G)(s)(\phi)(e_\mathsf{a}, e_\mathsf{b})(c_\mathsf{a}, c_\mathsf{b})(\eta_\mathsf{a}, \eta_\mathsf{b}) = \{\, (m_\mathsf{a}, m_\mathsf{b}) \mid$$
$$(\textsc{core})\ \phi = \bot \wedge m_\mathsf{a} \in \mathbf{real}(e_\mathsf{a} \cdot c_\mathsf{a} \cdot \eta_\mathsf{a}) \wedge m_\mathsf{b} \in \mathbf{real}(e_\mathsf{b} \cdot c_\mathsf{b} \cdot \eta_\mathsf{b}) \wedge$$
$$e_\mathsf{a} \in \mathbf{core} \wedge e_\mathsf{b} \in \mathbf{core} \wedge (c_\mathsf{a}, c_\mathsf{b}) \in C(G)(s)$$
$$\vee\ (\textsc{more})\ \phi = (\eta_\mathsf{a}, \eta_\mathsf{b}) \wedge m_\mathsf{a} \in \mathbf{real}(e_\mathsf{a}) \wedge m_\mathsf{b} \in \mathbf{real}(e_\mathsf{b}) \wedge$$
$$c_\mathsf{a} = \emptyset \wedge c_\mathsf{b} = \emptyset \,\}$$

$$\mathbf{U} \in W.\mathsf{S} \to \mathsf{VRelF}_{A,B} \to \mathsf{VRelF}_{A,B} \to \mathsf{KRel}_{A,B} \to \mathsf{CTy} \to \mathsf{CRel}_{A,B}$$
$$\mathbf{U}(s)(R_f)(R_v)(R_k)(\tau) := \{\, (e_\mathsf{a}, e_\mathsf{b}) \mid$$
$$(\textsc{app})\ \exists \tau_v, \tau_r.\ \exists (\mathbf{k}_a, \mathbf{k}_b) \in R_k(\tau_r)(\tau).$$
$$\exists (f_\mathsf{a}, f_\mathsf{b}) \in \langle\!\langle R_f \rangle\!\rangle^s(\tau_v \to \tau_r).\ \exists (v_\mathsf{a}, v_\mathsf{b}) \in \langle\!\langle R_v \rangle\!\rangle^s(\tau_v).$$
$$(e_\mathsf{a}, e_\mathsf{b}) \in W.\mathsf{cqh}_\mathsf{a}(s)(\mathsf{app}\ f_\mathsf{a}\ v_\mathsf{a}\ \mathbf{k}_\mathsf{a}) \times W.\mathsf{cqh}_\mathsf{b}(s)(\mathsf{app}\ f_\mathsf{b}\ v_\mathsf{b}\ \mathbf{k}_\mathsf{b})$$
$$\vee\ (\textsc{inst})\ \exists \alpha, \tau_r, \tau_i.\ \exists (\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b}) \in R_k(\tau_r[\tau_i/\alpha])(\tau).$$
$$\exists (f_\mathsf{a}, f_\mathsf{b}) \in \langle\!\langle R_f \rangle\!\rangle^s(\forall \alpha.\, \tau_r).$$
$$(e_\mathsf{a}, e_\mathsf{b}) \in W.\mathsf{cqh}_\mathsf{a}(s)(\mathsf{inst}\ f_\mathsf{a}\ \mathbf{k}_\mathsf{a}) \times W.\mathsf{cqh}_\mathsf{b}(s)(\mathsf{inst}\ f_\mathsf{b}\ \mathbf{k}_\mathsf{b}) \,\}$$

$$\mathsf{sat} \in (w\!\uparrow\!.\mathsf{S} \to \mathsf{VRelF}_{A,B}) \to \mathcal{P}(w\!\uparrow\!.\mathsf{S})$$
$$\mathsf{sat}(G) := \{\, s \mid \exists (c_\mathsf{a}, c_\mathsf{b}) \in w.\mathsf{C}(G)(s) \,\}$$

$$\mathsf{stable}(\Omega) \in \mathcal{P}(\mathrm{LWorld}^\Omega_{A,B})$$
$$\mathsf{stable}(\Omega) := \{\, w \mid \forall G, s_\mathrm{g}, s, s'_\mathrm{g}, c_\mathsf{a}, c_\mathsf{b}, e_\mathsf{a}, e_\mathsf{b}, \eta_\mathsf{a}, \eta_\mathsf{b}, c'_\mathsf{a}, c'_\mathsf{b}, m_\mathsf{a}, m_\mathsf{b}.$$
$$G \in \mathbf{GK}_{w\uparrow} \wedge (c_\mathsf{a}, c_\mathsf{b}) \in w.\mathbf{C}(G)(s_\mathrm{g}, s) \wedge s'_\mathrm{g} \sqsupseteq s_\mathrm{g} \wedge$$
$$(m_\mathsf{a}, m_\mathsf{b}) \in \mathsf{cfg}(\Omega.\mathsf{C})(G(-, s))(s_\mathrm{g})(e_\mathsf{a}, e_\mathsf{b})(c'_\mathsf{a}, c'_\mathsf{b})(c_\mathsf{a} \cdot \eta_\mathsf{a}, c_\mathsf{b} \cdot \eta_\mathsf{b}) \implies$$
$$\exists s' \sqsupseteq_\mathrm{pub} s.\ (c_\mathsf{a}, c_\mathsf{b}) \in w.\mathbf{C}(G)(s'_\mathrm{g}, s') \,\}$$

$$\mathsf{GoodFuns} \in W.\mathsf{S} \to \mathsf{VRelF}_{A,B}$$
$$\mathsf{GoodFuns}(s)(\tau) = \{\, (\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \mid$$
$$(\textsc{fix})\ \exists \tau_v, \tau_r.\ \tau = \tau_v \to \tau_r \wedge (\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in W.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{goodfix}) \times W.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{goodfix})$$
$$\vee\ (\textsc{gen})\ \exists \alpha, \tau_r.\ \tau = \forall \alpha.\, \tau_r \wedge (\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in W.\mathsf{vqh}_\mathsf{a}(s)(\mathsf{goodgen}) \times W.\mathsf{vqh}_\mathsf{b}(s)(\mathsf{goodgen}) \,\}$$

Figure 4.21: Key components of PILS (continued)

in the CALL case, we use queries to check if the configurations in question represent function calls (see the definition of **U** in Figure 4.21).

Notice that **E**'s argument type τ (in the typed version) is the type of the values being returned to the initial continuations.

The K relation. Like for PBs, the continuation relation **K**—referenced in **E**'s RET case—is itself defined straightforwardly in terms of **E**, but is now phrased with the help two return queries to express the use of the two continuations language-independently.

The F relation. Recall from Section 4.3 that, in PILS, we want to always work with what we knew in PBs as the greatest consistent local knowledge, but we do not want to have it as an explicit component of the world, because that would be pointless. We achieve this as follows. We define once and for all the *function similarity* **F**, which can be thought of as the greatest consistent local knowledge but is not part of the world. This relation is easy to define on top of **E**, as shown in Figure 4.20. It roughly says that future uses of the functions with arguments related by the global knowledge and with arbitrary initial continuations must be related by **E**. The second argument to **U**, $sng(\tau)(\mathbf{v_a}, \mathbf{v_b})$, is a shortcut for the singleton value relation $\lambda\tau'.\ \{\,(\mathbf{v_a}, \mathbf{v_b}) \mid \tau' = \tau\,\}$. The last argument, $sng(\mathbf{k}_a, \mathbf{k}_b)$, stands for the continuation relation $\lambda\tau, \tau'.\ \{\,(\mathbf{k_a}, \mathbf{k_b}) \mid \tau = \tau'\,\}$, whose untyped version is a singleton and whose typed version relates the two continuations whenever input type and output type coincide. Also note how we "remember" the initial continuations by passing them on to **E**.

How does **F** tie in with the other relations? Note that **E** is defined with the help of **GK**, the set of valid global knowledges. In PBs, this set in turn was defined to contain a global knowledge G iff G included the world's local knowledge ($G \supseteq W.\mathsf{L}(G)$ in the simple version from Section 3.5). In PILS, we are forced to define it in terms of **F** instead, as is shown in the figure. Hence, instead of having a local knowledge component in worlds, in PILS we have an explicit function relation **F**. Moreover, **F**, **GK**, and **E** are all mutually recursive. This leads to some significant differences in the metatheory and the coinductive reasoning in PILS, as we will discover in Section 4.7.

Guardedness. Guardedness ensures the soundness of coinductive reasoning. PBs enforced guardedness via their consistency condition, by requiring that we beta-reduce the function applications before showing them related by **E** (Section 3.5). In PILS, this might naturally be done in an analogous way via the definition of **F**. However, PILS actually work a little differently, for the following reason: for our machine language, we implement the configuration query such that function calls are actually configurations whose program counter already points to the first instruction of the function's code. For uniformity and to avoid mismatches in proofs, we do the same for the other languages as well. Now, requiring such function calls to take a

step before showing them related by **E** would be somewhat restrictive as it would rule out functions that don't actually do anything (*e.g.,* $\mathsf{fix}\, f(x).\, 42$ in $\mathcal{S}$).

Therefore, things are slightly different in PILS. We do not require any step-taking in **F**, as evident from the formal definition. So, when showing functions like the above related by **F**, one can jump right to the RETURN case in **E**. However, when attempting to appeal to **E**'s CALL case, we are forced to take a step on both sides. This formulation actually fits very naturally with the asymmetric small-step formulation of **E**. Contrast this with PB's symmetric big-step formulation (Figure 3.15 in Section 3.6.5), where in the CALL case both reduction sequences can be empty.

Module similarity. At the bottom of Figure 4.20 we finally define module similarity. Simply put, this says that there exists a local world such that if the imported values are related, then so are the exported values. The details are more complicated because the exports may make sense only in the proper context.

More formally, we are first given a countably infinite set of type names $\mathcal{N}$, just like in PBs. We then require that there exists a stable local world w (potentially dependent on the global world Ω), whose type names are covered by $\mathcal{N}$. Next, given global and local configurations obtained from loading the modules, there must exist an initial state s_0 of the full world such that these configurations are related at s_0 by $\Omega.\mathsf{C}$ and $w.\mathsf{C}$, respectively. (We also require that the initial state does not relate any global references; this is a technical condition that simplifies the proof of modularity.) Finally, for any exported label, the two modules must provide values that are related at any satisfiable future state and global knowledge at which the imported values (in γ_a and γ_b) are also related. Satisfiable here means that there are some configurations related by w at this state; it is a technical condition later used in the proof of transitivity.

4.5.1 Stuttering according to algebraic well-founded orders.

The **E** relation allows stuttering: in the STEP case, when we are given a silent step of the target program e_a, we are *not* forced to take a step in the source program e_b. Instead, following a standard technique [75], we can decrease i, which is an element of a well-founded set[8] and was given to **E** as an argument. The well-foundedness guarantees that we can procrastinate only finitely many times, thus avoiding bogus "proofs" where we stutter ad inifinitum and could thereby trivially relate any divergent target program to any source program.

Let us look at the details. Whenever we need to show two functions related by **F**, we get to choose an i for that particular (sub–)proof (cf. Figure 4.20). This i must be an element of the well-founded set $W.\mathsf{O}$. In **E**'s STEP/REC case, we can avoid taking a step in e_b by providing a smaller i', meaning $i' < i$ according to the $W.\mathsf{O}$'s well-founded order. Then we can continue our proof with respect to i' (notice how it is passed to the recursive occurrence of **E**). It thus makes sense to think of this

[8] A *well-founded set* X consists of a carrier set $X.\mathsf{C}$ and a well-founded order $(X.<) \subset X \times X$.

$$
\begin{array}{l}
\mathsf{AWFO} := \{\, (\mathsf{C} \in \mathrm{Set},\ (<) \in \mathcal{P}(\mathsf{C} \times \mathsf{C}),\ 0 \in \mathsf{C},\ 1 \in \mathsf{C},\ (+) \in \mathsf{C} \to \mathsf{C} \to \mathsf{C}) \mid \\
\quad (<) \text{ transitive and well-founded} \wedge \\
\quad (\forall i \in \mathsf{C}.\ i + 0 = i = 0 + i) \wedge \\
\quad (\forall i, j, k \in \mathsf{C}.\ i + (j + k) = (i + j) + k) \wedge \\
\quad (\forall i, i', j \in \mathsf{C}.\ i < i' \implies i + j < i' + j) \wedge \\
\quad (\forall i, j, j' \in \mathsf{C}.\ j < j' \implies i + j < i + j') \wedge \\
\quad (\forall i.\ 0 \leq i) \wedge \\
\quad 0 \neq 1 \,\}
\end{array}
$$

Symmetric product: $(\times) \in \mathsf{AWFO} \to \mathsf{AWFO} \to \mathsf{AWFO}$

$$
\begin{array}{lcl}
(X \times Y).\mathsf{C} & := & X.\mathsf{C} \times Y.\mathsf{C} \\
(X \times Y).0 & := & (0, 0) \\
(X \times Y).1 & := & (1, 1) \\
(i, j) < (i', j') & := & (i < i' \wedge j \leq j') \vee (i \leq i' \wedge j < j') \\
(i, j) + (i', j') & := & (i + i', j + j')
\end{array}
$$

Lexicographic product: $(\#) \in \mathsf{AWFO} \to \mathsf{AWFO} \to \mathsf{AWFO}$

$$
\begin{array}{lcl}
(X \,\#\, Y).\mathsf{C} & := & X.\mathsf{C} \times Y.\mathsf{C} \\
(X \,\#\, Y).0 & := & \langle 0, 0 \rangle \\
(X \,\#\, Y).1 & := & \langle 1, 1 \rangle \\
\langle i, j \rangle < \langle i', j' \rangle & := & i < i' \vee (i = i' \wedge j < j') \\
\langle i, j \rangle + \langle i', j' \rangle & := & \langle i + i', j + j' \rangle
\end{array}
$$

Figure 4.22: Algebraically well-founded sets.

argument as a credit or budget. Actually, we get to pick a new i' even if we don't stutter, but then i' does not need to be smaller than i (in fact, there is no restriction on i' at all). Hence as soon as we take a proper step on the source side after stuttering for a while, we are once again eligible for stuttering a finite number of times.

Since i's universe is a component of the world, we can choose it as we like when we construct the world. Proofs of, say, different program transformations can therefore employ different metrics.

As Figure 4.22 shows, we actually require a little extra monoidal structure of $W.\mathsf{O}$, in addition to it being well-founded. It must also provide an order-respecting "addition" with unit 0 and its elements must be non-negative, *e.g.,* such that $i \leq i+j$ and $j \leq i+j$ for any i, j. This is important for composing programs and their proofs—in particular the PILS version of the external call lemma (see Section 4.7.1). We also require that the carrier be non-trivial, *i.e.,* contain at least another element (here written 1) in addition to 0. This allows us to embed the natural numbers, which is sometimes convenient. We call such monoids *algebraically well-founded sets.*

The figure also shows two product constructions (of two algebraically well-founded sets). The symmetric product is used in the definition of the product of two local

worlds that we have seen earlier in Figure 4.14. The lexicographic product will be used in the proof of transitivity in Section 4.8.

In order to make life easier for the user, we provide an embedding of any ordinary well-founded order $X \in \mathsf{WFO}$ into a particular algebraically well-founded order, namely into gwf $\in \mathsf{AWFO}$:

Definition 23.

$$\begin{array}{lcl} \mathit{gwf} \in \mathsf{AWFO} & & \\ \mathit{gwf}.\mathsf{C} & := & \mathit{List}(\{(X,x) \in \mathsf{WFO} \times X.\mathsf{C}\}) \\ \mathit{gwf}.\mathsf{0} & := & \epsilon \\ \mathit{gwf}.\mathsf{1} & := & (\mathbb{N}_<, 42) \\ i + j & := & i, j \end{array}$$

$$\frac{i < j}{i < (X,x), j} \qquad \frac{i < j}{(X,x), i < (X,x), j} \qquad \frac{x < x'}{(X,x), i < (X,x'), i}$$

The elements of *gwf* are heterogeneous lists of elements from arbitrary well-founded orders (formally, each list component is a pair of a well-founded order X and an element $x \in X.\mathsf{C}$ thereof). Addition is defined as concatenation and the empty list is the neutral element. The $\mathsf{1}$ element is arbitrarily chosen to be a natural number. The order $\mathit{gwf}.<$ is defined as the least fixed point of the stated rules.

An arbitrary well-founded set X can be embedded in *gwf* in the sense of the lemma below. Therefore, instead of worrying about how to make her well-founded set of choice have the required extra structure, a user may simply pick *gwf* as her local world's $w.\mathsf{O}$.

Lemma 42. Given a well-founded set $X \in \mathsf{WFO}$ and elements $x, x' \in X.\mathsf{C}$, we have:

$$x < x' \iff (X,x) < (X,x')$$

In the Coq formalization, we do not require $<$ to be transitive. Instead, we simply use its transitive closure in **E**, as well as in the conclusions (but not premises) of the three order-specific axioms. This is equivalent but slightly more convenient for the user who has to define the order and prove these conditions.

4.5.2 The two modes of E and cfg.

Recall the ϕ argument of **E** and cfg, which we haven't explained yet. In the discussion of cfg above, we assumed for simplicity that ϕ's value is always $\bot$. Indeed, it is $\bot$ initially, *i.e.,* when showing two function calls related (cf. the definition of **F**). However, it may actually change from $\bot$ to a pair $(\eta_\mathsf{a}, \eta_\mathsf{b})$ of frame configurations—and back and forth—depending on how the proof proceeds. Whenever it changes, the meanings of **E** and/or cfg change with it, so we speak of two different modes in which these construct can function, controlled by the mode argument ϕ.

The purpose of the ϕ argument is to indicate *internal* steps of computation . If it were always $\bot$, then essentially **E** would treat each step as if it might result in

control being passed to the environment. Concretely, after *each* step of the target program and matching steps of the source program, we would be obliged to show that the memory constraints currently imposed by the world are again met. And then, in reasoning about the next step of the target program, we would be forced to quantify over completely new configurations yet again.

Although sound, this is of course unnecessarily strict. Intuitively, we don't need to show that the world's conditions are satisfied again until the point where we pass control to the environment; similarly, we don't need to quantify over new configurations except at points where control is passed to *us*, because there is no way that the memory could have changed in between the internal steps of our local computation.

PILS facilitate this with the help of ϕ as follows. Initially, we have cores $e_\mathsf{a}, e_\mathsf{b}$ and ϕ is $\bot$ (*e.g.,* as set up by $\mathbf{F}$). We call this the *default mode*. Inside $\mathbf{E}$, ϕ is used to compose machines via cfg. That is, we complete the cores with world-related configurations $c_\mathsf{a}, c_\mathsf{b}$ and arbitrary frame configurations $\eta_\mathsf{a}, \eta_\mathsf{b}$ to machines $m_\mathsf{a}, m_\mathsf{b}$, as dictated by the CORE clause in cfg. Now, suppose we use $\mathbf{E}$'s REC case to take a target step to m'_a and possibly some source steps to m''_b. STEP asks us to find ϕ' and s' and $e'_\mathsf{a}, e'_\mathsf{b}$ and $c'_\mathsf{a}, c'_\mathsf{b}$ such that

$$(m'_\mathsf{a}, m''_\mathsf{b}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s')(\phi')(e'_\mathsf{a}, e'_\mathsf{b})(c'_\mathsf{a}, c'_\mathsf{b})(\eta_\mathsf{a}, \eta_\mathsf{b})$$

and REC requires that $e'_\mathsf{a}, e'_\mathsf{b}$ are related by $\mathbf{E}$ for ϕ'. Moreover, REC insists that $\phi' \neq \bot$, *i.e.,* it initiates a mode change. Note that by definition of cfg (the MORE clause), ϕ' must thus necessarily be "chosen" to be $(\eta_\mathsf{a}, \eta_\mathsf{b})$. What implications does this have?

First, in order to show that $m'_\mathsf{a}, m''_\mathsf{b}$ are related as stated above, we are *not* required to show that the world is currently satisfied. Second, cfg now enforces that we pick $e'_\mathsf{a}, e'_\mathsf{b}$ such that they actually account for the whole machines $m'_\mathsf{a}, m'_\mathsf{b}$, *i.e.,* the configurations that we need to show related recursively for ϕ' are no longer cores but essentially are the whole machines, including the frame configurations. We thus call this the *fixed mode*.

Third, when we continue the reasoning by setting out to show $e'_\mathsf{a}, e'_\mathsf{b}$ related in fixed mode (for ϕ'), the new machines that we are given by cfg's MORE do not comprise any new parts—they simply realize e'_a and e'_b, respectively.

Assume that we repeat this process for a few times—essentially just taking a few of steps with the target machine and a few steps with the source machine—and then wish to appeal to one of the other $\mathbf{E}$ cases. In RET, we are forced to switch back to the default mode (cfg is used with the $\bot$ argument) and thus finally have to get back in synch with the world. That is, we must find a proper future state and decompositions of the machines that satisfy the world at that state and also contain the original pristine frame configurations (this makes sense because control moves on to code that may expect the world's constraints to be satisfied and its local memory to be uncompromised). We know what the original frame configurations are because we stored them in the mode argument (when entering fixed mode earlier) and repeatedly passed on this argument since then.

The same happens in the CALL case. Here, of course, we must continue reasoning about the continuations in which the calls are made. Like **F**, **K** is defined in terms of **E**'s default mode. This crucially forces the proof to consider fresh completing configurations in a future state (in which the functions potentially return).

4.5.3 A Note on the Untyped Model

Since our source programs are typesafe and therefore "don't go wrong", neither will correctly produced $\mathcal{I}$ or $\mathcal{T}$ programs. One may thus wonder why our model takes faulty programs into account (the ERR case in **E**). This feature is actually crucial for verifying transformations in the untyped version of the model. (Recall that we obtain this version by erasing all the type arguments, which are highlighted, from the definitions in the relevant figures.)

To see this, first consider the following optimization at the *source* level (where x is a variable of type nat):

$$\mathsf{fix}\, f(x).\ \mathsf{ifnz}\ x\ \mathsf{then}\ e\ \mathsf{else}\ e \quad \rightsquigarrow \quad \mathsf{fix}\, f(x).\, e$$

In the process of showing that $\mathsf{fix}\, f(x).\, e$ is similar to the original function, we will be given arguments related at some unknown relation G and state s, $(\mathsf{nat}, v_a, v_b) \in \overline{G}(s)$. Now, by inverting the definition of the value closure, we learn that $v_a = v_b = n$, for some number n.

Let us ignore the remaining proof steps and instead consider this transformation at the IL level, where we would be working in the untyped version of the model. There, we will still be given related arguments, $(v_a, v_b) \in \overline{G}(s)$, but this time the type information is missing. Consequently, when inverting the value closure, we don't end up with the single case above (where $v_a = v_b = n$), but must also consider all the other cases such as v_a and v_b being pairs. Now, note two important points: First, the global world $\Omega_{\mathcal{I}\mathcal{I}}$ ensures that whenever v_b is a number, then so is v_a. In that case we can proceed as we would above in the typed model. Second, if v_b is *not* a number, then the original program produces an error and, thanks to ERR, there is nothing more to show.

4.5.4 Convenience Lemmas

Reasoning directly using the definition of **E** can be tedious. Here we present a few lemmas that aim at making proofs more convenient and cleaner by providing higher-level reasoning principles. These lemmas (and variants thereof) are used extensively throughout our Coq development. In particular when reasoning about $\mathcal{T}$ code, they have become indispensable. To better grasp the meaning of these lemmas, it often helps to read them bottom-up, a la "in order to show ..., it suffices to show ...".

The first two express the relationship between **E**'s default mode and its fixed mode. Most remaining lemmas are stated in terms of the fixed mode, so we typically start out our reasoning by applying Lemma 43, which acquires world-related configurations and frames.

Lemma 43 (Acquire)**.** If

$$\forall (c_\mathsf{a}, c_\mathsf{b}) \in W.\mathsf{C}(G)(s).\ \forall (\eta_\mathsf{a}, \eta_\mathsf{b}) \in \mathbf{frame} \times \mathbf{frame}.\\ (e_\mathsf{a} \cdot c_\mathsf{a} \cdot \eta_\mathsf{a}, e_\mathsf{b} \cdot c_\mathsf{b} \cdot \eta_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$$

then $(e_\mathsf{a}, e_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\bot)(\tau)$.

Lemma 44 (Release)**.** If

- $(e_\mathsf{a}, e_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\bot)(\tau)$
- $(c_\mathsf{a}, c_\mathsf{b}) \in W.\mathsf{C}(G)(s)$
- $e_\mathsf{a} \in \mathbf{core} \wedge e_\mathsf{b} \in \mathbf{core}$

then $(e_\mathsf{a} \cdot c_\mathsf{a} \cdot \eta_\mathsf{a}, e_\mathsf{b} \cdot c_\mathsf{b} \cdot \eta_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$.

Lemma 45 states that in fixed mode it is always safe to assume the configurations to be realizable.

Lemma 45. If

$$(\exists (m_\mathsf{a}, m_\mathsf{b}) \in \mathbf{real}(c_\mathsf{a}) \times \mathbf{real}(c_\mathsf{b})) \implies\\ (c_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$$

then $(c_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$.

Similarly, Lemma 46 allows us to assume that the global knowledge is valid (this holds even in default mode).

Lemma 46. If

$$G \in \mathbf{GK}_W \implies (c_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$$

then $(c_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$.

Lemma 47 lets us take several (silent) steps in the source program. If this results in an error, we are done immediately. Otherwise it suffices to show the original target program related to the new source program for budget j. Since we take at least one step, we may choose j arbitrarily (there is no point in allowing zero steps in this lemma).

Lemma 47 (Stepping Source)**.** If

$$\forall m_\mathsf{b} \in \mathbf{real}(c_\mathsf{b}).\ \exists m'_\mathsf{b}.\ m_\mathsf{b} \overset{\iota}{\hookrightarrow}^{+} m'_\mathsf{b} \wedge\\ (m'_\mathsf{b} \in \mathbf{error} \vee\\ \exists c'_\mathsf{b}.\ m'_\mathsf{b} \in \mathbf{real}(c'_\mathsf{b}) \wedge (c_\mathsf{a}, c'_\mathsf{b}) \in \mathbf{E}_W(j)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau))$$

then $(c_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$.

How about taking steps in the target program? Recall that **E** asks us to consider a single target step at a time and that such a formulation is generally necessary because of possible non-determinism in the target language. However, reasoning in such a way can be extremely tedious since low-level programs are typically very long. Moreover, usually a single source step translates into many target steps, so for most of the target steps one would simply stutter on the source side. Often one actually knows exactly how the given target program will execute. In these cases, instead of being given one target step after the other, each time arguing that it must be a particular step, one would like to just provide a step sequence of the target program and then continue reasoning with its result.

Fortunately, we can prove a lemma that allows exactly this. Of course, it applies only when such reasoning is sound, *i.e.,* when the target execution in question is guaranteed to be deterministic. The lemma thus rests on the idea of lowering determinism from being a property of a language to being a property of a machine and execution.

Definition 24 (Locally deterministic machines)**.** A machine m is *locally deterministic*, written $m \in \mathsf{LDet}$, iff

$$\begin{aligned}&\forall c, t, m', c'.\\ &\quad m \in \mathbf{real}(c) \wedge m \overset{t}{\hookrightarrow} m' \wedge m' \in \mathbf{real}(c') \implies \forall m_1 \in \mathbf{real}(c).\\ &\quad m_1 \notin \mathbf{halted} \wedge \forall t_1, m_1'.\ m_1 \overset{t_1}{\hookrightarrow} m_1' \implies m_1' \in \mathbf{real}(c') \wedge t = t_1\end{aligned}$$

In words: if m realizes a configuration c and takes a step to a machine realizing c' then any other machine realizing c can also take a step. Moreover, any such step necessarily produces the same event and also results in a machine realizing c'.

Definition 25 (Deterministic step)**.** A *deterministic step* is a step of a locally deterministic machine:

$$m \overset{t}{\rightarrowtail} m' \quad := \quad m \in \mathsf{LDet} \wedge m \overset{t}{\hookrightarrow} m'$$

Obviously, in our $\mathcal{T}$ language, any step that doesn't read input (*i.e.,* $t \neq ?n$) is locally deterministic. In $\mathcal{S}$ and $\mathcal{I}$, allocations are also excluded.

Lemma 48 makes use of this notion of determinism. It allows us to take multiple (n) silent steps on the target side *if* these steps are deterministic. The "difference" between the original budget i and the new budget j (for which the new target program must be shown related to the original source program) has to be at least n. This is because we need to stutter n times in the proof. We also require n to be non-zero as the property would otherwise not hold generically—but allowing $n = 0$ would be pointless anyways.

Lemma 48 (Stepping Target)**.** If

$$\begin{aligned}&m_{\mathsf{a}} \in \mathbf{real}(c_{\mathsf{a}}) \ \wedge\ m_{\mathsf{a}} \overset{\iota\ n}{\rightarrowtail} m'_{\mathsf{a}} \ \wedge\ m'_{\mathsf{a}} \in \mathbf{real}(c'_{\mathsf{a}}) \ \wedge\\ &n > 0 \ \wedge\ j <^n i \ \wedge\\ &(c'_{\mathsf{a}}, c_{\mathsf{b}}) \in \mathbf{E}_W(j)(\mathbf{k}_{\mathsf{a}}, \mathbf{k}_{\mathsf{b}})(G)(s_0)(s)(\eta_{\mathsf{a}}, \eta_{\mathsf{b}})(\tau)\end{aligned}$$

then $(c_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$.

Proof. Since $n > 0$, we have $m_\mathsf{a} \in \mathsf{LDet}$ and $m_\mathsf{a} \hookrightarrow m''_\mathsf{a}$ with $m''_\mathsf{a} \overset{\iota}{\rightarrowtail}{}^{n-1} m'_\mathsf{a}$ for some m''_a.

We prove the goal by induction on n. After unfolding $\mathbf{E}$, we are given (via cfg) machines $m \in \mathbf{real}(c_\mathsf{a})$ and $m_\mathsf{b} \in \mathbf{real}(c_\mathsf{b})$. Since m realizes the same configuration as m_a, we can exploit m_a's determinacy: the fact that m_a is not halted implies that m is not halted either. This allows us to appeal to $\mathbf{E}$'s STEP case. So suppose $m \overset{t}{\hookrightarrow} m''$.

Case $n = 1$**:** Further exploiting the determinacy of m_a tells us that t must be ι and that $m'' \in \mathbf{real}(c'_\mathsf{a})$. By using the REC case and stuttering, it suffices to show that there is $i' < i$ such that

$$(c'_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i')(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau),$$

which is a premise of the lemma (for $i' := j$).

Case $n > 1$**:** Since m'_a is not an error (it realizes a configuration) and errors cannot be undone (as required by the language specification), m''_a cannot be an error either. Hence there is c''_a such that $m''_\mathsf{a} \in \mathbf{real}(c''_\mathsf{a})$. Further exploiting the determinacy of m_a thus tells us that t must be ι and that $m'' \in \mathbf{real}(c''_\mathsf{a})$ as well. By using the REC case and stuttering, it suffices to show that there is $i' < i$ such that

$$(c''_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i')(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau).$$

From $j <^n i$ we have i' with $j <^{n-1} i' < i$, so we are done by using the inductive hypothesis on $m''_\mathsf{a} \overset{\iota}{\rightarrowtail}{}^{n-1} m'_\mathsf{a}$.

□

Of course, if one wants to take steps on both sides, one can simply compose Lemma 48 with Lemma 47. In that case, Lemma 47 should be applied first, because it allows one to pick the intermediate budget such that Lemma 48's condition on it trivially holds.

Lemma 48 has a minor shortcoming, though. Suppose that after taking some steps, we want to finish the proof of relatedness by appeal to $\mathbf{E}$'s CALL case. Because the premise is stated in terms of $\mathbf{E}$, however, the lemma does not allow us to make the last step—the one leading to a machine representing a call—part of the deterministic execution that we provide. Instead, we are forced to provide a deterministic execution that ends one step earlier, then unfold $\mathbf{E}$ and manually take the last step. Since this can sometimes be a little tedious, we also provide the following specialized variant of the lemma, where the deterministic execution ends in the actuall call. Since the source program also must take at least one step in order for the CALL case to be applicable, we bake in part of Lemma 47 here.

Lemma 49 (Stepping to Call)**.** If

- $m_{\mathsf{a}} \in \mathbf{real}(c_{\mathsf{a}})$
- $m_{\mathsf{a}} \overset{\iota}{\rightarrowtail}^{n} m'_{\mathsf{a}}$
- $\exists j' <^{n-1} i$
- $n > 0$
- $\forall m_{\mathsf{b}} \in \mathbf{real}(c_{\mathsf{b}}).\ \exists m'_{\mathsf{b}}, e_{\mathsf{a}}, e_{\mathsf{b}}, c'_{\mathsf{a}}, c'_{\mathsf{b}}.$
  $m_{\mathsf{b}} \overset{\iota}{\hookrightarrow}^{+} m'_{\mathsf{b}} \wedge$
  $(m'_{\mathsf{a}}, m'_{\mathsf{b}}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s)(\bot)(e_{\mathsf{a}}, e_{\mathsf{b}})(c'_{\mathsf{a}}, c'_{\mathsf{b}})(\eta_{\mathsf{a}}, \eta_{\mathsf{b}}) \wedge$
  $(e_{\mathsf{a}}, e_{\mathsf{b}}) \in \mathbf{U}(s)(G(s))(G(s))(\mathbf{K}_W(j)(\mathbf{k}_{\mathsf{a}}, \mathbf{k}_{\mathsf{b}})(G)(s_0)(s))(\tau)$

then $(c_{\mathsf{a}}, c_{\mathsf{b}}) \in \mathbf{E}_W(i)(\mathbf{k}_{\mathsf{a}}, \mathbf{k}_{\mathsf{b}})(G)(s_0)(s)(\eta_{\mathsf{a}}, \eta_{\mathsf{b}})(\tau)$.

(The condition $\exists j' <^{n-1} i$ is actually only needed for the case where $m_{\mathsf{b}} \overset{\iota}{\hookrightarrow}^{+} m'_{\mathsf{b}}$ takes exactly one step. For simplicity, we require it in any case.)

What if we can't show that the resulting programs constitute calls in state s? After all, that state may be completely out of synch with the machines by now. Fortunately, we can first apply Lemma 50, which lets us advance the current state arbitrarily when in fixed mode.

Lemma 50 (Advancing the State)**.** If

$$(e_{\mathsf{a}}, e_{\mathsf{b}}) \in \mathbf{E}_W(i)(\mathbf{k}_{\mathsf{a}}, \mathbf{k}_{\mathsf{b}})(G)(s_0)(s')(\eta_{\mathsf{a}}, \eta_{\mathsf{b}})(\tau) \wedge s' \sqsupseteq s$$

then $(e_{\mathsf{a}}, e_{\mathsf{b}}) \in \mathbf{E}_W(i)(\mathbf{k}_{\mathsf{a}}, \mathbf{k}_{\mathsf{b}})(G)(s_0)(s)(\eta_{\mathsf{a}}, \eta_{\mathsf{b}})(\tau)$.

For the sake of completeness, we also state a variant of Lemma 48 for ending the proof using **E**'s RET case.

Lemma 51 (Stepping Target to Return)**.** If

- $m_{\mathsf{a}} \in \mathbf{real}(c_{\mathsf{a}})$
- $m_{\mathsf{a}} \overset{\iota}{\rightarrowtail}^{n} m'_{\mathsf{a}}$
- $s \sqsupseteq_{\mathrm{pub}} s_0$
- $\exists j <^{n} i$
- $n > 0$
- $\forall m_{\mathsf{b}} \in \mathbf{real}(c_{\mathsf{b}}).\ \exists e_{\mathsf{a}}, e_{\mathsf{b}}, c'_{\mathsf{a}}, c'_{\mathsf{b}}, \mathbf{v}_{\mathsf{a}}, \mathbf{v}_{\mathsf{b}}.$
  $(m'_{\mathsf{a}}, m_{\mathsf{b}}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s)(\bot)(e_{\mathsf{a}}, e_{\mathsf{b}})(c'_{\mathsf{a}}, c'_{\mathsf{b}})(\eta_{\mathsf{a}}, \eta_{\mathsf{b}}) \wedge$
  $(e_{\mathsf{a}}, e_{\mathsf{b}}) \in W.\mathsf{cqh}_{\mathsf{a}}(s)(\mathsf{ret}\ \mathbf{v}_{\mathsf{a}}\ \mathbf{k}_{\mathsf{a}}) \times W.\mathsf{cqh}_{\mathsf{b}}(s)(\mathsf{ret}\ \mathbf{v}_{\mathsf{b}}\ \mathbf{k}_{\mathsf{b}}) \wedge$
  $(\mathbf{v}_{\mathsf{a}}, \mathbf{v}_{\mathsf{b}}) \in \overline{G}(s)(\tau)$

then $(c_\mathsf{a}, c_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})(\tau)$.

In combination, the lemmas shown in this section provide a powerful abstraction and in most cases keep us from having to work directly with the definition of $\mathbf{E}$. Moreover, although PILS are defined in an asymmetric small-step style, these lemmas essentially let us reason most of the time in the symmetric big-step style of PBs (cf. Section 3.6.5). We present a simple example that employs these lemmas in Section 4.6.

Remark. In all the stepping lemmas above, we only allow silent steps. In principle, we could also allow output steps in Lemma 49, such that both the target and the source execution produces the same sequence of outputs. This would require a slightly different proof (each target step that produces output must be matched immediately by the corresponding source step) but should be straightforward. Similarly, we could also have a straightforward combination of Lemmas 47 and 48 that allows output steps. For our compiler proofs, these generalizations are not important since there is basically only two places where we need to reason about an output step, namely the translation of an $\mathcal{S}$ or $\mathcal{I}$ output expression.

.

4.6 Example

As an example of using the PILS model, we now sketch the proof of a very simple example simulation between a source module $M_\mathcal{S}$ and a target module $M_\mathcal{T}$.

4.6.1 Modules

The source module is defined as follows.

$$\begin{aligned}
&\Gamma := F : \mathsf{unit} + \mathsf{nat} \to \mathsf{nat} \\
&\Gamma' := E : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\
&M_\mathcal{S} := E = \lambda x.\, F\ (\mathsf{inr}\ x.2) \\
&\Gamma \vdash M_\mathcal{S} : \Gamma'
\end{aligned}$$

It consists of (and therefore exports) a single function E. This function takes as argument a pair from which it extracts the second component. It then right-injects this value into a sum, on which it calls the imported function F, returning whatever F returns (if it does).

The target module is defined as follows.

$$
\begin{array}{l}
M_{\mathcal{T}} := g \\
g := (\{(E, 0)\}, d) \\
d := \mathbb{E}(\mathsf{ld\ aux_1\ [arg} + 1]), \\
\qquad \mathbb{E}(\mathsf{ld\ clo\ 2}), \\
\qquad \mathbb{E}(\mathsf{alloc\ arg\ clo}), \\
\qquad \mathbb{E}(\mathsf{sto\ [arg} + 0]\ \mathsf{clo}), \\
\qquad \mathbb{E}(\mathsf{sto\ [arg} + 1]\ \mathsf{aux_1}), \\
\qquad \mathbb{E}(\mathsf{lpc\ aux_1}), \\
\qquad \mathbb{E}(\mathsf{bop}\ (-)\ \mathsf{aux_1\ aux_1}\ 8), \\
\qquad \mathbb{E}(\mathsf{ld\ clo\ [aux_1} + 0]), \\
\qquad \mathbb{E}(\mathsf{jmp\ [clo} + 0]) \\
\vdash M_{\mathcal{T}} : E
\end{array}
$$

It consists of a single group, whose code pointer table in turn consists of a single entry, mapping E to offset 0 in the data block d. The code starting there first projects the second component of the argument pair and stores it in register $\mathsf{aux_1}$. Next, it allocates two heap cells for the sum value in the argument register. It writes 2 into the tag component, indicating "in right" (any non-zero value would do), and writes the contents of $\mathsf{aux_1}$ into the value component. It then calculates the address of F's entry in the label environment by getting the current program counter and subtracting 8 (there are 5 instructions before lpc, as well as the full group header of size 3, at the beginning of which F's value is stored). Finally, the code loads F's function value from there into register clo and performs an indirect jump to it (following the calling convention). Note that E does not modify registers ret and env and thus passes them on to F, such that F will return to wherever E was supposed to return to.

4.6.2 Proof

Our goal now is to show that the target module refines the source module relative to the given typing:

$$\Gamma \vdash M_{\mathcal{T}} \precsim_{\mathcal{TS}} M_{\mathcal{S}} : \Gamma'$$

The first part of this concerns the module level and is very tedious, but we spell it out here for once. The second part concerns the behavior of the actual functions.

4.6.2.1 First Part

Before we construct a suitable local world, let us look at what we know about loaded configurations

$$
\begin{array}{l}
(c^g_{\mathcal{T}}, c^l_{\mathcal{T}}) \in \mathbf{cload}(M_{\mathcal{T}})(\Psi_{\mathcal{T}})(\{F \mapsto v_{\mathcal{T}}\}) \\
(c^g_{\mathcal{S}}, c^l_{\mathcal{S}}) \in \mathbf{cload}(M_{\mathcal{S}})(\Psi_{\mathcal{S}})(\{F \mapsto v_{\mathcal{S}}\})
\end{array}
$$

as we will be given in a moment ($v_{\mathcal{T}}$ and $v_{\mathcal{S}}$ are the arbitrary values of the imported function F). By the respective implementations of **cload**, we have:

$$c^g_{\mathcal{T}} = (\emptyset, R, (\boldsymbol{\lambda}\_.0), [v_{\mathcal{T}} \mapsto w] \sqcup [\Psi_{\mathcal{T}}{+}2 \mapsto \Psi_{\mathcal{T}}{+}3])$$
$$c^l_{\mathcal{T}} = (\emptyset, \emptyset, \emptyset, [\Psi_{\mathcal{T}} \longmapsto v_{\mathcal{T}}, \Psi_{\mathcal{T}}{+}2] \sqcup [\Psi_{\mathcal{T}}{+}3 \longmapsto d])$$

$$c^g_{\mathcal{S}} = (\emptyset, \{F{\mapsto}v_{\mathcal{S}}\} \uplus M_{\mathcal{S}} \uplus \rho)$$
$$c^l_{\mathcal{S}} = \emptyset$$

The global part of $\mathcal{T}$'s heap consists of two cells. The first cell's address is F's value ($v_{\mathcal{T}}$) and its contents is an arbitrary code pointer w. The second cell's address is E's value, namely $\Psi_{\mathcal{T}}{+}2$, and its code pointer refers to the next cell. This next cell is contained in the local part of $\mathcal{T}$'s heap and marks the beginning of the data block d. This heap also contains the label environment (cells $\Psi_{\mathcal{T}}$ and $\Psi_{\mathcal{T}}{+}1$), through which the code accesses F.

For $\mathcal{S}$, things are simpler. The global configuration merely contains a label environment. It maps F to $v_{\mathcal{S}}$ and E to $\lambda x.\, F\ (\mathsf{inr}\ x.2)$—and possibly further labels to further values (in ρ) but we do not care about those.

We now construct a local world $w \in \mathrm{LWorld}^{\Omega_{\mathcal{T}\mathcal{S}}.\mathsf{T}}$ such that its configuration relation insists that $\mathcal{T}$'s heap contains the local heap shown above (in $c^l_{\mathcal{T}}$). Since that heap depends on the load address ($\Psi_{\mathcal{T}}$) and the value of F ($v_{\mathcal{T}}$), which will only be given *after* we have constructed w, we must define w's state space to range over all such possible values and then later pick the initial state to match whatever we were given. We could define states to be pairs of a load address and a value for F, but a more general pattern is to define states simply as heaps:

$$\begin{array}{lcl}
w.\mathsf{S} & := & \mathsf{Heap}_{\mathcal{T}} \\
w.\sqsupseteq & := & (=) \\
w.\sqsupseteq_{\mathrm{pub}} & := & (=) \\
w.\mathsf{O} & := & \mathbb{N}_{\mathrm{awf}} \\
w.\mathsf{C}(_)(_, s) & := & \{\, ((\emptyset, \emptyset, \emptyset, s), \emptyset) \,\} \\
w.\mathsf{NS} & := & \emptyset \\
w.\mathsf{NR}(_)(_) & := & \emptyset
\end{array}$$

As algebraic well-founded set we choose the natural numbers.

The configuration relation $w.\mathsf{C}$ does not depend on the global state (or even the global knowledge) and thus w is trivially stable. Note how $w.\mathsf{C}$ contains a single pair, where $\mathcal{T}$'s heap is exactly s.

Since the source program is so simple (*e.g.,* does not involve any memory), w's states do not need to track any $\mathcal{S}$-related information. Moreover, the target implementation of E does not involve any other local memory, so the only states we are interested in are of the form $[\Psi_{\mathcal{T}} \longmapsto v_{\mathcal{T}}, \Psi_{\mathcal{T}}{+}2] \sqcup [\Psi_{\mathcal{T}}{+}3 \longmapsto d]$ (for some $\Psi_{\mathcal{T}}$ and $v_{\mathcal{T}}$). Since the code is not self-modifying, this heap won't ever change, which is why we allow no transitions.

Having constructed w, let us now suppose we are given anchors, import values and corresponding loadings $(c^g_{\mathcal{T}}, c^l_{\mathcal{T}})$ and $(c^g_{\mathcal{S}}, c^l_{\mathcal{S}})$ as stated and destructed above. We pick the obvious initial state s_0 that matches these configurations:

$$\begin{array}{lcl} s_0 & := & ((s^\circ_{\mathcal{T}}, \emptyset, s^\circ_{\mathcal{S}}), [\Psi_{\mathcal{T}} \longmapsto v_{\mathcal{T}}, \Psi_{\mathcal{T}}{+}2] \sqcup [\Psi_{\mathcal{T}}{+}3 \longmapsto d]) \\ s^\circ_{\mathcal{T}} & := & (R, \{(\mathsf{fun}_{\mathcal{T}}\, w, v_{\mathcal{T}}), (\mathsf{fun}_{\mathcal{T}}\, (\Psi_{\mathcal{T}}{+}3), \Psi_{\mathcal{T}}{+}2)\}) \\ s^\circ_{\mathcal{S}} & := & \{F{\mapsto}v_{\mathcal{S}}\} \uplus M_{\mathcal{S}} \uplus \rho \end{array}$$

The registry in $s^\circ_{\mathcal{T}}$ contains two function entries, one for $v_{\mathcal{T}}$ with code pointer w (this is F) and one for $\Psi_{\mathcal{T}}{+}2$ with code pointer $\Psi_{\mathcal{T}}{+}3$ (this is E). It is easy to check that the following holds for any G, which concludes the first part.

$$\begin{array}{l} (c^g_{\mathcal{T}}, c^g_{\mathcal{S}}) \in \Omega_{\mathcal{TS}}.\mathsf{C}(G(-, [\Psi_{\mathcal{T}} \longmapsto v_{\mathcal{T}}, \Psi_{\mathcal{T}}{+}2] \sqcup [\Psi_{\mathcal{T}}{+}3 \longmapsto d]))(s^\circ_{\mathcal{T}}, \emptyset, s^\circ_{\mathcal{S}}) \\ (c^l_{\mathcal{T}}, c^l_{\mathcal{S}}) \in w.\mathsf{C}(G)(s_0) \end{array}$$

4.6.2.2 Second Part

It remains to reason about the exported functions. By implementation of **vload** we have:

$$\begin{array}{lcl} \Psi_{\mathcal{T}}{+}2 & \in & \mathbf{vload}(M_{\mathcal{T}})(\Psi_{\mathcal{T}})(\{F{\mapsto}v_{\mathcal{T}}\})(E) \\ \lambda x.\, F\ (\mathsf{inr}\ x.2) & \in & \mathbf{vload}(M_{\mathcal{S}})(\Psi_{\mathcal{S}})(\{F{\mapsto}v_{\mathcal{S}}\})(E) \end{array}$$

Note that we only need to consider future states of s_0. This is crucial because otherwise $\Psi_{\mathcal{T}}{+}2$ may not have the meaning that we expect it to have (it might not be registered as a function value, or its code pointer might point to arbitrary code or invalid instructions or even non-existant memory).

So suppose we are given $G \in \mathbf{GK}_{w\uparrow}$ and $s \sqsupseteq s_0$ for which the imported values are related:

$$(v_{\mathcal{T}}, v_{\mathcal{S}}) \in \overline{G}(s)(\mathsf{unit} + \mathsf{nat} \to \mathsf{nat})$$

We must show that the exported values are then related as well:

$$(\Psi_{\mathcal{T}}{+}2, \lambda x.\, F\ (\mathsf{inr}\ x.2)) \in \overline{G}(s)(\mathsf{nat} \times \mathsf{nat} \to \mathsf{nat})$$

Showing that they count as functions in s, *i.e.,* that they match the fix query, is easy to do by relying on $s \sqsupseteq s^0$, because we explicitly registered $\Psi_{\mathcal{T}}{+}2$ as a closure in $s^\circ_{\mathcal{T}}$, which is only allowed to grow.

Showing that the functions are related by $G(s)$ takes more work. Since G is a valid global knowledge, we have $\mathbf{F}_{w\uparrow}(G) \subseteq G$. So it suffices to show the functions similar according to $\mathbf{F}$ (the $\mathsf{GoodFuns}$ condition holds trivially), for which we consider their applications in a future state. Given $s' \sqsupseteq s$, $G' \supseteq G$ and initial continuations $\mathbf{k}_{\mathcal{T}}, \mathbf{k}_{\mathcal{S}}$, it suffices to show that there is an i such that

$$(e_{\mathcal{T}}, e_{\mathcal{S}}) \in \mathbf{E}_{w\uparrow}(i)(\mathbf{k}_{\mathcal{T}}, \mathbf{k}_{\mathcal{S}})(G')(s')(s')(\bot)(\mathsf{nat}).$$

Here, the configurations $e_{\mathcal{T}}$ and $e_{\mathcal{S}}$ are applications of (the values for) E to related values according to $\mathbf{U}$ (recall that these "applications" actually describe the machine *after* the jump or beta reduction):

$$\begin{array}{l} e_{\mathcal{T}} = (\Psi_{\mathcal{T}}+3, \emptyset, \emptyset, \emptyset) \\ e_{\mathcal{S}} = (\emptyset, \emptyset, \mathbf{k}_{\mathcal{S}}[F\ (\mathsf{inr}\ v'_{\mathcal{S}}.2)]) \\ s'.R(\mathsf{ret}) = \mathbf{k}_{\mathcal{T}} \\ s'.R(\mathsf{arg}) = v'_{\mathcal{T}} \\ (v'_{\mathcal{T}}, v'_{\mathcal{S}}) \in \overline{G'}(s')(\mathsf{nat} \times \mathsf{nat}) \end{array}$$

The choice of i somewhat depends on the lemmas that we want to use, so for now we will keep i abstract and pick its value at the end of the proof, when all constraints are known. (Of course we need to be careful that the choice does not depend on anything that got introduced in the meantime).

We start by applying Lemmas 43 and 45. This leaves us having to show

$$(e_{\mathcal{T}} \cdot c_{\mathcal{T}} \cdot \eta_{\mathcal{T}}, e_{\mathcal{S}} \cdot c_{\mathcal{S}} \cdot \eta_{\mathcal{S}}) \in \mathbf{E}_{w\uparrow}(i)(\mathbf{k}_{\mathcal{T}}, \mathbf{k}_{\mathcal{S}})(G')(s')(s')(\eta_{\mathcal{T}}, \eta_{\mathcal{S}})(\mathsf{nat})$$

for any $(c_{\mathcal{T}}, c_{\mathcal{S}}) \in w\uparrow.\mathsf{C}(G')(s')$ and any $\eta_{\mathcal{T}}, \eta_{\mathcal{S}}$, but we may assume that there is $m_{\mathcal{T}} \in \mathbf{real}(e_{\mathcal{T}} \cdot c_{\mathcal{T}} \cdot \eta_{\mathcal{T}})$ (we don't care about a source machine for now).

Next, we apply Lemmas 50 and 49 such that it suffices to show the existence of $n > 0$, j, s'' and $m'_{\mathcal{T}}$ satisfying the following:

1. $m_{\mathcal{T}} \overset{\iota}{\rightarrowtail}{}^{n} m'_{\mathcal{T}}$
2. $\exists j' <^{n-1} i$
3. $s'' \sqsupseteq s'$
4. $\forall m_{\mathcal{S}} \in \mathbf{real}(e_{\mathcal{S}} \cdot c_{\mathcal{S}} \cdot \eta_{\mathcal{S}}).\ \exists m'_{\mathcal{S}}, e'_{\mathcal{T}}, e'_{\mathcal{S}}, c'_{\mathcal{T}}, c'_{\mathcal{S}}.$
   $m_{\mathcal{S}} \overset{\iota}{\hookrightarrow}{}^{+} m'_{\mathcal{S}} \wedge$
   $(m'_{\mathcal{T}}, m'_{\mathcal{S}}) \in \mathsf{cfg}(w\uparrow.\mathsf{C})(G')(s'')(\bot)(e'_{\mathcal{T}}, e'_{\mathcal{S}})(c'_{\mathcal{T}}, c'_{\mathcal{S}})(\eta_{\mathcal{T}}, \eta_{\mathcal{S}}) \wedge$
   $(e'_{\mathcal{T}}, e'_{\mathcal{S}}) \in \mathbf{U}(s'')(G'(s''))(G'(s''))(\mathbf{K}_{w\uparrow}(j)(\mathbf{k}_{\mathcal{T}}, \mathbf{k}_{\mathcal{S}})(G')(s')(s''))(\mathsf{nat})$

(1) First, due to $s' \sqsupseteq s_0$ we know that the heap contains $[\Psi_{\mathcal{T}} + 3 \longmapsto d]$. This means that $m_{\mathcal{T}}$'s proram counter (which equals that of $e_{\mathcal{T}}$) currently points to the first instruction of E's code. From $(v'_{\mathcal{T}}, v'_{\mathcal{S}}) \in \overline{G'}(s')(\mathsf{nat} \times \mathsf{nat})$ we know by definition of the value closure and $\Omega_{\mathcal{TS}}$ that $v'_{\mathcal{T}}$ is registered in the global state's database of $\mathcal{T}$ values, *i.e.,* we have $v'_{\mathcal{T}} \in s'.\rho(\mathsf{pair}_{\mathcal{T}}\, v^1_{\mathcal{T}}\, v^2_{\mathcal{T}})$ for some machine values $v^1_{\mathcal{T}}, v^2_{\mathcal{T}}$. This, and the connection between s' and $m_{\mathcal{T}}$, tells us that the target heap contains $v^2_{\mathcal{T}}$ at address $v'_{\mathcal{T}} + 1$. Hence, after one (deterministic) step of $m_{\mathcal{T}}$, which performs the first ld instruction, register $\mathsf{aux_1}$ holds $v^2_{\mathcal{T}}$ (and of course the program counter has advanced to the second ld instruction)—otherwise nothing changes.

Two further steps write the constant 2 into register clo and the address of a newly allocated heap block of size 2 into register arg. The next two steps write to

this block of memory. The subsequent lpc instruction has the effect of writing $\Psi_\mathcal{T}+8$ into register $\mathsf{aux_1}$, so that the bop instruction updates $\mathsf{aux_1}$ to $\Psi_\mathcal{T}$.

In order to reason about the remaining two instructions, note that due to $s' \sqsupseteq s_0$ we know that the machine heap contains both $[\Psi_\mathcal{T} \mapsto v_\mathcal{T}]$ and $[v_\mathcal{T} \mapsto w]$. Accordingly, the next two steps write $v_\mathcal{T}$, the value of F, into register clo and set the program counter to its first instruction at address w.

All in all, we have the end-to-end deterministic execution $m_\mathcal{T} \overset{\iota}{\rightarrowtail}{}^{9} m'_\mathcal{T}$, where

$$\begin{array}{l} m'_\mathcal{T} \in \mathbf{real}(e'_\mathcal{T} \cdot c'_\mathcal{T} \cdot \eta_\mathcal{T}) \\ e'_\mathcal{T} = (w, \emptyset, \emptyset, \emptyset) \end{array}$$

and $c'_\mathcal{T}$ is obtained from $c_\mathcal{T}$ by extending its heap with the freshly allocated (and thus disjoint) chunk $[w' \longmapsto 2, v^2_\mathcal{T}]$ and changing its register file $s'.R$ to

$$s'.R[\mathsf{arg}{\mapsto}w'][\mathsf{aux_1}{\mapsto}\Psi_\mathcal{T}][\mathsf{clo}{\mapsto}v_\mathcal{T}].$$

(2) With $n = 9$, it is easy to satisfy $\exists j' <^{n-1} i$ by choosing $i = 8$ and $j' = 0$ (recall that we kept i abstract until now).

(3) We pick s'' basically such that it matches the resulting machine $m'_\mathcal{T}$ above. Concretely, we derive s'' from s' by making two changes. First, we set its R component to $s'.R[\mathsf{arg}{\mapsto}w'][\mathsf{aux_1}{\mapsto}\Psi_\mathcal{T}][\mathsf{clo}{\mapsto}v_\mathcal{T}]$. Second, we extend the value registry $s'.\rho$ with an entry saying that w' now represents $\mathsf{inr}_\mathcal{T}\, v^2_\mathcal{T}$. It is trivial to check that $s'' \sqsupseteq s'$.

(4) Let us focus on the source program. From $(v'_\mathcal{T}, v'_\mathcal{S}) \in \overline{G'}(s')(\mathsf{nat} \times \mathsf{nat})$ we know by definition of the value closure and $\Omega_{\mathcal{TS}}$ that $v'_\mathcal{S}$ equals $\langle v^1_\mathcal{S}, v^2_\mathcal{S}\rangle$ for some source values $v^1_\mathcal{S}, v^2_\mathcal{S}$ that are related to $v^1_\mathcal{T}, v^2_\mathcal{S}$ from above. In particular, we have $(v^2_\mathcal{T}, v^2_\mathcal{S}) \in \overline{G'}(s')(\mathsf{nat})$.

Moreover, from $s' \sqsupseteq s \sqsupseteq s_0$ we know that $c_\mathcal{S}$ consists of the environment $s^\circ_\mathcal{S}$, which maps F to $v_\mathcal{S}$. Since $v_\mathcal{S}$ is related by $\overline{G'}(s')$ at function type, it must have the form $\mathsf{fix}\, f(x).\, e$.

All in all, we thus easily derive $m_\mathcal{S} \hookrightarrow^+ m'_\mathcal{S}$, where:

$$\begin{array}{l} m'_\mathcal{S} \in \mathbf{real}(e'_\mathcal{S} \cdot c_\mathcal{S} \cdot \eta_\mathcal{S}) \\ e'_\mathcal{S} = \mathbf{k}_\mathcal{S}[e[v_\mathcal{S}/f][\mathsf{inr}\ v^2_\mathcal{S}/x]] \end{array}$$

Note that this represents a function call in s'' (in fact, in any state):

$$e'_\mathcal{S} \in w{\uparrow}.\mathsf{cqh_b}(s'')(\mathsf{app}\, v_\mathcal{S}\, (\mathsf{inr}\ v^2_\mathcal{S})\, \mathbf{k}_\mathcal{S})$$

It is easy to see that $e'_\mathcal{T}$ from (1) also represents a function call in s'':

$$e'_\mathcal{T} \in w{\uparrow}.\mathsf{cqh_a}(s'')(\mathsf{app}\, v_\mathcal{T}\, w'\, \mathbf{k}_\mathcal{T})$$

Moreover, using monotonicity, the functions are obviously related by $\overline{G'}(s'')$. The arguments w' and $\mathsf{inr}\ v^2_\mathcal{S}$ are as well, because they represent right-injections of $v^2_\mathcal{T}$ and $v^2_\mathcal{S}$, respectively, which are related by $\overline{G'}(s')$.

Consequently, the **U** condition in (4) is satisfied if we can show that the continuations $\mathbf{k}_\mathcal{T}$ and $\mathbf{k}_\mathcal{S}$ are related by $\mathbf{K}_{w\uparrow}(j)(\mathbf{k}_\mathcal{T}, \mathbf{k}_\mathcal{S})(G')(s')(s'')$ for an arbitrary j of our choosing. Indeed, this is an easy general lemma: the initial continuations are always related (at any public extension of the initial state).

Lemma 52.

$$\forall s \sqsupseteq_{\text{pub}} s_0.\ (\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b}) \in \mathbf{K}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(s_0)(s)(\tau)(\tau)$$

Proof. By unfolding the definitions and immediately using **E**'s RET case. □

To conlude the proof of (4)—and thus the proof of the example—it remains to establish the cfg condition, in particular that $c'_\mathcal{T}$ and $c'_\mathcal{S}$ satisfy the world. This is very easy to check.

4.7 Metatheory

4.7.1 Basics

Definition 26 (Least global knowledge)**.** For $W \in \text{World}_{A,B}$ and a function $g \in W.\mathsf{S} \to \mathsf{VRelF}_{A,B}$, we define $[W]_g \in W.\mathsf{S} \to \mathsf{VRelF}_{A,B}$ as the least global knowledge containing g. Formally it is constructed as a greatest fixed point of a monotone function such that the following holds (using a pointwise union over functions):

$$[W]_g = g \cup \mathbf{F}_W([W]_g) \cup W.\mathsf{NR}([W]_g)$$

We write $[W]$ short for $[W]_\emptyset$.

Lemma 53 (Least global knowledge)**.** For any W and g, we have $[W]_g \in \mathbf{GK}_W$ if the following hold:

1. g is monotone: $\forall s', s.\ s' \sqsupseteq s \implies g(s') \supseteq g(s)$
2. $\forall \mathbf{n} \in W.\mathsf{NS}.\ \forall s.\ g(s)(\mathbf{n}) = \emptyset$

. In particular, we always have $[W] \in \mathbf{GK}_W$.

Moreover, $[W]_g \subseteq G$ for any $G \in \mathbf{GK}_W$ with $g \subseteq G$.

Lemma 54 (Composition of **E** and **K**)**.** If

1. $(e_\mathsf{a}, e_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s)(s)(\phi)(\tau)$ and
2. $(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b}) \in \mathbf{K}_W(j)(\mathbf{k}^0_\mathsf{a}, \mathbf{k}^0_\mathsf{b})(G)(s_0)(s)(\tau)(\tau_0)$,

then $(e_\mathsf{a}, e_\mathsf{b}) \in \mathbf{E}_W(i+j)(\mathbf{k}^0_\mathsf{a}, \mathbf{k}^0_\mathsf{b})(G)(s_0)(s)(\phi)(\tau_0)$.

Proof. By a fairly straightforward coinduction, at the same time proving a similar property for **K**. □

Analogously to Section 3.7.1, we define a version of $\mathbf{E}$ (in terms of the original $\mathbf{E}$) that parameterizes over the value relation by which functions are related in the CALL case.

Definition 27.

$$
\begin{aligned}
&\mathbf{E}^{\mathcal{G}}_W \in \mathrm{ES}^W_{A,B}\\
&\mathbf{E}^{\mathcal{G}}_W(i)(\mathbf{k}^0_\mathsf{a},\mathbf{k}^0_\mathsf{b})(G)(s_0)(s)(\phi)(\tau) = \{\,(e_\mathsf{a},e_\mathsf{b}) \mid G \in \mathbf{GK}_W \implies\\
&\quad \forall c_\mathsf{a},c_\mathsf{b},\eta_\mathsf{a},\eta_\mathsf{b}.\ \forall (m_\mathsf{a},m_\mathsf{b}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s)(\phi)(e_\mathsf{a},e_\mathsf{b})(c_\mathsf{a},c_\mathsf{b})(\eta_\mathsf{a},\eta_\mathsf{b}).\\
&\quad (\textsc{err})\ \ \exists m'_\mathsf{b}.\ m_\mathsf{b} \overset{\iota}{\hookrightarrow}{}^{*}\ m'_\mathsf{b} \wedge m'_\mathsf{b} \in \mathbf{error}\\
&\quad \vee\ (\textsc{ret})\ \ \exists m'_\mathsf{b}, s', \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}, e'_\mathsf{a}, e'_\mathsf{b}, c'_\mathsf{a}, c'_\mathsf{b}.\ m_\mathsf{b} \overset{\iota}{\hookrightarrow}{}^{*}\ m'_\mathsf{b} \wedge s' \sqsupseteq s \wedge s' \sqsupseteq_{\mathrm{pub}} s_0 \wedge\\
&\qquad (m_\mathsf{a},m'_\mathsf{b}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s')(\bot)(e'_\mathsf{a},e'_\mathsf{b})(c'_\mathsf{a},c'_\mathsf{b})(\eta_\mathsf{a},\eta_\mathsf{b}) \wedge (\mathbf{v}_\mathsf{a},\mathbf{v}_\mathsf{b}) \in \overline{G}(s')(\tau) \wedge\\
&\qquad (e'_\mathsf{a},e'_\mathsf{b}) \in W.\mathsf{cqh}_\mathsf{a}(s')(\mathsf{ret}\,\mathbf{v}_\mathsf{a}\,\mathbf{k}^0_\mathsf{a}) \times W.\mathsf{cqh}_\mathsf{b}(s')(\mathsf{ret}\,\mathbf{v}_\mathsf{b}\,\mathbf{k}^0_\mathsf{b})\\
&\quad \vee\ (\textsc{step})\ m_\mathsf{a} \notin \mathbf{halted} \wedge \forall t, m'_\mathsf{a}.\ m_\mathsf{a} \overset{t}{\hookrightarrow} m'_\mathsf{a} \implies \exists i', e'_\mathsf{a}, e'_\mathsf{b}, c'_\mathsf{a}, c'_\mathsf{b}, m'_\mathsf{b}, m''_\mathsf{b}, s', \phi'.\\
&\qquad m_\mathsf{b} \overset{\iota}{\hookrightarrow}{}^{*}\ m'_\mathsf{b} \wedge s' \sqsupseteq s \wedge (m'_\mathsf{a},m''_\mathsf{b}) \in \mathsf{cfg}(W.\mathsf{C})(G)(s')(\phi')(e'_\mathsf{a},e'_\mathsf{b})(c'_\mathsf{a},c'_\mathsf{b})(\eta_\mathsf{a},\eta_\mathsf{b}) \wedge\\
&\qquad\quad (\textsc{rec})\ \ (m'_\mathsf{b} \overset{t}{\hookrightarrow} m''_\mathsf{b} \wedge \vee m'_\mathsf{b} = m''_\mathsf{b} \wedge t = \iota \wedge i' < i) \wedge \phi' \neq \bot \wedge\\
&\qquad\qquad (e'_\mathsf{a},e'_\mathsf{b}) \in \mathbf{E}_W(i')(\mathbf{k}^0_\mathsf{a},\mathbf{k}^0_\mathsf{b})(G)(s_0)(s')(\phi')(\tau)\\
&\qquad\quad \vee\ (\textsc{call})\ m'_\mathsf{b} \overset{t}{\hookrightarrow} m''_\mathsf{b} \wedge \phi' = \bot \wedge\\
&\qquad\qquad (e'_\mathsf{a},e'_\mathsf{b}) \in \mathbf{U}(s')(\mathcal{G}(s'))(G(s'))(\mathbf{K}_W(i')(\mathbf{k}^0_\mathsf{a},\mathbf{k}^0_\mathsf{b})(G)(s_0)(s'))(\tau)\,\}
\end{aligned}
$$

Lemma 55 (External call)**.** If $G = \mathbf{F}_W(G) \cup W.\mathsf{NR}(G) \cup \mathcal{G}$, then

$$\mathbf{E}_W(i)(\mathbf{k}_\mathsf{a},\mathbf{k}_\mathsf{b})(G) = \mathbf{E}^{\mathcal{G}}_W(i)(\mathbf{k}_\mathsf{a},\mathbf{k}_\mathsf{b})(G).$$

Proof. The proof is fairly similar to the PB version (Lemma 12 in Section 3.7.1). The interesting direction is "$\subseteq$". Due to the single-step style of PILS, we don't need to do any induction on steps here, in contrast to the PB proof. As before, the crucial part is the composition of $\mathbf{E}$ and $\mathbf{K}$ in the CALL case (here via Lemma 54).

Note that although the difference between $\mathbf{E}$ and $\mathbf{E}^{\mathcal{G}}$ is apparently "lost" if the programs are related via the REC case, it can of course be restored by immediately re-applying the lemma. Indeed, we will do so in the proof of Lemma 56. □

4.7.2 Adequacy

With the external call lemma at hand, we can now approach adequacy. We are only interested in adequacy of $\precsim_{\mathcal{TS}}$, the target-source module refinement. As a lemma, we nevertheless prove adequacy of $\mathbf{E}$ generically for any instance of the typed model (because we can). It states that if e_a and e_b are related by $\mathbf{E}$ at a *consistent* global knowledge and relative to *terminal* continuations, then any completions of e_a and e_b to proper machines m_a and m_b satisfy the standard refinement property. *Consistent* here means that, modulo type names, the "external part" of G is empty—see condition 2 below. *Terminal* here means that as soon as the continuations obtain control, the programs are halted—see conditions 3 and 4 below. Otherwise, the relatedness of e_a and e_b would only account for their behavior until they reach their

continuations, while $\mathbf{behav}(m_\mathsf{a})$ and $\mathbf{behav}(m_\mathsf{b})$ of course include the behavior of these continuations.

Lemma 56 (Generic adequacy of typed **E**)**.** In the context of $W \in \mathrm{World}_{A,B}$, if

1. $(e_\mathsf{a}, e_\mathsf{b}) \in \mathbf{E}_W(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\phi)(\tau)$
2. $G \in \mathbf{GK}_W \wedge G = \mathbf{F}(G) \cup W.\mathsf{NR}(G)$
3. $\forall s \sqsupseteq s_0.\ \forall v_\mathsf{a}.\ \forall e'_\mathsf{a} \in W.\mathsf{cqh}_\mathsf{a}(s)(\mathsf{ret}\, v_\mathsf{a}\, \mathbf{k}_\mathsf{a}).\ \forall c'_\mathsf{a}.\ \mathbf{real}(e'_\mathsf{a} \cdot c'_\mathsf{a}) \subseteq \mathbf{halted}$
4. $\forall s \sqsupseteq s_0.\ \forall v_\mathsf{b}.\ \forall e'_\mathsf{b} \in W.\mathsf{cqh}_\mathsf{b}(s)(\mathsf{ret}\, v_\mathsf{b}\, \mathbf{k}_\mathsf{b}).\ \forall c'_\mathsf{b}.\ \mathbf{real}(e'_\mathsf{b} \cdot c'_\mathsf{b}) \subseteq \mathbf{halted}$
5. $\eta_\mathsf{a} \in \mathbf{frame} \wedge \eta_\mathsf{b} \in \mathbf{frame}$
6. $(m_\mathsf{a}, m_\mathsf{b}) \in \mathsf{cfg}(G)(s)(\phi)(e_\mathsf{a}, e_\mathsf{b})(c_\mathsf{a}, c_\mathsf{b})(\eta_\mathsf{a}, \eta_\mathsf{b})$

then $\mathbf{behav}(m_\mathsf{a}) \subseteq \mathbf{behav}(m_\mathsf{b})$.

Proof. By coinduction on m_b's behavior, with a nested well-founded induction on i. We critically rely on the external call Lemma 55 to rule out any use of the CALL disjunct in (1). □

Theorem 13 (Adequacy of $\precsim_{\mathcal{TS}}$)**.** If

1. $\epsilon \vdash M_\mathsf{a} \precsim_{\mathcal{TS}} M_\mathsf{b} : \Gamma$
2. $\Gamma(F_{\mathrm{main}}) = \mathsf{unit} \to \tau$
3. $m_\mathsf{a} = \mathit{load}(M_\mathsf{a})$
4. $m_\mathsf{b} = \mathsf{load}(M_\mathsf{b})$

then $\mathbf{behav}(m_\mathsf{a}) \subseteq \mathbf{behav}(m_\mathsf{b})$.

Proof. We define the following configurations:

$$c^1_\mathsf{a} := (\emptyset, R, m_\mathsf{a}.q, h_1)$$
$$c^2_\mathsf{a} := (\emptyset, \emptyset, \emptyset, h_2)$$

$$c^1_\mathsf{b} := (\emptyset, m_\mathsf{b}.\sigma, \emptyset)$$
$$c^2_\mathsf{b} := \mathbf{\emptyset}$$

Here, R is $m_\mathsf{a}.R$ and h_1, h_2 are the global and local heap (respectively) returned by load' in the definition of $\mathit{load}(M_\mathsf{a})$.

With the help of (3) and (4), it is easy to show that these configurations are proper loadings of the given modules (using load address 0 in the case of M_a):

5. $(c^1_\mathsf{a}, c^2_\mathsf{a}) \in \mathbf{cload}(M_\mathsf{a})(0)(\emptyset)$
6. $(c^1_\mathsf{b}, c^2_\mathsf{b}) \in \mathbf{cload}(M_\mathsf{b})(1)(\emptyset)$

Using these, we instantiate (1). This yields a local world $w \in \mathrm{LWorld}(\Omega_{\mathcal{TS}})$ and, for $G := [w\uparrow]$, a state s_0 and values v_a, v_b such that:

7. $(c^1_\mathsf{a}, c^1_\mathsf{b}) \in \Omega_{\mathcal{TS}}.\mathsf{C}(G(-, \mathit{snd}\, s_0))$
8. $(c^2_\mathsf{a}, c^2_\mathsf{b}) \in w.\mathsf{C}(G)(s_0)$
9. $v_\mathsf{a} \in \mathbf{vload}(M_\mathsf{a})(0)(\emptyset)(F_{\mathrm{main}}) \wedge v_\mathsf{b} \in \mathbf{vload}(M_\mathsf{b})(1)(\emptyset)(F_{\mathrm{main}})$
10. $(v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s_0)(\mathsf{unit} \to \tau)$

Note that (10) implies $(v_\mathsf{a}, v_\mathsf{b}) \in \mathbf{F}_{w\uparrow}(G)(s_0)(\mathsf{unit} \to \tau)$ due to Definition 26, and also that $v_\mathsf{b} = \mathsf{fix}\, f(x).\, e$ (for some f, x, e).

We know that m_a and m_b look as follows (for a particular n):

$$m_\mathsf{a} = (n, \emptyset, \emptyset, \emptyset) \cdot c^1_\mathsf{a} \cdot c^2_\mathsf{a}$$
$$m_\mathsf{b} = (\emptyset, \emptyset, v_\mathsf{b}\ \langle\rangle) \cdot c^1_\mathsf{b} \cdot c^2_\mathsf{b}$$

Now, let $e_\mathsf{a} = (n, \emptyset, \emptyset, \emptyset)$ and $e_\mathsf{b} = (\emptyset, \emptyset, e[v_\mathsf{b}/f][\langle\rangle/x])$. Then it is easy to see that e_a and e_b represent the following "applications" of v_a and v_b:

$$e_\mathsf{a} \in w\uparrow.\mathsf{cqh}_\mathsf{a}(s_0)(\mathsf{app}\ v_\mathsf{a}\ R(\mathsf{arg})\ R(\mathsf{ret}))$$
$$e_\mathsf{b} \in w\uparrow.\mathsf{cqh}_\mathsf{b}(s_0)(\mathsf{app}\ v_\mathsf{b}\ \langle\rangle\ \bullet)$$

Since the arguments are related by $\overline{G}(s_0)(\mathsf{unit})$ trivially and since (7) and (8) compose to $(c^1_\mathsf{a} \cdot c^2_\mathsf{a}, c^1_\mathsf{b} \cdot c^2_\mathsf{b}) \in w\uparrow.\mathsf{C}(G)(s_0)$, we can instantiate $(v_\mathsf{a}, v_\mathsf{b}) \in \mathbf{F}_{w\uparrow}(G)(s_0)(\mathsf{unit} \to \tau)$ such that we obtain the following:

$$(e_\mathsf{a}, e_\mathsf{b}) \in \mathbf{E}_{w\uparrow}(i)(R(\mathsf{ret}), \bullet)(G)(s_0)(s_0)(\bot)(\tau)$$

It suffices to apply Lemma 56 to this, using empty frames. Note that $R(\mathsf{ret}) = 0$ and thus the target continuation is terminal as required (a target machine cannot step if the program counter is 0). The source continuation $\bullet$ is terminal as well, as a source machine cannot step if its expression is a value. □

4.7.3 Modularity

We now sketch the proof of modularity, starting with some lemmas about the product of local worlds. Recall its definition, as well as that of $G^{s_2}_{\langle 1\rangle}$ and $G^{s_1}_{\langle 2\rangle}$ from Figure 4.14.

4.7.3.1 Generic modularity of E

Lemma 57 (PILS version of Lemma 9)**.** If $w_1, w_2 \in \mathsf{stable}(\Omega)$ and $w = w_1 \otimes w_2$, then:

1. $s_2 \sqsupseteq_{\mathrm{pub}} s^0_2 \implies$
   $\mathbf{E}_{w_1\uparrow}(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G^{s_2}_{\langle 1\rangle})(s^0, s^0_1)(s, s_1)(\bot) \subseteq$
   $\mathbf{E}_{w\uparrow}(i, 0)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s^0, (s^0_1, s^0_2))(s, (s_1, s_2))(\bot)$

2. $s_1 \sqsupseteq_{\text{pub}} s_1^0 \implies$
   $\mathbf{E}_{w_2\uparrow}(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G^{s_1}_{\langle 2\rangle})(s^0, s_2^0)(s, s_2)(\bot) \subseteq$
   $\mathbf{E}_{w\uparrow}(0, i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s^0, (s_1^0, s_2^0))(s, (s_1, s_2))(\bot)$

3. $s_2 \sqsupseteq_{\text{pub}} s_2^0 \wedge s' \sqsupseteq s \wedge (c_\mathsf{a}, c_\mathsf{b}) \in w_2.\mathsf{C}(G^{s_1'}_{\langle 2\rangle})(s, s_2) \implies$
   $\mathbf{E}_{w_1\uparrow}(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G^{s_2}_{\langle 1\rangle})(s^0, s_1^0)(s', s_1')(c_\mathsf{a} \cdot \eta_\mathsf{a}, c_\mathsf{b} \cdot \eta_\mathsf{b}) \subseteq$
   $\mathbf{E}_{w\uparrow}(i, 0)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s^0, (s_1^0, s_2^0))(s', (s_1', s_2))(\eta_\mathsf{a}, \eta_\mathsf{b})$

4. $s_1 \sqsupseteq_{\text{pub}} s_1^0 \wedge s' \sqsupseteq s \wedge (c_\mathsf{a}, c_\mathsf{b}) \in w_1.\mathsf{C}(G^{s_2'}_{\langle 1\rangle})(s, s_1) \implies$
   $\mathbf{E}_{w_2\uparrow}(i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G^{s_1}_{\langle 2\rangle})(s^0, s_2^0)(s', s_2')(c_\mathsf{a} \cdot \eta_\mathsf{a}, c_\mathsf{b} \cdot \eta_\mathsf{b}) \subseteq$
   $\mathbf{E}_{w\uparrow}(0, i)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s^0, (s_1^0, s_2^0))(s', (s_1, s_2'))(\eta_\mathsf{a}, \eta_\mathsf{b})$

We actually only care about first two, but need to prove the last two at the same time. Moreover, in contrast to PBs, here we cannot prove (1) and (2) separately, in each relying only on the stability of the respective local world. Why is this? In PBs, due to GK being defined in terms of the world's local knowledge, we could prove Lemma 8 a priori, and then prove Lemma 9 on top of that. In PILS, **GK** is mutually recursive with **E**. Proving the analogue of Lemma 8 hence already requires the analogue of Lemma 9. More precisely, the proof of "$\mathbf{E}_{w_1\uparrow} \subseteq \mathbf{E}_{w\uparrow}$" relies on stability of w_2. In order to make use of this stability, we also need to know part (2) of the lemma below. However, proving that part in turn requires "$\mathbf{E}_{w_2\uparrow} \subseteq \mathbf{E}_{w\uparrow}$". So we actually require stability of both w_1 and w_2 and have to prove everything simultaneously (also for **F** and **K**.

Lemma 58 (PILS version of Lemma 8)**.** If $w_1, w_2 \in \mathsf{stable}(\Omega)$ and $w = w_1 \otimes w_2$ and $G \in \mathbf{GK}_{w\uparrow}$, then:

1. $\forall s_2 \in w_2.\mathsf{S}.\ G^{s_2}_{\langle 1\rangle} \in \mathbf{GK}_{w_1\uparrow}$
2. $\forall s_1 \in w_1.\mathsf{S}.\ G^{s_1}_{\langle 2\rangle} \in \mathbf{GK}_{w_2\uparrow}$

Lemma 59 (PILS version of part of Lemma 10)**.** If $w_1, w_2 \in \mathsf{stable}(\Omega)$ and $w = w_1 \otimes w_2$, then $w \in \mathsf{stable}(\Omega)$.

4.7.3.2 Modularity of $\precsim_{\mathcal{TS}}$

Based on these generic results, we now sketch the proof of the key modularity property that we are interested in, namely that $\precsim_{\mathcal{TS}}$ is preserved under linking:

$$\frac{\Gamma \vdash M_\mathsf{a}^1 \precsim_{\mathcal{TS}} M_\mathsf{b}^1 : \Gamma_1 \quad \Gamma, \Gamma_1 \vdash M_\mathsf{a}^2 \precsim_{\mathcal{TS}} M_\mathsf{b}^2 : \Gamma_2}{\Gamma \vdash \mathsf{link}(M_\mathsf{a}^1, M_\mathsf{a}^2) \precsim_{\mathcal{TS}} \mathsf{link}(M_\mathsf{b}^1, M_\mathsf{b}^2) : \Gamma_1, \Gamma_2}$$

(Here we drop a few well-formedness side conditions. The fulll theorem is stated at the end of the section as Theorem 14.) This proof necessarily involves several language details, as otherwise we could have proven a generic version.

Let $M_\mathsf{a} := \mathsf{link}(M_\mathsf{a}^1, M_\mathsf{a}^2)$ and $M_\mathsf{b} := \mathsf{link}(M_\mathsf{b}^1, M_\mathsf{b}^2)$. We are first given a countably infinite set $\mathcal{N}$ of type names, which we split into two disjoint infinite parts $\mathcal{N}_1$ and $\mathcal{N}_2$. Using these to instantiate the two premises gives us stable local worlds w_1 and w_2, respectively, with disjoint type name sets. As our local world, we choose their product $w := w_1 \otimes w_2$, which is stable as well by Lemma 59.

Next we are given anchors, import lists, and pairs of configurations corresponding to appropriate loadings of the modules:

$$(c_\mathsf{a}^g, c_\mathsf{a}^l) \in \mathbf{cload}(M_\mathsf{a})(\Psi_\mathsf{a})(\gamma_\mathsf{a})$$
$$(c_\mathsf{b}^g, c_\mathsf{b}^l) \in \mathbf{cload}(M_\mathsf{b})(\Psi_\mathsf{b})(\gamma_\mathsf{b})$$

Because we know the $\mathcal{T}$ and S implementations of **cload**, we know what these configurations look like:

$$c_\mathsf{a}^g = (\emptyset, R, q, h \sqcup h^g) \qquad c_\mathsf{b}^g = (\emptyset, \gamma_\mathsf{b} \uplus M_\mathsf{b} \uplus \rho, \emptyset)$$
$$c_\mathsf{a}^l = (\emptyset, \emptyset, \emptyset, h^l) \qquad c_\mathsf{b}^l = \mathbf{\emptyset}$$

$$\text{where } (h^g, h^l, x) = \mathit{load}'(M_\mathsf{a})(\Psi_\mathsf{a})(\mathit{map}\ \mathit{snd}\ \gamma_\mathsf{a})$$

We can now take these loadings of M_a and M_b, and cut away some data in order to obtain loadings of M_a^1 and M_b^1, and similarly of M_a^2 and M_b^2. Formally, we define

$$\Psi_\mathsf{a}^1 := \Psi_\mathsf{a} \qquad \Psi_\mathsf{b}^1 := 1 \quad (= \Psi_\mathsf{b})$$
$$\gamma_\mathsf{a}^1 := \gamma_\mathsf{a} \qquad \gamma_\mathsf{b}^1 := \gamma_\mathsf{b}$$
$$c_\mathsf{a}^{g,1} := (\emptyset, R, q, h \sqcup h_1^g) \qquad c_\mathsf{b}^{g,1} := (\emptyset, \gamma_\mathsf{b} \uplus M_\mathsf{b}^1 \uplus (M_\mathsf{b}^2 \uplus \rho), \emptyset)$$
$$c_\mathsf{a}^{l,1} := (\emptyset, \emptyset, \emptyset, h_1^l) \qquad c_\mathsf{b}^{l,1} := \mathbf{\emptyset}$$

$$\Psi_\mathsf{a}^2 := \Psi_\mathsf{a} + \mathit{size}(h_1^g \sqcup h_1^l) \qquad \Psi_\mathsf{b}^2 := 1$$
$$\gamma_\mathsf{a}^2 := \gamma_\mathsf{a}, x_1 \qquad \gamma_\mathsf{b}^2 := \gamma_\mathsf{b}, M_\mathsf{b}^1$$
$$c_\mathsf{a}^{g,2} := (\emptyset, R, q, (h \sqcup h_1^g) \sqcup h_2^g) \qquad c_\mathsf{b}^{g,2} := (\emptyset, \gamma_\mathsf{b}^2 \uplus M_\mathsf{b}^2 \uplus \rho, \emptyset)$$
$$c_\mathsf{a}^{l,2} := (\emptyset, \emptyset, \emptyset, h_2^l) \qquad c_\mathsf{b}^{l,2} := \mathbf{\emptyset}$$

$$\text{where } (h_1^g, h_1^l, x_1) = \mathit{load}'(M_\mathsf{a}^1)(\Psi_\mathsf{a}^1)(\mathit{map}\ \mathit{snd}\ \gamma_\mathsf{a}^1)$$
$$(h_2^g, h_2^l, x_2) = \mathit{load}'(M_\mathsf{a}^2)(\Psi_\mathsf{a}^2)(\mathit{map}\ \mathit{snd}\ \gamma_\mathsf{a}^2)$$

and are then able to show the following:

$$(c_\mathsf{a}^{g,1}, c_\mathsf{a}^{l,1}) \in \mathbf{cload}(M_\mathsf{a}^1)(\Psi_\mathsf{a}^1)(\gamma_\mathsf{a}^1)$$
$$(c_\mathsf{b}^{g,1}, c_\mathsf{b}^{l,1}) \in \mathbf{cload}(M_\mathsf{b}^1)(\Psi_\mathsf{b}^1)(\gamma_\mathsf{b}^1)$$

$$(c_\mathsf{a}^{g,2}, c_\mathsf{a}^{l,2}) \in \mathbf{cload}(M_\mathsf{a}^2)(\Psi_\mathsf{a}^2)(\gamma_\mathsf{a}^2)$$
$$(c_\mathsf{b}^{g,2}, c_\mathsf{b}^{l,2}) \in \mathbf{cload}(M_\mathsf{b}^2)(\Psi_\mathsf{b}^2)(\gamma_\mathsf{b}^2)$$

This may look complicated but is actually quite simple. There aren't any real choices involved here—everything fits together in a canonical way. For instance, Ψ_a^2 is exactly

the address in the heap where $load'(\mathsf{link}(M^1_\mathsf{a}, M^2_\mathsf{a}))$ places the environment for and code of M^2_a.

Note that, assuming well-typedness, (the functions exported by) $\mathbf{Mod}^1_\mathsf{b}$ and M^2_b are disjoint. Hence c^g_b, $c^{g,1}_\mathsf{b}$, and $c^{g,2}_\mathsf{b}$ are actually all the same, just written in a different way. Also note that $h^g_1 \sqcup h^g_2 = h^g$ and $h^l_1 \sqcup h^l_2 = h^l$ and $x_1, x_2 = x$ by definition of $load'$.

We now further instantiate the premises with these constructed loadings. The first premise yields an initial state $(s^\circ_1, s_1) \in w_1{\uparrow}.\mathsf{S}$ such that:

1. $\forall G \in \mathbf{GK}_{w_1\uparrow}.\ (c^{g,1}_\mathsf{a}, c^{g,1}_\mathsf{b}) \in \Omega_{\mathcal{TS}}.\mathsf{C}(G(-, s_1))(s^\circ_1)$
2. $\forall G \in \mathbf{GK}_{w_1\uparrow}.\ (c^{l,1}_\mathsf{a}, c^{l,1}_\mathsf{b}) \in w_1.\mathsf{C}(G)(s^\circ_1, s_1)$
3. $\Omega_{\mathcal{TS}}.\mathsf{rqh}(s^\circ_1) = \emptyset$
4. $\forall f'{:}\tau' \in \Gamma_1.\ \exists (v_\mathsf{a}, v_\mathsf{b}) \in \mathbf{vload}(M^1_\mathsf{a})(\Psi^1_\mathsf{a})(\gamma^1_\mathsf{a})(f') \times \mathbf{vload}(M^1_\mathsf{b})(\Psi^1_\mathsf{b})(\gamma^1_\mathsf{b})(f').$
   $\forall s' \sqsupseteq (s^\circ_1, s_1).\ \forall G \in \mathbf{GK}_{w_1\uparrow}.\ (s_1 \in \mathsf{sat}(G) \wedge \forall f{:}\tau \in \Gamma.\ (\gamma^1_\mathsf{a} f, \gamma^1_\mathsf{b} f) \in \overline{G}(s')(\tau)) \implies$
   $(v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s')(\tau')$

Similarly, the second premise yields a state $(s^\circ_2, s_2) \in w_2{\uparrow}.\mathsf{S}$ such that:

5. $\forall G \in \mathbf{GK}_{w_2\uparrow}.\ (c^{g,2}_\mathsf{a}, c^{g,2}_\mathsf{b}) \in \Omega_{\mathcal{TS}}.\mathsf{C}(G(-, s_2))(s^\circ_2)$
6. $\forall G \in \mathbf{GK}_{w_2\uparrow}.\ (c^{l,2}_\mathsf{a}, c^{l,2}_\mathsf{b}) \in w_2.\mathsf{C}(G)(s^\circ_2, s_2)$
7. $\Omega_{\mathcal{TS}}.\mathsf{rqh}(s^\circ_2) = \emptyset$
8. $\forall f'{:}\tau' \in \Gamma_2.\ \exists (v_\mathsf{a}, v_\mathsf{b}) \in \mathbf{vload}(M^2_\mathsf{a})(\Psi^2_\mathsf{a})(\gamma^2_\mathsf{a})(f') \times \mathbf{vload}(M^2_\mathsf{b})(\Psi^2_\mathsf{b})(\gamma^2_\mathsf{b})(f').$
   $\forall s' \sqsupseteq (s^\circ_2, s_2).\ \forall G \in \mathbf{GK}_{w_2\uparrow}.\ (s_2 \in \mathsf{sat}(G) \wedge \forall f{:}\tau \in \Gamma, \Gamma_1.\ (\gamma^2_\mathsf{a} f, \gamma^2_\mathsf{b} f) \in \overline{G}(s')(\tau)) \implies$
   $(v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s')(\tau')$

Note that (4) only cares about values provided for Γ, while (8) cares about values provided for Γ, Γ_1 (recall that linking is asymmetric—the right module may depend on the left but not vice versa).

We now need to come up with a state $(s^\circ, s) \in w{\uparrow}.\mathsf{S}$ such that:

9. $\forall G \in \mathbf{GK}_{w\uparrow}.\ (c^g_\mathsf{a}, c^g_\mathsf{b}) \in \Omega_{\mathcal{TS}}.\mathsf{C}(G(-, s))(s^\circ)$
10. $\forall G \in \mathbf{GK}_{w\uparrow}.\ (c^l_\mathsf{a}, c^l_\mathsf{b}) \in w.\mathsf{C}(G)(s^\circ, s)$
11. $\Omega_{\mathcal{TS}}.\mathsf{rqh}(s^\circ) = \emptyset$
12. $\forall f'{:}\tau' \in \Gamma_1, \Gamma_2.\ \exists (v_\mathsf{a}, v_\mathsf{b}) \in \mathbf{vload}(M_\mathsf{a})(\Psi_\mathsf{a})(\gamma_\mathsf{a})(f') \times \mathbf{vload}(M_\mathsf{b})(\Psi_\mathsf{b})(\gamma_\mathsf{b})(f').$
    $\forall s' \sqsupseteq (s^\circ, s).\ \forall G \in \mathbf{GK}_{w\uparrow}.\ (s \in \mathsf{sat}(G) \wedge \forall f{:}\tau \in \Gamma.\ (\gamma_\mathsf{a} f, \gamma_\mathsf{b} f) \in \overline{G}(s')(\tau)) \implies$
    $(v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s')(\tau')$

Before we define s° and s, we observe that $s_1^\circ \in \Omega_{\mathcal{TS}}.\mathsf{S}$ and $s_2^\circ \in \Omega_{\mathcal{TS}}.\mathsf{S}$ necessarily look as follows (for some registries κ_1, κ_2):

$$s_1^\circ = ((R, \kappa_1), \emptyset, \gamma_\mathsf{b} \uplus M_\mathsf{b} \uplus \rho) \qquad s_2^\circ = ((R, \kappa_2), \emptyset, \gamma_\mathsf{b} \uplus M_\mathsf{b} \uplus \rho)$$

The fact that in both states the $\mathsf{T}_{\mathsf{ref}}^{\mathcal{TS}}$ component is empty ($\emptyset$) follows from condition (3) and (7), respectively. The knowledge about the unary components comes from (1) and (5), which we can instantiate using the least global knowledges $[w_1\uparrow]$ and $[w_2\uparrow]$, respectively.

We define $s^\circ \in \Omega_{\mathcal{TS}}.\mathsf{S}$ as follows:

$$s^\circ := ((R, \kappa_1 \cup \kappa_2), \emptyset, \gamma_\mathsf{b} \uplus M_\mathsf{b} \uplus \rho)$$

Observe that we have $s^\circ \sqsupseteq_{\text{pub}} s_1^\circ$ and $s^\circ \sqsupseteq_{\text{pub}} s_2^\circ$. With the help of mainly (1), (5), and Lemma 58, it is not hard to show condition (9) for any s satisfying $s \sqsupseteq (s_1, s_2)$. For instance, since h_1^g conforms to κ_1 and h_2^g conforms to κ_2, h^g conforms to $\kappa_1 \cup \kappa_2$.

So what about s? The obvious choice of $s := (s_1, s_2)$ does not let us prove (10). To see this, suppose $G \in \mathbf{GK}_{w\uparrow}$. Since $c_\mathsf{a}^l = c_\mathsf{a}^{l,1} \cdot c_\mathsf{a}^{l,2}$ (and similarly for c_b^l), the goal $(c_\mathsf{a}^l, c_\mathsf{b}^l) \in w.\mathsf{C}(G)(s^\circ, s)$ would follow directly from

$$(c_\mathsf{a}^{l,1}, c_\mathsf{b}^{l,1}) \in w_1.\mathsf{C}(G_{\langle 1\rangle}^{s_2})(s^\circ, s_1) \qquad \text{and} \qquad (c_\mathsf{a}^{l,2}, c_\mathsf{b}^{l,2}) \in w_2.\mathsf{C}(G_{\langle 2\rangle}^{s_1})(s^\circ, s_2)$$

by construction of w as the product of w_1 and w_2. However, (2) and (6) only tell us

$$(c_\mathsf{a}^{l,1}, c_\mathsf{b}^{l,1}) \in w_1.\mathsf{C}(G_{\langle 1\rangle}^{s_2})(s_1^\circ, s_1) \qquad \text{and} \qquad (c_\mathsf{a}^{l,2}, c_\mathsf{b}^{l,2}) \in w_2.\mathsf{C}(G_{\langle 2\rangle}^{s_1})(s_2^\circ, s_2).$$

While $s^\circ \sqsupseteq s_i^\circ$ holds, these configuration relations may of course not be monotone in their state argument.

Fortunately, the solution is easy: we apply stability of w_1 and w_2 to the previous statements. Doing so yields $s_1' \sqsupseteq_{\text{pub}} s_1$ and $s_2' \sqsupseteq_{\text{pub}} s_2$ such that

$$(c_\mathsf{a}^{l,1}, c_\mathsf{b}^{l,1}) \in w_1.\mathsf{C}(G_{\langle 1\rangle}^{s_2})(s^\circ, s_1') \qquad \text{and} \qquad (c_\mathsf{a}^{l,2}, c_\mathsf{b}^{l,2}) \in w_2.\mathsf{C}(G_{\langle 2\rangle}^{s_1})(s^\circ, s_2').$$

Now we can pick $s := (s_1', s_2')$ and (10) follows by monotonicity in the global knowledge.

Condition (11) obviously holds because we chose the reference bijection in s° to be empty. It thus remains to show (12). In order to do so, the following will be very helpful:

13. $\forall f'{:}\tau' \in \Gamma_1.\ \mathbf{vload}(M_\mathsf{a}^1)(\Psi_\mathsf{a}^1)(\gamma_\mathsf{a}^1)(f') \subseteq \mathbf{vload}(\mathsf{link}(M_\mathsf{a}^1, M_\mathsf{a}^2))(\Psi_\mathsf{a})(\gamma_\mathsf{a})(f')$
14. $\forall f'{:}\tau' \in \Gamma_2.\ \mathbf{vload}(M_\mathsf{a}^2)(\Psi_\mathsf{a}^2)(\gamma_\mathsf{a}^2)(f') \subseteq \mathbf{vload}(\mathsf{link}(M_\mathsf{a}^1, M_\mathsf{a}^2))(\Psi_\mathsf{a})(\gamma_\mathsf{a})(f')$
15. $\forall f'{:}\tau' \in \Gamma_1.\ \mathbf{vload}(M_\mathsf{b}^1)(\Psi_\mathsf{b}^1)(\gamma_\mathsf{b}^1)(f') \subseteq \mathbf{vload}(\mathsf{link}(M_\mathsf{b}^1, M_\mathsf{b}^2))(\Psi_\mathsf{b})(\gamma_\mathsf{b})(f')$
16. $\forall f'{:}\tau' \in \Gamma_2.\ \mathbf{vload}(M_\mathsf{b}^2)(\Psi_\mathsf{b}^2)(\gamma_\mathsf{b}^2)(f') \subseteq \mathbf{vload}(\mathsf{link}(M_\mathsf{b}^1, M_\mathsf{b}^2))(\Psi_\mathsf{b})(\gamma_\mathsf{b})(f')$

They roughly say that retrieving the value of exported function f' from the composed module is the same as retrieving it from the component module in which it is defined (recall that Γ_i describes the exports of M^i_{a} and M^i_{b}). Their proofs are straightforward, relying on the exports of the linked modules being disjoint.

So consider $f'{:}\tau' \in \Gamma_1, \Gamma_2$. This means either $f'{:}\tau' \in \Gamma_1$ or $f'{:}\tau' \in \Gamma_2$. In the first case, we use (4) with (13) and (15) to obtain values $v_{\mathsf{a}}, v_{\mathsf{b}}$ for which we know

$$\begin{array}{l}\forall s' \sqsupseteq (s_1^\circ, s_1).\ \forall G \in \mathbf{GK}_{w_1\uparrow}.\\ (s_1 \in \mathsf{sat}(G) \wedge \forall f{:}\tau \in \Gamma.\ (\gamma^1_{\mathsf{a}} f, \gamma^1_{\mathsf{b}} f) \in \overline{G}(s')(\tau)) \implies (v_{\mathsf{a}}, v_{\mathsf{b}}) \in \overline{G}(s')(\tau')\end{array}$$

and must show:

$$\begin{array}{l}\forall s' \sqsupseteq (s^\circ, s).\ \forall G \in \mathbf{GK}_{w\uparrow}.\\ (s \in \mathsf{sat}(G) \wedge \forall f{:}\tau \in \Gamma.\ (\gamma_{\mathsf{a}} f, \gamma_{\mathsf{b}} f) \in \overline{G}(s')(\tau)) \implies (v_{\mathsf{a}}, v_{\mathsf{b}}) \in \overline{G}(s')(\tau')\end{array}$$

We know that the given future state s' must have the form $(s^{\circ\prime}, (s''_1, s''_2))$ with $s^{\circ\prime} \sqsupseteq s^\circ \sqsupseteq s_1^\circ$ and $s''_1 \sqsupseteq s'_1 \sqsupseteq s_1$. Therefore we can instantiate the premise above with $(s^{\circ\prime}, s''_1) \sqsupseteq (s_1^\circ, s_1)$. It is then easy to check the rest. In particular, if $\gamma_{\mathsf{a}}, \gamma_{\mathsf{b}}$ is pointwise related for labels in Γ, then so is $\gamma^1_{\mathsf{a}}, \gamma^1_{\mathsf{b}}$ since it is the same pair by construction.

In the second case, where $f'{:}\tau' \in \Gamma_2$, we use (8) with (14) and (16) to obtain $v_{\mathsf{a}}, v_{\mathsf{b}}$ for which we know

$$\begin{array}{l}\forall s' \sqsupseteq (s_2^\circ, s_2).\ \forall G \in \mathbf{GK}_{w_2\uparrow}.\\ (s_2 \in \mathsf{sat}(G) \wedge \forall f{:}\tau \in \Gamma, \Gamma_1.\ (\gamma^2_{\mathsf{a}} f, \gamma^2_{\mathsf{b}} f) \in \overline{G}(s')(\tau)) \implies (v_{\mathsf{a}}, v_{\mathsf{b}}) \in \overline{G}(s')(\tau')\end{array}$$

and must show:

$$\begin{array}{l}\forall s' \sqsupseteq (s^\circ, s).\ \forall G \in \mathbf{GK}_{w\uparrow}.\\ (s \in \mathsf{sat}(G) \wedge \forall f{:}\tau \in \Gamma.\ (\gamma_{\mathsf{a}} f, \gamma_{\mathsf{b}} f) \in \overline{G}(s')(\tau)) \implies (v_{\mathsf{a}}, v_{\mathsf{b}}) \in \overline{G}(s')(\tau')\end{array}$$

The interesting part here is: how do we know that if $\gamma_{\mathsf{a}}, \gamma_{\mathsf{b}}$ is pointwise related for labels in Γ, then so is $\gamma^2_{\mathsf{a}}, \gamma^2_{\mathsf{b}}$ for Γ, Γ_1? Recall that γ^2_{a} and γ^2_{b} extend γ_{a} and γ_{b} with the values exported by (the concrete loadings of) M^1_{a} and M^2_{b}, respectively. We have just shown, in the first case above, that these exported values are indeed related!

We conclude this section by stating the full modularity theorem.

Theorem 14 (Modularity of $\precsim_{\mathcal{TS}}$)**.**

$$\frac{\begin{array}{cc}\Gamma \vdash M^1_{\mathsf{b}} : \Gamma_1 & \Gamma, \Gamma_1 \vdash M^2_{\mathsf{b}} : \Gamma_2\\ \vdash M^1_{\mathsf{a}} : |\Gamma_1| & \vdash M^2_{\mathsf{a}} : |\Gamma_2|\\ \Gamma \vdash M^1_{\mathsf{a}} \precsim_{\mathcal{TS}} M^1_{\mathsf{b}} : \Gamma_1 & \Gamma, \Gamma_1 \vdash M^2_{\mathsf{a}} \precsim_{\mathcal{TS}} M^2_{\mathsf{b}} : \Gamma_2\end{array}}{\Gamma \vdash \mathsf{link}(M^1_{\mathsf{a}}, M^2_{\mathsf{a}}) \precsim_{\mathcal{TS}} \mathsf{link}(M^1_{\mathsf{b}}, M^2_{\mathsf{b}}) : \Gamma_1, \Gamma_2}$$

.

4.8 Proof of Transitivity

4.8.1 Overview

Let us begin with the end, the theorem we wish to prove.

Theorem 15 (Transitivity)**.**

$$\frac{|\Gamma| \vdash M_{\mathcal{T}} \precsim_{\mathcal{TI}} M_{\mathcal{I}} : |\Gamma'| \qquad |\Gamma| \vdash M_{\mathcal{I}} \precsim^{*}_{\mathcal{II}} M'_{\mathcal{I}} : |\Gamma'| \qquad \Gamma \vdash M'_{\mathcal{I}} \precsim_{\mathcal{IS}} M_{\mathcal{S}} : \Gamma'}{\Gamma \vdash M_{\mathcal{T}} \precsim_{\mathcal{TS}} M_{\mathcal{S}} : \Gamma'}$$

We derive this as a corollary of two lemmas, one about composing $\precsim_{\mathcal{TI}}$ with $\precsim_{\mathcal{IS}}$ and one about composing $\precsim_{\mathcal{II}}$ with $\precsim_{\mathcal{IS}}$:

Lemma 60 (Transitivity, Part 1)**.**

$$\frac{|\Gamma| \vdash M_{\mathcal{T}} \precsim_{\mathcal{TI}} M_{\mathcal{I}} : |\Gamma'| \qquad \Gamma \vdash M_{\mathcal{I}} \precsim_{\mathcal{IS}} M_{\mathcal{S}} : \Gamma'}{\Gamma \vdash M_{\mathcal{T}} \precsim_{\mathcal{TS}} M_{\mathcal{S}} : \Gamma'}$$

Lemma 61 (Transitivity, Part 2)**.**

$$\frac{|\Gamma| \vdash M_{\mathcal{I}} \precsim_{\mathcal{II}} M'_{\mathcal{I}} : |\Gamma'| \qquad \Gamma \vdash M'_{\mathcal{I}} \precsim_{\mathcal{IS}} M_{\mathcal{S}} : \Gamma'}{\Gamma \vdash M_{\mathcal{I}} \precsim_{\mathcal{IS}} M_{\mathcal{S}} : \Gamma'}$$

Note that we do not need to show transitivity of $\precsim_{\mathcal{II}}$ itself because, in order to obtain Theorem 15, we can simply iterate Lemma 61.

Thanks to our uniform setup, the proofs of Lemma 60 and Lemma 61 mirror each other. In fact, they are so similar that large parts of their formalization are identical. They also rouhgly follow the original transitivity proof of PBs (Section 3.9, but of course many details are necessarily different. One key simplification in PILS is that since our intermediate language is untyped, the complexity having to do with abstract types can be avoided. On the other hand, one key complication in PILS is due to **E**'s asymmetric small-step formulation of **E**.

Note that, as an alternative to Lemma 61, we could try to instead show the following:

$$\frac{|\Gamma| \vdash M_{\mathcal{T}} \precsim_{\mathcal{TI}} M'_{\mathcal{I}} : |\Gamma'| \qquad |\Gamma| \vdash M'_{\mathcal{I}} \precsim_{\mathcal{II}} M_{\mathcal{I}} : |\Gamma'|}{|\Gamma| \vdash M_{\mathcal{T}} \precsim_{\mathcal{TI}} M_{\mathcal{I}} : |\Gamma'|}$$

Together with Lemma 60, this would also yield Theorem 15. However, we choose to prove Lemma 61 because its statement is closer to that of Lemma 60: in each the second premise refers to an instance of the *typed* model. In the rest of this section we go into details of the proof of Lemma 60 only, since that of Lemma 61 is so similar.

Recall that the proof of transitivity for PBs consists of two parts. In the first, we construct a world W that relates the programs in question but has the wrong shape. In the second, we construct a world of the right shape and prove—with the help of a notion of isomorphism—that it relates the same programs as W. Here, for simplicity, we *directly* construct a world of the correct shape that relates the programs in question, *i.e.,* we skip the indirection through a theory of world morphisms.

We do not spell out all the details here but mainly present our definitions and lemma statements and sketch the proofs of some of the key lemmas.

4.8.2 Constructing the Local World

At the very outset, we are given a set of type names. We use this set, unchanged, to instantiate the second premise ($\Gamma \vdash M_{\mathcal{I}} \precsim_{\mathcal{IS}} M_{\mathcal{S}} : \Gamma'$). The first premise doesn't require any such set since it is phrased in terms of the untyped model $\precsim_{\mathcal{TI}}$. Hence the two premises give us local worlds $w_1 \in \mathrm{LWorld}^{\Omega_{\mathcal{TI}}.\mathsf{T}}$ and $w_2 \in \mathrm{LWorld}^{\Omega_{\mathcal{IS}}.\mathsf{T}}$, respectively. Based on these, we construct a local world $w \in \mathrm{LWorld}^{\Omega_{\mathcal{TS}}.\mathsf{T}}$. The difficulty, of course, is to do so such that w satisfies all the required properties (*e.g.,* stability).

Like for PBs (cf. Lemma 33 in Section 3.9), one of these key properties concerns the decomposition of states and global knowledges: given $G \in \mathbf{GK}_{w\uparrow}$ and a state $s \in w\uparrow.\mathsf{S}$, we must be able to decompose G into a global knowledge for $w_1\uparrow$ (say G_1) and one for $w_2\uparrow$ (say G_2) as well as decompose s into s_1 and s_2 such that

$$\overline{G_1}(s_1) \circ \overline{G_2}(s_2)(\tau) = \overline{G}(s)(\tau).$$

We start by defining w's transition system (the remaining components will be defined later on). Basically, we follow the same approach as in the proof for PBs, which was to define $w.\mathsf{S}$ as the product of $w_1\uparrow.\mathsf{S}$ and $w_2\uparrow.\mathsf{S}$. In PILS, however, the global part of a state contains some information that does not need to end up in w's state. More concretely, recall that any states $s_1 \in w_1\uparrow.\mathsf{S}$ and $s_2 \in w_2\uparrow.\mathsf{S}$ look as follows:

$$s_1 = (s_1^{\circ}, s_{\mathrm{lc}}^1) = ((s_1^{\mathcal{T}}, s_{\mathsf{ref}}^1, s_1^{\mathcal{I}}), s_{\mathrm{lc}}^1) \qquad s_2 = (s_2^{\circ}, s_{\mathrm{lc}}^2) = ((s_2^{\mathcal{I}}, s_{\mathsf{ref}}^2, s_2^{\mathcal{S}}), s_{\mathrm{lc}}^2),$$

Here,

- $s_1^{\mathcal{T}}$ is the unary component concerning $\mathcal{T}$ (register file and registry);
- $s_1^{\mathcal{I}}$ and $s_2^{\mathcal{I}}$ are the unary components concerning $\mathcal{I}$ (*i.e.,* carry no information)
- $s_2^{\mathcal{S}}$ is the unary component concerning $\mathcal{S}$ (label environment);
- $s_{\mathsf{ref}}^1 \in \mathsf{T}_{\mathsf{ref}}^{\mathcal{TI}}$ is the (untyped) partial bijection on global references between $\mathcal{T}$ and $\mathcal{I}$;
- $s_{\mathsf{ref}}^2 \in \mathsf{T}_{\mathsf{ref}}^{\mathcal{IS}}$ is the (typed) partial bijection on global references between $\mathcal{I}$ and $\mathcal{S}$;
- s_{lc}^1 is the local state from the unknown $w_1.\mathsf{S}$;
- s_{lc}^2 is the local state from the unknown $w_2.\mathsf{S}$.

We define w's states to be products of these states without the unary parts, because $\Omega_{\mathcal{TS}}.\mathsf{S}$ already provides those[9]:

$$w.\mathsf{T} := (\mathsf{T}_{\mathsf{ref}}^{\mathcal{TI}} \times w_1.\mathsf{T}) \times (\mathsf{T}_{\mathsf{ref}}^{\mathcal{IS}} \times w_2.\mathsf{T})$$

[9]It does, of course, not provide the unary part specific to the intermediate program, but that carries no information anyway. If it would be non-trivial, we would have to include it in $w.\mathsf{S}$.

Like for PBs, it it crucial that we embed $w_1\uparrow$ and $w_2\uparrow$'s global reference information in $w.\mathsf{S}$.

Given $s_1 \in w_1\uparrow$ and $s_2 \in w_2\uparrow$, we compose them to a $w\uparrow$ state, written $s_1 \diamond s_2$, in the following way:

Definition 28.

$$
\begin{aligned}
&(-) \diamond (-) \in w_1\uparrow.\mathsf{S} \to w_2\uparrow.\mathsf{S} \to w\uparrow.\mathsf{S} \\
&((s_1^{\mathcal{T}}, s^1_{\mathsf{ref}}, s_1^{\mathcal{I}}), s^1_{\mathrm{lc}}) \diamond ((s_2^{\mathcal{I}}, s^2_{\mathsf{ref}}, s_2^{\mathcal{S}}), s^2_{\mathrm{lc}}) := ((s_1^{\mathcal{T}}, s_{\mathsf{ref}}, s_2^{\mathcal{S}}), ((s^1_{\mathsf{ref}}, s^1_{\mathrm{lc}}), (s^2_{\mathsf{ref}}, s^2_{\mathrm{lc}}))) \\
&\qquad\qquad \text{where } s_{\mathsf{ref}} := \{\, (\tau, x_1, x_3) \mid \exists x_2.\ (x_1, x_2) \in s^1_{\mathsf{ref}} \wedge (\tau, x_2, x_3) \in s^2_{\mathsf{ref}} \,\}
\end{aligned}
$$

To decompose a $w\uparrow$ state, we ignore $\Omega_{\mathcal{TS}}$'s information on global references:

Definition 29.

$$
\begin{aligned}
&(-)_{(1)} \in w\uparrow.\mathsf{S} \to w_1\uparrow.\mathsf{S} \\
&((s_1^\circ, s_{\mathsf{ref}}, s_2^\circ), ((s^1_{\mathsf{ref}}, s_1), (s^2_{\mathsf{ref}}, s_2)))_{(1)} := ((s_1^\circ, s^1_{\mathsf{ref}}, 1), s_1)
\end{aligned}
$$

$$
\begin{aligned}
&(-)_{(2)} \in w\uparrow.\mathsf{S} \to w_2\uparrow.\mathsf{S} \\
&((s_1^\circ, s_{\mathsf{ref}}, s_2^\circ), ((s^1_{\mathsf{ref}}, s_1), (s^2_{\mathsf{ref}}, s_2)))_{(2)} := ((1, s^2_{\mathsf{ref}}, s_2^\circ), s_2)
\end{aligned}
$$

It is easy to see that $(s_1 \diamond s_2)_{(1)} = s_1$ and $(s_1 \diamond s_2)_{(2)} = s_2$ for any $s_1 \in w_1\uparrow$ and $s_2 \in w_2\uparrow$.

These definitions immediately yield the following nice properties, independent of how exactly the other components of w will be defined.

Lemma 62.

- $w\uparrow.\mathsf{vqh_a}(s_1 \diamond s_2) = w_1\uparrow.\mathsf{vqh_a}(s_1)$
- $w\uparrow.\mathsf{vqh_b}(s_1 \diamond s_2) = w_2\uparrow.\mathsf{vqh_b}(s_2)$
- $w\uparrow.\mathsf{cqh_a}(s_1 \diamond s_2) = w_1\uparrow.\mathsf{cqh_a}(s_1)$
- $w\uparrow.\mathsf{cqh_b}(s_1 \diamond s_2) = w_2\uparrow.\mathsf{cqh_b}(s_2)$
- $w\uparrow.\mathsf{rqh}(s_1 \diamond s_2)(\tau) = w_1\uparrow.\mathsf{rqh}(s_1) \circ w_2\uparrow.\mathsf{rqh}(s_2)(\tau)$

With the help of these operations, we now define the decomposition of a global knowledge for the yet-to-be-defined w. The idea is the same as in the PB transitivity proof: for each tuple τ, v_1, v_3 in the global knowledge, we craft a value $\mathbf{B}(\tau, v_1, v_3)$ "in the middle" (here: an $\mathcal{I}$ value) that serves three purposes:

1. It uniquely encodes the tuple (*i.e.,* $\mathbf{B}$ is injective).
2. It violates the $\mathsf{GoodFuns}$ condition (if τ is a function type), so it cannot possibly be related by $\mathbf{F}$.

3. It rules out certain cases in the proof of a transitivity property for $\mathbf{E}$ (Lemma 66). For instance, an $\mathcal{I}$ machine representing a call to such a function must not be able to produce an error.

$\mathbf{B}$ is defined using an arbitrary (but fixed) injective function $\mathbf{I} \in \mathsf{CTyF} \times \mathcal{T}.\mathbf{Val} \times \mathcal{S}.\mathbf{Val} \to \mathcal{I}.\mathbf{Val}$ that always returns a natural number value. We wrap $\mathbf{I}$'s result depending on the type τ such that $\mathbf{B}(\tau, v_1, v_3)$ has the right shape according to the value query handler for $\mathcal{I}$ from Figure 4.18. For instance, if $\tau = \tau_1 \to \tau_2$, then it is important that $\mathbf{B}(\tau, v_1, v_3)$ is a proper function value, *i.e.,* a closure with a fix term inside.

Definition 30.

$$\begin{array}{lcll} \mathbf{B} \in \mathsf{CTyF} \times \mathcal{T}.\mathbf{Val} \times \mathcal{S}.\mathbf{Val} \to \mathcal{I}.\mathbf{Val} & & & \\ \mathbf{B}(\tau \to \tau', v_1, v_3) & := & \langle \epsilon; \mathsf{fix}\, f(y,k).\, \mathsf{halt}\, n \rangle & \text{where } n := \mathbf{I}(\tau \to \tau', v_1, v_3) \\ \mathbf{B}(\forall \alpha.\, \tau, v_1, v_3) & := & \langle \epsilon; \Lambda k.\, \mathsf{halt}\, n \rangle & \text{where } n := \mathbf{I}(\forall \alpha.\, \tau, v_1, v_3) \\ \mathbf{B}(\mathbf{n}, v_1, v_3) & := & \langle \epsilon; n \rangle & \text{where } n := \mathbf{I}(\mathbf{n}, v_1, v_3) \end{array}$$

Using this, we first construct $G^{s_2}_{\{1\}}$ and $G^{s_1}_{\{2\}}$, and then close them over $\mathbf{F}$ and $w_1.\mathsf{NR}$ (respectively $w_2.\mathsf{NR}$) to ensure that they can be shown to be proper global knowledges later.

Definition 31. For $G \in w{\uparrow}.\mathsf{S} \to \mathsf{VRelF}_{\mathcal{T},\mathcal{S}}$ and $s_2 \in w_2{\uparrow}.\mathsf{S}$, we define $G^{s_2}_{(1)} \in w_1{\uparrow}.\mathsf{S} \to \mathsf{VRelF}_{\mathcal{T},\mathcal{I}}$ (untyped):

$$G^{s_2}_{\{1\}}(s_1) := \{\, (v_1, \mathbf{B}(\tau, v_1, v_3)) \mid \tau \notin w_2.\mathsf{NS} \wedge \exists s_1' \sqsubseteq s_1.\ \exists s_2' \sqsubseteq s_2.\ (v_1, v_3) \in G(s_1' \diamond s_2')(\tau) \,\}$$

$$G^{s_2}_{(1)} := [w_1{\uparrow}]_{G^{s_2}_{\{1\}}}$$

Definition 32. For $G \in w{\uparrow}.\mathsf{S} \to \mathsf{VRelF}_{\mathcal{T},\mathcal{S}}$ and $s_1 \in w_1{\uparrow}.\mathsf{S}$, we define $G^{s_1}_{(2)} \in w_2{\uparrow}.\mathsf{S} \to \mathsf{VRelF}_{\mathcal{I},\mathcal{S}}$ (typed).

$$G^{s_1}_{\{2\}}(s_2)(\tau) := \{\, (\mathbf{B}(\tau, v_1, v_3), v_3) \mid \tau \notin w_2.\mathsf{NS} \wedge \exists s_1' \sqsubseteq s_1.\ \exists s_2' \sqsubseteq s_2.\ (v_1, v_3) \in G(s_1' \diamond s_2')(\tau) \,\}$$

$$G^{s_1}_{(2)} := [w_2{\uparrow}]_{G^{s_1}_{\{2\}}}$$

We can now fully define w. As its order we choose the lexicographic product of w_1 and w_2's order. (This will be justified in the proof of Lemma 66.)

Definition 33.

$$\begin{array}{lcl} s^1_{\mathsf{ref}} \setminus_{[1]} s_{\mathsf{ref}} & := & \{\, (v_1, v_2) \in s^1_{\mathsf{ref}} \mid \nexists \tau, v_3.\ (\tau, v_1, v_3) \in s_{\mathsf{ref}} \,\} \\ s^2_{\mathsf{ref}} \setminus_{[2]} s_{\mathsf{ref}} & := & \{\, (\tau, v_2, v_3) \in s^2_{\mathsf{ref}} \mid \nexists v_1.\ (\tau, v_1, v_3) \in s_{\mathsf{ref}} \,\} \end{array}$$

Definition 34.

$$
\begin{array}{lcl}
w.\mathsf{T} & := & (\mathsf{T}^{\mathcal{TI}}_{\mathsf{ref}} \times w_1.\mathsf{T}) \times (\mathsf{T}^{\mathcal{IS}}_{\mathsf{ref}} \times w_2.\mathsf{T}) \\[2ex]
w.\mathsf{O} & := & w_2.\mathsf{O} \,\#\, w_1.\mathsf{O} \\[2ex]
w.\mathsf{C}(G)(s) & := & \{\, (c^{\mathcal{T}}_{\mathsf{ref}} \cdot c^{\mathcal{T}}, d^{\mathcal{S}}_{\mathsf{ref}} \cdot d^{\mathcal{S}}) \mid \\
 & & \quad \exists s_1, s_2, s_{\mathsf{ref}}.\ s = s_1 \diamond s_2 \wedge s = ((\_, s_{\mathsf{ref}}, \_), \_) \wedge \\
 & & \quad \exists s^{\mathcal{T}}_1, s^1_{\mathsf{ref}}, s^{\mathcal{I}}_1, s^1_{\mathrm{lc}}.\ s_1 = ((s^{\mathcal{T}}_1, s^1_{\mathsf{ref}}, s^{\mathcal{I}}_1), s^1_{\mathrm{lc}}) \wedge \\
 & & \quad \exists s^{\mathcal{I}}_2, s^2_{\mathsf{ref}}, s^{\mathcal{S}}_2, s^2_{\mathrm{lc}}.\ s_2 = ((s^{\mathcal{I}}_2, s^2_{\mathsf{ref}}, s^{\mathcal{S}}_2), s^2_{\mathrm{lc}}) \wedge \\
 & & \quad \exists (c^{\mathcal{T}}_{\mathsf{ref}}, c^{\mathcal{I}}_{\mathsf{ref}}) \in C^{\mathcal{TI}}_{\mathsf{ref}}(s^1_{\mathsf{ref}} \setminus_{[1]} s_{\mathsf{ref}})(\overline{G^{s_2}_{(1)}}(s_1)). \\
 & & \quad \exists (d^{\mathcal{I}}_{\mathsf{ref}}, d^{\mathcal{S}}_{\mathsf{ref}}) \in C^{\mathcal{IS}}_{\mathsf{ref}}(s^2_{\mathsf{ref}} \setminus_{[2]} s_{\mathsf{ref}})(\overline{G^{s_1}_{(2)}}(s_2)). \\
 & & \quad \exists (c^{\mathcal{T}}, c^{\mathcal{I}}) \in w_1.\mathsf{C}(G^{s_2}_{(1)})(s_1). \\
 & & \quad \exists (d^{\mathcal{I}}, d^{\mathcal{S}}) \in w_2.\mathsf{C}(G^{s_1}_{(2)})(s_2). \\
 & & \quad c^{\mathcal{I}}_{\mathsf{ref}} \cdot c^{\mathcal{I}} = d^{\mathcal{I}}_{\mathsf{ref}} \cdot d^{\mathcal{I}} \wedge \\
 & & \quad \forall h \neq \bot.\ \mathrm{dom}(h) \subseteq \{l_2 \mid \exists l_1.\ (l_1, l_2) \in s^1_{\mathsf{ref}}\} \implies \\
 & & \quad \forall c \in \mathbf{core}.\ \exists m \in \mathbf{real}(c \cdot (h, \emptyset) \cdot c^{\mathcal{I}}) \,\} \\[2ex]
w.\mathsf{NS} & := & w_2.\mathsf{NS} \\
w.\mathsf{NR}(G)(s)(\mathbf{n}) & := & \{\, (v_1, v_3) \in \overline{G^{s_{(2)}}_{(1)}}(s_{(1)}) \circ \overline{G^{s_{(1)}}_{(2)}}(s_{(2)})(\mathbf{n}) \mid \mathbf{n} \in w.\mathsf{NS} \,\}
\end{array}
$$

The construction of its configuration relation $w.\mathbf{C}$ closely follows that of the heap relation in the PB proof. The last condition basically asserts that $c^{\mathcal{I}}$ is disjoint from the intermediate locations in s^1_{ref} and from any core.

Lemma 63. If $G \in \mathbf{GK}_{w\uparrow}$, then

- $\forall s_2.\ G^{s_2}_{(1)} \in \mathbf{GK}_{w_1\uparrow}$, and
- $\forall s_1.\ G^{s_1}_{(2)} \in \mathbf{GK}_{w_2\uparrow}$.

Lemma 64. If $G \in \mathbf{GK}_{w\uparrow}$, then

$$\overline{G^{s_2}_{(1)}}(s_1) \circ \overline{G^{s_1}_{(2)}}(s_2)(\tau) = \overline{G}(s_1 \diamond s_2)(\tau).$$

Proof. The $\supseteq$ direction is fairly straightforward by induction on the value closure (which is a least fixed point). One interesting point has to do with type names $\mathbf{n}$ that lie outside the control of w, *i.e.,* $\mathbf{n} \notin w.\mathsf{NS}$. If values v_1, v_3 are related at such a type, then we establish

$$(v_1, v_3) \in \overline{G^{s_2}_{(1)}}(s_1) \circ \overline{G^{s_1}_{(2)}}(s_2)(\tau)$$

by showing

$$(v_1, \mathbf{B}(\mathbf{n}, v_1, v_3)) \in \overline{G^{s_2}_{\{1\}}}(s_1) \quad \wedge \quad (\mathbf{B}(\mathbf{n}, v_1, v_3), v_3) \in \overline{G^{s_1}_{\{2\}}}(s_2)(\mathbf{n}).$$

In order for this to work, it is crucial that the *untyped* value closure operator also includes a "type name" case (cf. Figure 4.13 in Section 4.3), and that this case does *not* restrict the value on the target side (here v_1) as we know nothing about it.

The $\subseteq$ direction, like for PBs, is proven by induction on the left value closure. Formally, the statement that we show by induction is

$$\overline{G^{s_2}_{(1)}} \subseteq X,$$

where $X(s_1) := \{\, (v_1, v_2) \mid \forall \tau, v_3.\ (v_2, v_3) \in \overline{G^{s_1}_{(2)}}(s_2)(\tau) \implies \overline{G}(s_1 \diamond s_2)(\tau) \,\}$.

There is an interesting PILS twist here, too. Consider the case where inverting the value closure's generating function yields the "unit" case. Then we know $v_1 \in \Omega_{\mathcal{TI}}.\mathsf{vqh_a}(\_)(\mathsf{unit})$ and $v_2 \in \Omega_{\mathcal{TI}}.\mathsf{vqh_b}(\_)(\mathsf{unit})$, *i.e.,* v_1 is an arbitrary machine word and v_2 is $\langle\rangle$. However, since the $\mathcal{TI}$ model is untyped, we know nothing about τ yet. So when we invert $\overline{G^{s_1}_{(2)}}(s_2)(\tau)$, there are many possible cases to deal with—and several of them cannot be ruled out. For instance, $(v_2, v_3) \in \overline{G^{s_1}_{(2)}}(s_2)(\tau)$ may hold due to the "roll" case where $v_2 \in \Omega_{\mathcal{TI}}.\mathsf{vqh_b}(\_)(\mathsf{roll}\ v_2')$. Note that this is not in contradiction to $v_2 = \langle\rangle$ because $\mathcal{I}$ does not have an explicit roll construct. Hence all we know is $v_2 = v_2'$ and $(v_2', v_3') \in \overline{G^{s_1}_{(2)}}(s_2)(\tau)$, so we are going in circles. For this reason, unlike in the PB proof, we actually have to do a second induction, this time on $\overline{G^{s_1}_{(2)}}$.

Unrelatedly, and similar to PBs, the proof of this lemma is the only place where we exploit the $\mathsf{GoodFuns}$ restriction on $\mathbf{F}$ to infer that

$$\mathbf{F}_{w_1\uparrow}(G^{s_2}_{(1)})(s_1) \circ G^{s_1}_{\{2\}}(s_2)(\tau)$$

and

$$G^{s_2}_{\{1\}}(s_1) \circ \mathbf{F}_{w_2\uparrow}(G^{s_1}_{(2)})(s_2)(\tau)$$

are empty (for function types). □

The next lemma talks about the configuration relations. We did not have such a lemma explicitly in the PB proof, but it was hidden in establishing the world isomorphism. Here we have an additional condition, though, which states that the $\mathcal{I}$ configuration c_2 can be realized whenever composed with a core. This should be thought of as a sanity check.

Lemma 65. If $G \in \mathbf{GK}_{w\uparrow}$, then:

$$\begin{aligned}(c_1, c_3) \in w\uparrow.\mathsf{C}(G)(s_1 \diamond s_2) \iff\ & \exists c_2.\\ & (\forall e \in \mathbf{core}.\ \exists m \in \mathbf{real}(e \cdot c_2)) \wedge\\ & (c_1, c_2) \in w_1\uparrow.\mathsf{C}(G^{s_2}_{(1)})(s_1) \wedge (c_2, c_3) \in w_2\uparrow.\mathsf{C}(G^{s_1}_{(2)})(s_2)\end{aligned}$$

Of course, this property does not hold for arbitrary $w\uparrow$ states but only for those of the form $s_1 \diamond s_2$.

One of the most important and also most involved parts is the proof of transitivity of $\mathbf{E}$. We first prove the following lemma, concerning the case where the ϕ argument is non-trivial. The other case, Lemma 67, follows from that.

				err	ret	step		
						call	rec	
							log	phys
err	0			$\lightning_1$	$\lightning_1$	$\lightning_1$	$\lightning_1$	$\lightning_1$
	+			err	$\lightning_2$	$\lightning_3$	(ind)	(ind)
ret	0			err	ret*	$\lightning_4$	err	err
	+			err	$\lightning_5$	$\lightning_6$	(ind)	(ind)
step	0	call		err	$\lightning$	call*	err	err
		rec	log	err	log*	log*	log*	log*
			phys	err	$\lightning$	$\lightning$	log*	phys*
	+			err	$\lightning$	$\lightning$	(ind)	(ind)

Figure 4.23: Overview of the proof of transitivity of $\mathbf{E}$ (Lemma 66)

Definition 35.

$$\mathbf{B_k} := \langle \epsilon; \mathsf{cont}\ y.\, \mathsf{halt}\, 0 \rangle$$

Lemma 66. If

- $(e_1, e_2) \in \mathbf{E}_{w_1\uparrow}(i)(\mathbf{k}_1, \mathbf{B_k})(G^{s_2'}_{(1)})(s_1)(s_1')(\phi)$
- $(e_2, e_3) \in \mathbf{E}_{w_2\uparrow}(j)(\mathbf{B_k}, \mathbf{k}_3)(G^{s_1'}_{(2)})(s_2)(s_2')(\phi)(\tau)$
- $\mathbf{real}(e_2) \neq \emptyset$
- $s_1' \sqsupseteq s_1$
- $s_2' \sqsupseteq s_2$
- $\phi \neq \bot$

then $(e_1, e_3) \in \mathbf{E}_{w\uparrow}(\langle j, i \rangle)(\mathbf{k}_1, \mathbf{k}_3)(G)(s_1 \diamond s_2)(s_1' \diamond s_2')(\phi)(\tau)$.

The lemma is stated in terms of a very particular continuation for the intermediate program, namely $\mathbf{B_k}$. It is defined above as a continuation that is immediately halted (after one step). This is critical in ruling out certain cases in the proof.

Note that the Lemma's third condition is harmless, because, due to $\phi \neq \bot$, we are only interested in fully configured e_2's, which clearly are realizable. Without this condition, however, the first two assumptions might hold trivially, while the goal might not hold trivially (e_2 might not be realizable, while e_1 and e_3 might).

Proof. By coinduction. Suppose $G \in \mathbf{GK}_{w\uparrow}$ and

$$(m_1, m_3) \in \mathsf{cfg}(w\uparrow.\mathsf{C})(G)(s_1' \diamond s_2')(\phi)(e_1, e_3)(\emptyset, \emptyset)(\eta_1, \eta_3).$$

We can decompose this using Lemma 65 (with an empty frame in the middle) such that for any m_2 and e_2 with $m_2 \in \mathbf{real}(e_2)$ we have:

$$(m_1, m_2) \in \mathsf{cfg}(w_1\uparrow.\mathsf{C})(G_{(1)}^{s_2'})(s_1')(\phi)(e_1, e_2)(\emptyset, \emptyset)(\eta_1, \emptyset) \tag{4.1}$$

$$(m_2, m_3) \in \mathsf{cfg}(w_2\uparrow.\mathsf{C})(G_{(2)}^{s_1'})(s_2')(\phi)(e_2, e_3)(\emptyset, \emptyset)(\emptyset, \eta_3) \tag{4.2}$$

Note that by construction we have $G_{(1)}^{s_2'} = \mathbf{F}_{w_1\uparrow}(G_{(1)}^{s_2'}) \cup w_2\uparrow.\mathsf{NR}(G_{(1)}^{s_2'}) \cup G_{\{1\}}^{s_2'}$ (and similarly for $G_{(2)}^{s_1'}$. Hence we can apply the external call lemma (Lemma 55 to the two main premises to obtain the following:

$$(e_1, e_2) \in \mathbf{E}_{w_1\uparrow}^{G_{\{1\}}^{s_2'}}(i)(\mathbf{k}_1, \mathbf{B}_\mathbf{k})(G_{(1)}^{s_2'})(s_1)(s_1')(\phi) \tag{4.3}$$

$$(e_2, e_3) \in \mathbf{E}_{w_2\uparrow}^{G_{\{2\}}^{s_1'}}(j)(\mathbf{B}_\mathbf{k}, \mathbf{k}_3)(G_{(2)}^{s_1'})(s_2)(s_2')(\phi)(\tau) \tag{4.4}$$

We now instantiate 4.3 with 4.1 and do a case analysis on the result. The possible cases make up the vertical axis of Figure 4.23. There are three main cases, but in each we make further distinctions. Moreover, we also (instantiate and) analyze 4.4 each time, which yields the additional cases shown on the horizontal axis.

In each of the three main cases of 4.3 (ERR, RET, STEP on the vertical axis), we are given a silent reduction sequence $m_2 \hookrightarrow^* m_2'$ and do an induction on it. "0" in the figure refers to the base case, where the sequence is empty and thus $m_2 = m_2'$. "+" refers to the case where the sequence is non-empty and where the inductive hypothesis plays a role ("ind"). Stars (*e.g.,* RET*) indicate that the argument in the respective case makes use of the coinductive hypothesis. $\lightning$ indicates impossible situations, where we derive a contradiction.

Here we sketch mainly the (vertical) ERR and RET cases. We use the STEP case to explain the choice of the well-founded order $w.\mathsf{O}$.

Case ERR. The intermediate program is erroneous: $m_2 \hookrightarrow^* m_2'$ with $m_2' \in \mathbf{error}$. We show the goal by also using the ERR clause, *i.e.,* we show $m_3 \hookrightarrow^* m_3'$ for some $m_3' \in \mathbf{error}$.

We do so by induction on the reduction sequence $m_2 \hookrightarrow^* m_2'$. Note that $m_2 = m_2'$ is ruled out by 4.2, because an erroneous machine is not realizable ($\lightning_1$). So there is at least one step: $m_2 \hookrightarrow m_2'' \hookrightarrow^* m_2'$.

Let us now instantiate 4.4 with 4.2 and do another case analysis on the result.

Subcase ERR. The source machine m_3 is also erroneous, so we are done by appeal to ERR.

Subcase RET. The intermediate machine m_2 represents a return to "bad" continuation $\mathbf{B_k}$. Therefore, after one step, it reaches a machine that is halted, and moreover, not an error. This contradicts that m_2 eventually produces an error ($\lightning_2$).

Subcase STEP. We apply $m_2 \hookrightarrow m_2''$ and must deal with two further cases.

In REC, m_3 steps to or equals a machine that, by the inductive hypothesis, eventually produces an error. The goal follows immediately.

In CALL, m_2'' realizes a function call and thus $m_2'' \notin \mathbf{error}$. Moreover, by construction of $G^{s_1'}_{\{2\}}$, we know that the called function is "bad", so m_2'' is actually halted and thus $m_2'' = m_2'$. This contradicts $m_2' \in \mathbf{error}$ ($\lightning_3$).

Case RET. The programs return to their continuations, with related values. In particular, $m_2 \hookrightarrow^* m_2'$ where m_2' returns some value v_2 to the "bad" continuation $\mathbf{B_k}$. We show the goal by induction on this reduction sequence, eventually using either the ERR or RET clause.

Suppose $m_2 = m_2'$. Unfortunately, this does not contradict anything. So we instantiate 4.4 with 4.2 and do another case analysis on the result.

Subcase ERR. As before.

Subcase RET. Using Lemmas 64 and 65, it is easy to compose all the information we have at this point and establish the goal using RET.

Subcase STEP. Since m_2 is a return to $\mathbf{B_k}$, it takes one step to a particular halted machine, say $m_2 \hookrightarrow m_2^\circ$. For this step, the subcase yields two further cases.

In REC, m_2° is again related to some source program by $\mathbf{E}^{G^{s_1'}_{\{2\}}}_{w_2\uparrow}$. It is easy to show that this can only hold due to the ERR case: $m_2^\circ \in \mathbf{halted}$ directly contradicts the STEP case; it also contradicts the RET case, because an $\mathcal{I}$ machine realizing a return is never halted.

In CALL, m_2° realizes a function call. It is easy to see that this contradicts m_2° being halted due to $\mathbf{B_k}$ ($\lightning_4$).

This was the base case of the induction. Now suppose there is at least one step: $m_2 \hookrightarrow m_2'' \hookrightarrow^* m_2'$. Here too, we instantiate 4.4 with 4.2 and do another case analysis on the result.

Subcase ERR. As before.

Subcase RET. Then m_2 is a return to $\mathbf{B_k}$. Its immediate successor m_2'' is therefore halted by construction of $\mathbf{B_k}$. Hence m_2'' must be m_2', which is also a return. This is a contradiction: an $\mathcal{I}$ machine cannot realize a return and be halted at the same time ($\lightning_5$).

Subcase STEP. We apply $m_2 \hookrightarrow m_2''$ and must deal with two further cases.

In REC, we can easily conclude with the help of the inductive hypothesis.

In CALL, m_2'' realizes a call to a "bad" function and thus is halted. This is in contradiction to m_2' realizing a return, as in $\mathcal{I}$ a return is never halted ($\not\downarrow_6$).

Case STEP. Along the same lines as the previous two, but more tedious. As can be seen in Figure 4.23, we also need to further distinguish between stuttering and non-stuttering. Instead of going into the details, let us use this case to illustrate our choice of $w.\mathsf{O}$ as the lexicographic product of $w_2.\mathsf{O}$ and $w_1.\mathsf{O}$. While it is natural that we choose a product of the two, the two obvious questions are: why lexicographic, and why $w_2.\mathsf{O}$ before $w_1.\mathsf{O}$?

To answer this, let us consider the REC case and assume that there is no stuttering, *i.e.,* the recursive occurrence of $\mathbf{E}$ uses an *unknown* $i' \in w_1.\mathsf{O}$. Further assume that 4.4 yields the REC case as well, but that there stuttering happens. Hence the recursive occurence of $\mathbf{E}$ necessarily uses a $j' \in w_2.\mathsf{O}$ that is *smaller than* the initial j (from the lemma statement). Obviously we want to show the goal by appealing to the REC ourselves. And due to the stuttering in 4.4, we are forced to stutter as well. Fortunately, we can do so just fine by picking the budget $\langle j', i' \rangle$ since $j' < j$ implies $\langle j', i' \rangle < \langle j, i \rangle$. Note that no other product would work because i' may be greater than i.

If the intermediate program *does* stutter, witnessed by some $i' < i$, there is no reason to look into 4.4 at all. We can simply show the goal by stuttering immediately, leaving the source program untouched. This means choosing the new budget to be $\langle j, i' \rangle$, which is of course smaller than $\langle j, i \rangle$ due to $i' < i$ in that case.

□

Lemma 67. If

- $(e_1, e_2) \in \mathbf{E}_{w_1\uparrow}(i)(\mathbf{k}_1, \mathbf{B_k})(G^{s_2'}_{(1)})(s_1)(s_1')(\bot)$
- $(e_2, e_3) \in \mathbf{E}_{w_2\uparrow}(j)(\mathbf{B_k}, \mathbf{k}_3)(G^{s_1'}_{(2)})(s_2)(s_2')(\bot)(\tau)$
- $e_2 \in \mathbf{core}$
- $s_1' \sqsupseteq s_1$
- $s_2' \sqsupseteq s_2$

then $(e_1, e_3) \in \mathbf{E}_{w\uparrow}(\langle j, i \rangle)(\mathbf{k}_1, \mathbf{k}_3)(G)(s_1 \diamond s_2)(s_1' \diamond s_2')(\bot)(\tau)$.

Note that the third condition ($e_2 \in \mathbf{core}$) is harmless, as we are only interested in such e_2's. Without this condition, however, the first two assumptions might hold trivially, while the goal might not hold trivially (e_2 might not be a core while e_1 and e_3 might).

Proof. Straightforward, using Lemmas 65 and 66. □

The key thing left to do, is proving stability of w. This is indeed very tedious but closely follows its PB counterpart (there are no new ideas needed). With that, we can easily establish the final property.

Theorem 16.

$$\frac{|\Gamma| \vdash M_1 \precsim_{\mathcal{TI}} M_2 : |\Gamma'| \quad \Gamma \vdash M_2 \precsim_{\mathcal{IS}} M_3 : \Gamma'}{\Gamma \vdash M_1 \precsim_{\mathcal{TI}} M_3 : \Gamma'}$$

4.8.3 Discussion

As for PBs, one of the main complexities in the PILS transitivity proof lies in dealing with an ambiguity regarding reference allocation: while in one of the two given proofs, an allocation of the middle program may be treated as public (extending the global state), the same allocation may be treated as private (extending the local state) in the other proof. This is a result of transitivity being proven for completely arbitrary local worlds! One might wonder if we could not simplify matters significantly by resorting to an instrumentation of the IL that makes the choice of public vs. private allocation explicit in the program code. It seems to us that this approach only makes sense if one is willing to a priori decide on all subsequent optimizations. The issue is that a later added optimization might, for instance, figure out that some reference is never used and therefore can be removed. Such can only be proven correct if the reference was allocated as private, which it may not have been. For the sake of modularity, we therefore believe it is better to bite the bullet and deal with the ambiguity issue semantically, *e.g.,* in the way we did.

Chapter 5

The Pilsner Compiler and Its Verification

Contents

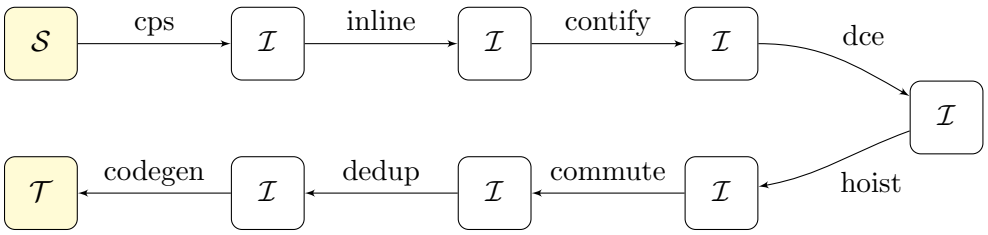


Figure 5.1: Structure of the Pilsner compiler

5.1 Overview

Using PILS, we have proven *in Coq* the correctness of two compilers from $\mathcal{S}$ to $\mathcal{T}$: Pilsner and Zwickel. Pilsner's structure is depicted in Figure 5.1. It uses $\mathcal{I}$ as its CPS-based intermediate language and performs several code transformations. Zwickel, on the other hand, is more simplistic: it directly translates $\mathcal{S}$ code into $\mathcal{T}$ code in a straightforward way, similar to Hur and Dreyer's one-pass compiler [33]. In particular, Zwickel neither uses an intermediate language nor performs any CPS transformation. In this chapter, we focus mostly on Pilsner, because it is by far the more interesting compiler.

Given a source module, Pilsner first translates it to $\mathcal{I}$ via a CPS transformation. It also takes care to alpha-rename all bound variables such that in the resulting $\mathcal{I}$ module, every variable is bound at most once. This *uniqueness condition* simplifies the implementation of most of the subsequent transformation passes, as one does not

have to worry about accidental variable capturing when rearranging code. Another nice characteristic of the produced intermediate code (not of $\mathcal{I}$ per se) is that continuations are used in an affine fashion [39], *i.e.,* called at most once. This property is preserved by all other transformations at the intermediate level and enables a more efficient treatment of continuation variables compared to ordinary variables in the code generation pass.

At the intermediate level, Pilsner performs six "optimizations". It first inlines selected top-level functions. For instance, if a module defines $F = \mathsf{fix}\, f(y, k).\, e$, then a call to F inside a subsequently defined function will be rewritten as follows:

$$F\ x\ k' \quad \rightsquigarrow \quad e[F/f][x/y][k'/k]$$

(If the function is recursive, *i.e.,* if f is used in e, this is essentially just an unrolling.) Since inlining destroys the uniqueness property of bound variables, we immediately follow it with a "freshening" pass that re-establishes uniqueness.

Next comes contification, which converts certain functions to continuations. Subsequently, Pilsner performs a simple dead code (and variable) elimination, rewriting $\mathsf{let}\ x = a\ \mathsf{in}\ e$ to e whenever x does not occur in e. This is justified because, in our IL, evaluation of a pure expression a does not have any observable side effects. In the same manner, it also eliminates unused read operations and unused allocations (but not write operations because that would be unsound). Following DCE, it hoists let-bindings out of function and continuation definitions, subject to some syntactic constraints. For example:

$$\begin{array}{rl} & \mathsf{let}\ f = (\mathsf{fix}\, f(y, k).\, \mathsf{let}\ z = x.1\ \mathsf{in}\ e)\ \mathsf{in}\ e' \\ \rightsquigarrow & \mathsf{let}\ z = x.1\ \mathsf{in}\ \mathsf{let}\ f = (\mathsf{fix}\, f(y, k).\, e)\ \mathsf{in}\ e' \end{array}$$

if x is none of f, y, k. This avoids recomputation of the projection each time f is called.

Next comes a pass that commutes let-bindings (where possible) in order to group together bindings that assign names to the same expression. For instance:

$$\begin{array}{rl} & \mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ \mathsf{let}\ z = a\ \mathsf{in}\ e \\ \rightsquigarrow & \mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ z = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ e \end{array}$$

The last IL transformation, deduplication, gets rid of such consecutive duplicate bindings by rewriting the above expression as follows:

$$\rightsquigarrow \quad \mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ e[x/z]$$

This can be seen as a common subexpression elimination.

Code generation, the final pass in the chain, translates to the machine language $\mathcal{T}$. Recall that there are three kinds of "variables" in $\mathcal{I}$: term variables x, continuation variables k, and labels F. Labels are translated to absolute addresses according to the import table. Term variable accesses are translated to lookups (based on position) in a linked list on the heap, pointed to by the env register. Functions are converted to

closures, *i.e.,* pairs of environment and code pointer (module-level functions simply have an empty environment), which live on the heap. A closure's environment is loaded into the env register when the function is called. Finally, continuations are allocated on the stack. Accordingly, continuation variable accesses are translated to lookups (based on position) on the stack, with the side effect that the continuation in question, as well as all more-recently defined ones (above it on the stack), are popped. This is safe because the affinity property mentioned earlier ensures that they won't be needed anymore.

5.2 From $\mathcal{S}$ to $\mathcal{I}$: CPS Transformation

5.2.1 Definition

The CPS transformation translates from $\mathcal{S}$ to $\mathcal{I}$. Figure 5.2 show its definition. The module-level translation *cps* is shown right at the top. It is defined in terms of an auxiliary function *mcps*, which recursively uses *fcps* to translate top-level functions. *fcps* itself relies on *ecps* to translate expressions, whose definition is continued in Figures 5.3 and 5.4.

Besides performing a fairly standard CPS conversion, our transformation also makes sure that in the resulting expresssion no variable is bound twice by always choosing a fresh one. To do so, it uses injections *tvar* and *kvar* that map numbers to variables. The various translation functions take as input a number n and assume that any variable corresponding to a number greater or equal n is unused. Whenever they introduce a bound variable, they will pick one of these unused variables. To maintain the set of unused ones, the translations return an updated number along with the converted program.

When the expression translation *ecps* introduces a bound variable y in correspondence to a bound variable x in the source program, this correspondence is remembered in the ϕ argument such that, in the recursion on subexpressions, the translation knows what to do when it hits upon a free occurrence of x.

The last argument of *ecps* is a meta-level continuation κ. It takes a number (the next free variable) and a term variable or label (the input to the continuation) and returns an $\mathcal{I}$ expression and an updated variable number. We use meta-level functions rather than object-level continuations in order to avoid creating too many redexes (this could be optimized further, tough) [39].

Let us look at a few cases.

- The translation of a label F simply passes the label to the meta-level continuation κ.
- The translation of a source variable x does the same—except that it first renames the variable according to ϕ.
- The translation of a number m essentially just passes the number to κ. To do so, it introduces a let-binding, for which it uses the fresh term variable

$$
\begin{array}{lcl}
\multicolumn{3}{l}{cps \in \mathcal{S}.\mathbf{Mod} \rightharpoonup \mathcal{I}.\mathbf{Mod}} \\
cps(M) & = & mcps(0)(M) \\
 & & \\
\multicolumn{3}{l}{mcps \in \mathbb{N} \to \mathcal{S}.\mathbf{Mod} \rightharpoonup \mathcal{I}.\mathbf{Mod}} \\
mcps(n)(\epsilon) & = & \epsilon \\
mcps(n)((F, \mathbf{v}), M) & = & let\ (n', v') := fcps(n)(\mathbf{v})\ in\ (F, v'), mcps(n')(M) \\
 & & \\
\multicolumn{3}{l}{fcps \in \mathbb{N} \to \mathcal{S}.\mathbf{Val} \rightharpoonup \mathsf{Exp}_{\mathcal{I}}} \\
fcps(n)(\mathsf{fix}\, f(y).\, e) & = & let\ f' := tvar(n)\ in \\
 & & let\ y' := tvar(n+1)\ in \\
 & & let\ k := kvar(n+2)\ in \\
 & & let\ \phi' := id[f'/f][y'/y]\ in \\
 & & let\ (n_1, e') := ecps(e)(n+3)(\phi')(\boldsymbol{\lambda} n_2, x.\ (n_2, k\ x))\ in \\
 & & (n_1, \mathsf{fix}\, f'(y', k).\, e') \\
fcps(n)(\Lambda.\, e) & = & let\ k := kvar(n)\ in \\
 & & let\ (n_1, e') := ecps(e)(n+1)(\phi)(\boldsymbol{\lambda} n_2, x.\ (n_2, k\ x))\ in \\
 & & (n_1, \Lambda k.\, e') \\
 & & \\
ret\ (f)\ (n, e) & = & (n, f(e)) \\
 & & \\
\multicolumn{3}{l}{ecps \in \mathsf{Exp}_{\mathcal{S}} \to \mathbb{N} \to (\mathsf{Var} \to \mathrm{TVar} \cup \mathsf{Lbl}) \to (\mathbb{N} \to \mathrm{TVar} \cup \mathsf{Lbl} \to \mathbb{N} \times \mathsf{Exp}_{\mathcal{I}}) \to \mathbb{N} \times \mathsf{Exp}_{\mathcal{I}}} \\
ecps(F)(n)(\phi)(\kappa) & = & \kappa(n)(F) \\
ecps(x)(n)(\phi)(\kappa) & = & \kappa(n)(\phi(x)) \\
ecps(\langle\rangle)(n)(\phi)(\kappa) & = & ret\ (\boldsymbol{\lambda} e.\ \mathsf{let}\ tvar(n) = \langle\rangle\ \mathsf{in}\ e)\ (\kappa(n+1)(tvar(n))) \\
ecps(m)(n)(\phi)(\kappa) & = & ret\ (\boldsymbol{\lambda} e.\ \mathsf{let}\ tvar(n) = m\ \mathsf{in}\ e)\ (\kappa(n+1)(tvar(n))) \\
ecps(\mathsf{input})(n)(\phi)(\kappa) & = & ret\ (\boldsymbol{\lambda} e.\ tvar(n) \leftarrow \mathsf{input};\ e)\ (\kappa(n+1)(tvar(n))) \\
ecps(\mathsf{roll}\ e)(n)(\phi)(\kappa) & = & ecps(e)(n)(\phi)(\kappa) \\
ecps(\mathsf{unroll}\ e)(n)(\phi)(\kappa) & = & ecps(e)(n)(\phi)(\kappa) \\
ecps(\mathsf{pack}\ e)(n)(\phi)(\kappa) & = & ecps(e)(n)(\phi)(\kappa)
\end{array}
$$

Figure 5.2: CPS transformation (part 1 of 3).

$$
\begin{array}{l}
ecps(\mathsf{unpack}\ y\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2)(n)(\phi)(\kappa) = \\
\quad ecps(e_1)(n)(\phi)(\boldsymbol{\lambda}(n', x_1). \\
\quad\quad ecps(e_2)(n')(\phi[x_1/y])(\kappa))
\end{array}
$$

$$
\begin{array}{l}
ecps(e.1)(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad ret\ (\boldsymbol{\lambda}e'.\ \mathsf{let}\ tvar(n') = x.1\ \mathsf{in}\ e')\ (\kappa(n'+1)(tvar(n'))))
\end{array}
$$

$$
\begin{array}{l}
ecps(e.2)(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad ret\ (\boldsymbol{\lambda}e'.\ \mathsf{let}\ tvar(n') = x.2\ \mathsf{in}\ e')\ (\kappa(n'+1)(tvar(n'))))
\end{array}
$$

$$
\begin{array}{l}
ecps(\mathsf{inl}\ e)(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad ret\ (\boldsymbol{\lambda}e'.\ \mathsf{let}\ tvar(n') = \mathsf{inl}\ x\ \mathsf{in}\ e')\ (\kappa(n'+1)(tvar(n'))))
\end{array}
$$

$$
\begin{array}{l}
ecps(\mathsf{inr}\ e)(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad ret\ (\boldsymbol{\lambda}e'.\ \mathsf{let}\ tvar(n') = \mathsf{inr}\ x\ \mathsf{in}\ e')\ (\kappa(n'+1)(tvar(n'))))
\end{array}
$$

$$
\begin{array}{l}
ecps(\mathsf{ref}\ e)(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad ret\ (\boldsymbol{\lambda}e'.\ tvar(n') \leftarrow \mathsf{ref}\ x;\ e')\ (\kappa(n'+1)(tvar(n'))))
\end{array}
$$

$$
\begin{array}{l}
ecps(!e)(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad ret\ (\boldsymbol{\lambda}e'.\ tvar(n') \leftarrow !x;\ e')\ (\kappa(n'+1)(tvar(n'))))
\end{array}
$$

$$
\begin{array}{l}
ecps(\mathsf{output}\ e)(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad ret\ (\boldsymbol{\lambda}e'.\ \mathsf{let}\ tvar(n') = \langle\rangle\ \mathsf{in}\ \mathsf{output}\ x;\ e')\ (\kappa(n'+1)(tvar(n'))))
\end{array}
$$

$$
\begin{array}{l}
ecps(\langle e_1, e_2\rangle)(n)(\phi)(\kappa) = \\
\quad ecps(e_1)(n)(\phi)(\boldsymbol{\lambda}(n', x_1). \\
\quad\quad ecps(e_2)(n')(\phi)(\boldsymbol{\lambda}(n'', x_2). \\
\quad\quad\quad ret\ (\boldsymbol{\lambda}e.\ \mathsf{let}\ tvar(n'') = \langle x_1, x_2\rangle\ \mathsf{in}\ e)\ (\kappa(n''+1)(tvar(n'')))))
\end{array}
$$

$$
\begin{array}{l}
ecps(e_1 \odot e_2)(n)(\phi)(\kappa) = \\
\quad ecps(e_1)(n)(\phi)(\boldsymbol{\lambda}(n', x_1). \\
\quad\quad ecps(e_2)(n')(\phi)(\boldsymbol{\lambda}(n'', x_2). \\
\quad\quad\quad ret\ (\boldsymbol{\lambda}e.\ \mathsf{let}\ tvar(n'') = x_1 \odot x_2\ \mathsf{in}\ e)\ (\kappa(n''+1)(tvar(y'')))))
\end{array}
$$

$$
\begin{array}{l}
ecps(e_1 := e_2)(n)(\phi)(\kappa) = \\
\quad ecps(e_1)(n)(\phi)(\boldsymbol{\lambda}(n', x_1). \\
\quad\quad ecps(e_2)(n')(\phi)(\boldsymbol{\lambda}(n'', x_2). \\
\quad\quad\quad ret\ (\boldsymbol{\lambda}e.\ \mathsf{let}\ tvar(n'') = \langle\rangle\ \mathsf{in}\ x_1 := x_2;\ e)\ (\kappa(n''+1)(tvar(n'')))))
\end{array}
$$

Figure 5.3: CPS transformation (part 2 of 3).

$$
\begin{array}{l}
ecps(e_1 == e_2)(n)(\phi)(\kappa) = \\
\quad ecps(e_1)(n)(\phi)(\boldsymbol{\lambda}(n', x_1). \\
\quad\quad ecps(e_2)(n')(\phi)(\boldsymbol{\lambda}(n'', x_2). \\
\quad\quad\quad ret\ (\boldsymbol{\lambda} e.\ \mathsf{let}\ tvar(n'') = x_1 == x_2\ \mathsf{in}\ e)\ (\kappa(n''+1)(tvar(n''))))) \\
\\
ecps(\mathsf{ifnz}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad let\ (n'', e_1') := ecps(e_1)(n')(\phi)(\kappa)\ in \\
\quad\quad ret\ (\boldsymbol{\lambda} e_2'.\ \mathsf{ifnz}\ x\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\ (ecps(e_2)(n'')(\phi)(\kappa))) \\
\\
ecps(e_1\ e_2)(n)(\phi)(\kappa) = \\
\quad ecps(e_1)(n)(\phi)(\boldsymbol{\lambda}(n', x_1). \\
\quad\quad ecps(e_2)(n')(\phi)(\boldsymbol{\lambda}(n'', x_2). \\
\quad\quad\quad ret\ (\boldsymbol{\lambda} e'.\ \mathsf{let}\ kvar(n'') = \mathsf{cont}\ tvar(n''+1).\, e'\ \mathsf{in}\ x_1\ x_2\ kvar(n'')) \\
\quad\quad\quad\quad (\kappa(n''+2)(tvar(n''+1))))) \\
\\
ecps(e\ [])(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad ret\ (\boldsymbol{\lambda} e'.\ \mathsf{let}\ kvar(n') = \mathsf{cont}\ tvar(n'+1).\, e'\ \mathsf{in}\ x\ []\ kvar(n')) \\
\quad\quad\quad (\kappa(n'+2)(tvar(n'+1)))) \\
\\
ecps(\mathsf{case}\ e\ (y.\ e_1)\ (y.\ e_2))(n)(\phi)(\kappa) = \\
\quad ecps(e)(n)(\phi)(\boldsymbol{\lambda}(n', x). \\
\quad\quad let\ \phi' := \phi[tvar(n')/y]\ in \\
\quad\quad let\ (n'', e_1') := ecps(e_1)(n'+1)(\phi')(\kappa)\ in \\
\quad\quad ret\ (\boldsymbol{\lambda} e_2'.\ \mathsf{case}\ x\ (tvar(n').\ e_1')\ (tvar(n').\ e_2'))\ (ecps(e_2)(n'')(\phi')(\kappa))) \\
\\
ecps(\mathsf{fix}\, f(y).\, e)(n)(\phi)(\kappa) = \\
\quad let\ f' := tvar(n+1)\ in \\
\quad let\ y' := tvar(n+2)\ in \\
\quad let\ k := kvar(n+3)\ in \\
\quad let\ \phi' := \phi[f'/f][y'/y]\ in \\
\quad let\ (n_1, e') := ecps(e)(n+4)(\phi')(\boldsymbol{\lambda} n_2, x.\ (n_2, k\ x))\ in \\
\quad ret\ (\boldsymbol{\lambda} e''.\ \mathsf{let}\ tvar(n) = \mathsf{fix}\, f'(y', k).\, e'\ \mathsf{in}\ e'')\ (\kappa(n_1)(tvar(n))) \\
\\
ecps(\Lambda.\, e)(n)(\phi)(\kappa) = \\
\quad let\ k := kvar(n+1)\ in \\
\quad let\ (n_1, e') := ecps(e)(n+2)(\phi')(\boldsymbol{\lambda} n_2, x.\ (n_2, k\ x))\ in \\
\quad ret\ (\boldsymbol{\lambda} e''.\ \mathsf{let}\ tvar(n) = \Lambda k.\, e'\ \mathsf{in}\ e'')\ (\kappa(n_1)(tvar(n)))
\end{array}
$$

Figure 5.4: CPS transformation (part 3 of 3).

$tvar(n)$. It passes this variable on as input to the continuation, together with the updated variable counter $n+1$ in order to ensure that $tvar(n)$ will not be chosen again.

- The translation of $\mathsf{pack}\ e$ simply translates e and forgets about the packing since there is no such concept in $\mathcal{I}$.

- The translation of $\mathsf{inl}\ e$ translates e with a special continuation. This continuation introduces a let-binding for inl_x , where x is the variable (either term variable or label) holding the value of e's translation. The body of the let-binding is determined by the meta-level continuation κ.

- The translation of $\mathsf{fix}\, f(y).\, e$ creates a new let-binding for the function definition. The body of the let-binding is determined by κ. The body of the function is the translation of e relative to the meta-level continuation $\boldsymbol{\lambda} n_2, x.\ (n_2, k\ x)$, which is a reified version of the function's formal continuation argument k. All bound variables are chosen to be fresh, and the renaming ϕ' used in translation of the function body knows which fresh variables correspond to occurrences of f and y in e. Note that *fcps* necessarily translates top-level functions slightly differently.

- The translation of $e_1\ e_2$ translates e_1 with a continuation that translates e_2, in turn with a continuation that issues the actual function call. The original continuation κ is reified into an object-level continuation by introducing a continuation let-binding. The continuation variable $kvar(n'')$ is then passed to the function x_1 in the call.

5.2.1.1 Example

Consider a source module $M_{\mathcal{S}}$ consisting of a single function labelled G:

$$G = \mathsf{fix}\, f(x).\, \langle F\ 42, !x \rangle$$

It returns a pair whose first component is the result of calling the imported function F on 42. Its second component is the value stored in the argument reference x.

The $\mathcal{I}$ module $cps(M_{\mathcal{S}})$, *i.e.,* the result of transforming $M_{\mathcal{S}}$, consists of the following single function (up to the exact choice of fresh variables):

$$\begin{array}{l} G = \mathsf{fix}\, f'(y, k). \\ \qquad \mathsf{let}\ y_1 = 42\ \mathsf{in} \\ \qquad \mathsf{let}\ k_1 = (\mathsf{cont}\ y_2.\ \ y_3 \leftarrow !y; \\ \qquad\qquad\qquad\qquad\quad \mathsf{let}\ y_4 = \langle y_2, y_3 \rangle\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\quad k_1\ y_4)\ \mathsf{in} \\ \qquad F\ y_1\ k_1 \end{array}$$

5.2.2 Verification

It is easy to show that the CPS transformation of a well-typed $\mathcal{S}$ module is a well-formed $\mathcal{I}$ module with the same imports and exports. Recall that $\mathcal{I}$ well-formedness includes the affinity of continuation variables (Section 4.2.4.2). It does not include the uniqueness of bound variables, but that does hold as well. These facts are important for the correctness of the subsequent transformations performed by Pilsner.

Theorem 17.

$$\frac{\Gamma \vdash M : \Gamma'}{|\Gamma| \vdash \mathit{cps}(M) : |\Gamma'| \quad \mathit{uniqmod}(\mathit{cps}(M))}$$

The main correctness statement is the following.

Theorem 18.

$$\frac{\Gamma \vdash M : \Gamma'}{\Gamma \vdash \mathit{cps}(M) \precsim_{\mathcal{IS}} M : \Gamma'}$$

For the proof of this, we pick the *empty local world* $w^\emptyset \in \mathrm{World}^{\Omega_{\mathcal{IS}}.\mathsf{T}}$ because the effects of the CPS transformation on memory are trivial. We define the empty world generically, such that it can be reused. Note the use of gwf for flexibility.

Definition 36 (Empty local world)**.**

$$\begin{array}{lcl}
w^\emptyset \in \mathrm{LWorld}^T_{A,B} & & \\
w^\emptyset.\mathsf{T}.\mathsf{S} & := & 1 \\
w^\emptyset.\mathsf{T}.\sqsupseteq & := & 1 \times 1 \\
w^\emptyset.\mathsf{T}.\sqsupseteq_{\mathrm{pub}} & := & 1 \times 1 \\
w^\emptyset.\mathsf{C}(G)(s) & := & \{\, (\emptyset, \emptyset) \,\} \\
w^\emptyset.\mathsf{N}.\mathsf{NS} & := & \emptyset \\
w^\emptyset.\mathsf{N}.\mathsf{NR}(G)(s)(\mathbf{n}) & := & \emptyset \\
w^\emptyset.\mathsf{O} & := & \mathrm{gwf}
\end{array}$$

The proof of Theorem 18 boils down to finding a relation $\precsim^{\mathrm{cps}}_{\mathcal{IS}}$ on expressions and their translations that satisfies

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathit{ecps}(e) \precsim^{\mathrm{cps}}_{\mathcal{IS}} e : \tau}$$

and at the same time is strong enough to let us derive the module-level correctness. We now present and justify our choice of $\precsim^{\mathrm{cps}}_{\mathcal{IS}}$.

Definition 37.

$$
\begin{aligned}
&\Gamma \vdash f \precsim_{\mathcal{IS}} e : \tau := \\
&\quad \exists i.\ \forall \delta, n, \phi, G, s_0, s, \sigma, \gamma, j, \mathbf{k_a}, \mathbf{k_b}, \tau', \kappa, K, e'. \\
&\qquad (\_, e') = f(n)(\phi)(\kappa) \wedge \\
&\qquad G \in \mathbf{GK}_{w^{\emptyset\uparrow}} \wedge s_0 \sqsupseteq_{\text{pub}} s \wedge \\
&\qquad (\forall F{:}\tau'' \in \Gamma.\ \exists \mathbf{v_a}, \mathbf{v_b}.\ \sigma(F) = \mathbf{v_a} \wedge s.\rho(F) = \mathbf{v_b} \wedge (\mathbf{v_a}, \mathbf{v_b}) \in \overline{G}(s)(\delta\tau'')) \wedge \\
&\qquad (\forall x{:}\tau'' \in \Gamma.\ \exists \mathbf{v_a}, \mathbf{v_b}.\ \sigma(\phi(x)) = \mathbf{v_a} \wedge \gamma(x) = \mathbf{v_b} \wedge (\mathbf{v_a}, \mathbf{v_b}) \in \overline{G}(s)(\delta\tau'')) \wedge \\
&\qquad \mathit{uniqvars}(e') \wedge \mathrm{bv}(e') \cap \mathrm{dom}(\sigma) = \emptyset \wedge \\
&\qquad ((\sigma, \kappa), K) \in \mathfrak{K}(j)(\mathbf{k_a}, \mathbf{k_b})(G)(s_0)(s)(\delta\tau)(\tau') \\
&\quad \implies ((\emptyset, (\sigma, e'), (\emptyset, \emptyset, K[\gamma(e)])) \in \mathbf{E}_{w^{\emptyset\uparrow}}(i + j)(\mathbf{k_a}, \mathbf{k_b})(G)(s_0)(s)(\bot)(\tau')
\end{aligned}
$$

Here, f is a meta-level function of type $\mathbb{N} \to (\mathsf{Var} \to \mathrm{TVar} \cup \mathsf{Lbl}) \to (\mathbb{N} \to \mathrm{TVar} \cup \mathsf{Lbl} \to \mathbb{N} \times \mathsf{Exp}_{\mathcal{I}}) \to \mathbb{N} \times \mathsf{Exp}_{\mathcal{I}}$ and e an $\mathcal{S}$ expression. Roughly, we want to say that f (think: $\mathit{ecps}(e)$) is related to e by the well-known $\mathbf{E}$ relation. Of course, for this to make sense, we must first turn them into suitable configurations.

First we need to quantify over the arguments to f, namely n, ϕ, and κ. Following the shown definition, let us call the resulting expression e'. Now we have two open expressions, e' and e. To close these with respect to labels—as listed in Γ—we quantify over an environment σ for e' and a state s containing a (label-only) environment $s.\rho$ for e. These should of course not be arbitrary, but related by (the value closure of) the global knowledge, over which we quantify as well, at the type indicated by Γ. Since that type might contain free variables as well, we first close it with a substitution δ mapping type variables to arbitrary closed types (which may contain type names).

Besides labels, Γ and thus e can also contain variables (recall that $\Gamma \vdash e : \tau$). To close e with respect to those, we quantify over a value substitution γ. These $\mathcal{S}$ variables correspond to $\mathcal{I}$ term variables and labels, as given by the renaming ϕ used in the CPS transformation. We require that the values in γ are related to the values in σ accordingly, *i.e.,* , if $x{:}\tau'' \in \Gamma$ then the value in σ for the renaming of x is related to $\gamma(x)$. In addition to closing e using γ, we also put it in an evaluation context K, since the expression may occurr anywhere in the source program.

The key remaining condition enforces a connection between this evaluation context K and the meta-level continuation κ used to generate e'. But what is $\mathfrak{K}$? Certainly we don't want to describe their relationship syntactically. Can we not just say that these continuations (or some conversions thereof) are related by PILS's $\mathbf{K}$ relation? The issue is that the way in which e' and e "return" to these continuations is very different from how functions return to their continuations and thus is not compatible with $\mathbf{K}$. This is not specific to the CPS transformation but rather is a general observation: the "continuations" inherent in the translation of inductively-defined expressions may be very different (in shape and/or behavior) from the continuations associated with function calls that are exposed in our language specification and configuration queries. This will become even more apparent when we get to code generation in Section 5.10.

So let us look at $\mathfrak{K}$, defined as follows.

Definition 38.

$$\begin{aligned}\mathfrak{K}(i)(\mathbf{k_a},\mathbf{k_b})(G)(s_0)(s)(\tau)(\tau') := \{\, ((\sigma,\kappa),K) \mid \\ \forall G', s', \mathbf{v_a}, \mathbf{v_b}, x, \sigma', n, e''. \\ (\_, e'') = \kappa(n)(x) \wedge \\ G' \supseteq G \wedge s' \sqsupseteq_{\text{pub}} s_0 \wedge s' \sqsupseteq s \wedge \\ (\mathbf{v_a},\mathbf{v_b}) \in \overline{G'}(s')(\tau) \wedge \\ \sigma \subseteq \sigma' \wedge \sigma'(x) = \mathbf{v_a} \wedge \\ \mathit{uniqvars}(e'') \wedge \mathrm{dom}(\sigma') \cap \mathrm{bv}(e') = \emptyset \\ \implies ((\emptyset,(\sigma',e'')),(\emptyset,\emptyset,K[\mathbf{v_b}])) \in \mathbf{E}_{w^{\emptyset}\uparrow}(i)(\mathbf{k_a},\mathbf{k_b})(G')(s_0)(s')(\bot)(\tau)\,\}\end{aligned}$$

First, it gives arguments n and x to κ in order to generate the actual expression e''. Then it roughly says that e'' in an environment σ' that maps the term variable or label x to a value $\mathbf{v_a}$ is related to $K[\mathbf{v_b}]$ by the usual $\mathbf{E}$ relation if $\mathbf{v_a}$ and $\mathbf{v_b}$ are related (by the value closure of the global knowledge, as usual). Without any further restrictions on σ', however, we would have little hope of showing continuations related by $\mathfrak{K}$. We must insist that σ' be an extension of the earlier σ in which we started executing e'. Note that x may or may not have already existed in σ. Also note that we quantify over a larger global knowledge G' because e' and e (from $\precsim_{\mathcal{IS}}$) may make external calls before the continuations are reached.

A word on the use of budgets: $\precsim^{\text{cps}}_{\mathcal{IS}}$ requires that there exists a budget i (intuitively: for f) such that the constructed configurations are related by $\mathbf{E}$ at budget $i+j$, where j is the continuation budget. This is very natural, as the configuration consists of both the continuation and the expression. The assumption that all continuations are related at the same budget j does not impose a restriction because one can always choose j to be large enough for all continuations in question[1].

As intended, we can prove the following key lemma about $\precsim^{\text{cps}}_{\mathcal{IS}}$.

Lemma 68.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathit{ecps}(e) \precsim^{\text{cps}}_{\mathcal{IS}} e : \tau}$$

Proof. By induction on e, with a nested coinduction on $\mathbf{E}$ when e is a recursive function $\mathsf{fix}\, f(y,k).\, e'$. □

This lemma is then used in the proof of Theorem 18, where it is applied to the body of top-level functions.

5.3 Infrastructure for Optimizations

After CPS conversion and thus translation into $\mathcal{I}$, Pilsner performs several transformations at the intermediate level, *i.e.,* transformations from $\mathcal{I}$ to $\mathcal{I}$. In preparation of verifying these, we develop some common infrastructure.

[1] The $\mathbf{E}$ relation, and thus $\mathfrak{K}$, is closed under budget increase.

5.3.1 Relating Open Expressions

Somewhat similar to $\precsim^{\text{cps}}_{\mathcal{IS}}$, we define a relation between open $\mathcal{I}$ expressions. This is a fairly straightforward lifting of $\mathbf{E}$:

Definition 39.

$$
\begin{aligned}
&\Gamma \vdash e_{\mathsf{a}} \precsim^{\phi}_{w} e_{\mathsf{b}} := \\
&\quad \exists i.\ \forall G, s_0, s, \sigma_{\mathsf{a}}, \sigma_{\mathsf{b}}, j, \mathbf{k}_{\mathsf{a}}, \mathbf{k}_{\mathsf{b}}. \\
&\qquad G \in \mathbf{GK}_{w\uparrow} \wedge s \sqsupseteq_{\text{pub}} s_0 \wedge \\
&\qquad (\forall x \in \Gamma.\ \exists \mathbf{v}_{\mathsf{a}}, \mathbf{v}_{\mathsf{b}}.\ \sigma_{\mathsf{a}}(\phi(x)) = \mathbf{v}_{\mathsf{a}} \wedge \sigma_{\mathsf{b}}(x) = \mathbf{v}_{\mathsf{b}} \wedge (\mathbf{v}_{\mathsf{a}}, \mathbf{v}_{\mathsf{b}}) \in \overline{G}(s)) \wedge \\
&\qquad (\forall k \in \Gamma.\ \exists \mathbf{k}'_{\mathsf{a}}, \mathbf{k}'_{\mathsf{b}}.\ \sigma_{\mathsf{a}}(\phi(k)) = \mathbf{k}'_{\mathsf{a}} \wedge \sigma_{\mathsf{b}}(k) = \mathbf{k}'_{\mathsf{b}} \wedge \\
&\qquad\quad (\mathbf{k}'_{\mathsf{a}}, \mathbf{k}'_{\mathsf{b}}) \in \mathbf{K}_{w\uparrow}(j)(\mathbf{k}_{\mathsf{a}}, \mathbf{k}_{\mathsf{b}})(G)(s_0)(s)) \\
&\quad \implies ((\emptyset, (\sigma_{\mathsf{a}}, e_{\mathsf{a}})), (\emptyset, (\sigma_{\mathsf{b}}, e_{\mathsf{b}}))) \in \mathbf{E}_{w\uparrow}(i + j)(\mathbf{k}_{\mathsf{a}}, \mathbf{k}_{\mathsf{b}})(G)(s_0)(s)(\bot)
\end{aligned}
$$

While $\precsim^{\text{cps}}_{\mathcal{IS}}$ was defined in terms of the empty local world, here we parameterize the relation by an arbitrary local world $w \in \text{LWorld}^{\Omega_{\mathcal{II}}.\mathsf{T}}$ for flexibility (we will need to use a non-trivial world in some proofs). The relation roughly says that the $\mathcal{I}$ expressions e_{a} and e_{b} must be related by $\mathbf{E}$ when paired with related environments σ_{a} and σ_{b}. Related environments must map term variables and labels x to values related by (the value closure) of the global knowledge. They must also map continuation variables k to continuations related by $\mathbf{K}$. In contrast to $\precsim^{\text{cps}}_{\mathcal{IS}}$, here we do not need to define a custom continuation relation because the notion of internal continuation coincides with the "official" notion of continuation, and so does their calling convention.

Note how we allow any variable in the source e_{b} to have been renamed in the target e_{a} by applying a renaming ϕ before the lookup in σ_{a}. This renaming is an index of the relation.

Besides control expressions e, there are pure expressions a in $\mathcal{I}$. For these we define a similar relation (in fact, we overload the notation). This relation roughly says that, given related environments, if the target expression a_{b} succesfully evaluates to a value, then so does the source expression a_{a}, and the resulting values are related. Regarding the environments, the relation only cares about term variables and labels in Γ, not about continuation variables—recall that a well-formed pure expression does not refer to any continuation variables (Section 4.2.4.2).

Definition 40.

$$
\begin{aligned}
&\Gamma \vdash a_{\mathsf{a}} \precsim^{\phi}_{w} a_{\mathsf{b}} := \\
&\quad \forall G, s, \sigma_{\mathsf{a}}, \sigma_{\mathsf{b}}, \mathbf{v}_{\mathsf{b}}. \\
&\qquad G \in \mathbf{GK}_{w\uparrow} \wedge \\
&\qquad (\forall x \in \Gamma.\ \exists \mathbf{v}'_{\mathsf{a}}, \mathbf{v}'_{\mathsf{b}}.\ \sigma_{\mathsf{a}}(\phi(x)) = \mathbf{v}'_{\mathsf{a}} \wedge \sigma_{\mathsf{b}}(x) = \mathbf{v}'_{\mathsf{b}} \wedge (\mathbf{v}'_{\mathsf{a}}, \mathbf{v}'_{\mathsf{b}}) \in \overline{G}(s)) \wedge \\
&\qquad [\![a_{\mathsf{b}}]\!]_{\sigma_{\mathsf{b}}} = \mathbf{v}_{\mathsf{b}} \\
&\quad \implies \exists \mathbf{v}_{\mathsf{a}}.\ [\![a_{\mathsf{a}}]\!]_{\sigma_{\mathsf{a}}} = \mathbf{v}_{\mathsf{a}} \wedge (\mathbf{v}_{\mathsf{a}}, \mathbf{v}_{\mathsf{b}}) \in \overline{G}(s)
\end{aligned}
$$

For these relations, we prove typical compatibility lemmas, which state that each relation is preserved by the respective forms of $\mathcal{I}$ expressions. These lemmas are very helpful in verifying transformations where subexpressions stay unchanged (as we will see in the coming sections). They are shown in Figures 5.5 and 5.6. For each well-formedness rule from Figure 4.5, we have a corresponding lemma here.

In contrast to the compatibility lemmas that we have seen for PBs in Section 3.7.3, here we allow variables of the source expression to be renamed in the target expression. As a consequence of this, lemmas for constructs that introduce bound variables require a side condition of the form $y_\mathsf{a} \lhd^\phi_Z y_\mathsf{b}$, where y_a is the variable bound by the left-side term and y_b is the variable bound by the right-side term. This relation is defined as follows.

Definition 41.

$$\begin{aligned} y_\mathsf{a} \lhd^\phi_Z y_\mathsf{b} \quad &:= \quad \forall y'_\mathsf{b} \in Z.\ \phi(y'_\mathsf{b}) = y_\mathsf{a} \implies y'_\mathsf{b} = y_\mathsf{b} \\ k_\mathsf{a} \lhd^\phi_Z k_\mathsf{b} \quad &:= \quad \forall k'_\mathsf{b} \in Z.\ \phi(k'_\mathsf{b}) = k_\mathsf{a} \implies k'_\mathsf{b} = k_\mathsf{b} \end{aligned}$$

This side condition enforces that a variable y_b can be renamed to y_a only if no other variable has been renamed to y_a. A special case of these lemmas is obtained when not renaming anything, *i.e.,* when $\phi = id$. In that case, all these side conditions hold trivially.

The lemmas are straightforward but tedious to show. Only the one about recursive functions requires a proof by coinduction, as one would expect.

Note that the lemma about allocation $(y \leftarrow \mathsf{ref}\ x;\ e)$ requires stability of w: in the proof of the lemma, when we extend the global state by adding a new pair of references to the partial bijection, we need to know that the local world's configuration relation can still be satisfied. None of the other lemmas requires stability.

Combining these lemmas with a straightforward induction yields the following lemma.

Lemma 69.

$$\frac{\Gamma \vdash a \quad w \in \mathsf{stable}(\Omega_{\mathcal{II}}) \quad \forall z \in \Gamma.\ \phi(z) \in \mathrm{bv}(e) \implies \phi(z) = z}{\Gamma \vdash \phi(a) \precsim^\phi_w a}$$

$$\frac{\Gamma \vdash e \quad w \in \mathsf{stable}(\Omega_{\mathcal{II}}) \quad \forall z \in \Gamma.\ \phi(z) \in \mathrm{bv}(e) \implies \phi(z) = z}{\Gamma \vdash \phi(e) \precsim^\phi_w e}$$

In the conclusions, $\phi(a)$ and $\phi(e)$ denote the application of the renaming ϕ to each *free* variable in a and e, respectively. The side condition on ϕ rules out that any of these is renamed to a variable captured by a binder in the expression. Naturally, if the renaming is the identity function, this holds again trivially and thus the lemma becomes a (conditional) reflexivity property:

$$\frac{}{Z \vdash \langle\rangle \precsim^{\phi}_{w} \langle\rangle} \qquad \frac{}{Z \vdash n \precsim^{\phi}_{w} n} \qquad \frac{x_{\mathsf{b}} \in Z \quad x_{\mathsf{a}} = \phi(x_{\mathsf{b}})}{Z \vdash x_{\mathsf{a}}.1 \precsim^{\phi}_{w} x_{\mathsf{b}}.1} \qquad \frac{x_{\mathsf{b}} \in Z \quad x_{\mathsf{a}} = \phi(x_{\mathsf{b}})}{Z \vdash x_{\mathsf{a}}.2 \precsim^{\phi}_{w} x_{\mathsf{b}}.2}$$

$$\frac{x_{\mathsf{b}} \in Z \quad x_{\mathsf{a}} = \phi(x_{\mathsf{b}})}{Z \vdash \mathsf{inl}\ x_{\mathsf{a}} \precsim^{\phi}_{w} \mathsf{inl}\ x_{\mathsf{b}}} \qquad \frac{x_{\mathsf{b}} \in Z \quad x_{\mathsf{a}} = \phi(x_{\mathsf{b}})}{Z \vdash \mathsf{inr}\ x_{\mathsf{a}} \precsim^{\phi}_{w} \mathsf{inr}\ x_{\mathsf{b}}}$$

$$\frac{x^1_{\mathsf{b}}, x^2_{\mathsf{b}} \in Z \quad x^1_{\mathsf{a}} = \phi(x^1_{\mathsf{b}}) \quad x^2_{\mathsf{a}} = \phi(x^2_{\mathsf{b}})}{Z \vdash \langle x^1_{\mathsf{a}}, x^2_{\mathsf{a}}\rangle \precsim^{\phi}_{w} \langle x^1_{\mathsf{b}}, x^2_{\mathsf{b}}\rangle} \qquad \frac{x^1_{\mathsf{b}}, x^2_{\mathsf{b}} \in Z \quad x^1_{\mathsf{a}} = \phi(x^1_{\mathsf{b}}) \quad x^2_{\mathsf{a}} = \phi(x^2_{\mathsf{b}})}{Z \vdash x^1_{\mathsf{a}} \odot x^2_{\mathsf{a}} \precsim^{\phi}_{w} x^1_{\mathsf{b}} \odot x^2_{\mathsf{b}}}$$

$$\frac{x^1_{\mathsf{b}}, x^2_{\mathsf{b}} \in Z \quad x^1_{\mathsf{a}} = \phi(x^1_{\mathsf{b}}) \quad x^2_{\mathsf{a}} = \phi(x^2_{\mathsf{b}})}{Z \vdash x^1_{\mathsf{a}} == x^2_{\mathsf{a}} \precsim^{\phi}_{w} x^1_{\mathsf{b}} == x^2_{\mathsf{b}}}$$

$$\frac{\begin{array}{c} Z^{\dagger}, f_{\mathsf{b}}, y_{\mathsf{b}}, k_{\mathsf{b}} \vdash e_{\mathsf{a}} \precsim^{\phi'}_{w} e_{\mathsf{b}} \\ \phi' = \phi[f_{\mathsf{a}}/f_{\mathsf{b}}][y_{\mathsf{a}}/y_{\mathsf{b}}][k_{\mathsf{a}}/k_{\mathsf{b}}] \quad f_{\mathsf{a}} = y_{\mathsf{a}} \implies f_{\mathsf{b}} = y_{\mathsf{b}} \quad f_{\mathsf{a}} \lhd^{\phi}_{Z} f_{\mathsf{b}} \quad y_{\mathsf{a}} \lhd^{\phi}_{Z} y_{\mathsf{b}} \end{array}}{Z \vdash \mathsf{fix}\, f_{\mathsf{a}}(y_{\mathsf{a}}, k_{\mathsf{a}}).\, e_{\mathsf{a}} \precsim^{\phi}_{w} \mathsf{fix}\, f_{\mathsf{b}}(y_{\mathsf{b}}, k_{\mathsf{b}}).\, e_{\mathsf{b}}}$$

$$\frac{Z^{\dagger}, k_{\mathsf{b}} \vdash e_{\mathsf{a}} \precsim^{\phi'}_{w} e_{\mathsf{b}} \quad \phi' = \phi[k_{\mathsf{a}}/k_{\mathsf{b}}]}{Z \vdash \Lambda k_{\mathsf{a}}.\, e_{\mathsf{a}} \precsim^{\phi}_{w} \Lambda k_{\mathsf{b}}.\, e_{\mathsf{b}}}$$

. . . . . .

$$\frac{\Gamma \vdash a_{\mathsf{a}} \precsim^{\phi}_{w} a_{\mathsf{b}} \quad \Gamma, y_{\mathsf{b}} \vdash e_{\mathsf{a}} \precsim^{\phi'}_{w} e_{\mathsf{b}} \quad \phi' = \phi[y_{\mathsf{a}}/y_{\mathsf{b}}] \quad y_{\mathsf{a}} \lhd^{\phi}_{Z} y_{\mathsf{b}}}{\Gamma \vdash \mathsf{let}\ y_{\mathsf{a}} = a_{\mathsf{a}}\ \mathsf{in}\ e_{\mathsf{a}} \precsim^{\phi}_{w} \mathsf{let}\ y_{\mathsf{b}} = a_{\mathsf{b}}\ \mathsf{in}\ e_{\mathsf{b}}}$$

$$\frac{\begin{array}{c} \Gamma, y_{\mathsf{b}} \vdash e_{\mathsf{a}} \precsim^{\phi'}_{w} e_{\mathsf{b}} \quad \phi' = \phi[y_{\mathsf{a}}/y_{\mathsf{b}}] \quad y_{\mathsf{a}} \lhd^{\phi}_{Z} y_{\mathsf{b}} \\ \Gamma, k_{\mathsf{b}} \vdash e'_{\mathsf{a}} \precsim^{\phi''}_{w} e'_{\mathsf{b}} \quad \phi'' = \phi[k_{\mathsf{a}}/k_{\mathsf{b}}] \quad k_{\mathsf{a}} \lhd^{\phi}_{Z} k_{\mathsf{b}} \end{array}}{\Gamma \vdash \mathsf{let}\ k_{\mathsf{a}} = \mathsf{cont}\ y_{\mathsf{a}}.\, e_{\mathsf{a}}\ \mathsf{in}\ e'_{\mathsf{a}} \precsim^{\phi}_{w} \mathsf{let}\ k_{\mathsf{b}} = \mathsf{cont}\ y_{\mathsf{b}}.\, e_{\mathsf{b}}\ \mathsf{in}\ e'_{\mathsf{b}}}$$

$$\frac{\Gamma, y_{\mathsf{b}} \vdash e_{\mathsf{a}} \precsim^{\phi'}_{w} e_{\mathsf{b}} \quad \phi' = \phi[y_{\mathsf{a}}/y_{\mathsf{b}}] \quad y_{\mathsf{a}} \lhd^{\phi}_{Z} y_{\mathsf{b}}}{\Gamma \vdash y_{\mathsf{a}} \leftarrow \mathsf{input};\ e_{\mathsf{a}} \precsim^{\phi}_{w} y_{\mathsf{b}} \leftarrow \mathsf{input};\ e_{\mathsf{b}}}$$

Figure 5.5: Compatibility lemmas for $\precsim^{\phi}_{w}$

$$\frac{x_\mathsf{b} \in Z \quad x_\mathsf{a} = \phi(x_\mathsf{b}) \quad \Gamma \vdash e_\mathsf{a} \precsim^\phi_w e_\mathsf{b}}{\Gamma \vdash \mathsf{output}\ x_\mathsf{a};\ e_\mathsf{a} \precsim^\phi_w \mathsf{output}\ x_\mathsf{b};\ e_\mathsf{b}}$$

$$\frac{x_\mathsf{b} \in Z \quad x_\mathsf{a} = \phi(x_\mathsf{b}) \quad \Gamma \vdash e^1_\mathsf{a} \precsim^\phi_w e^1_\mathsf{b} \quad \Gamma \vdash e^2_\mathsf{a} \precsim^\phi_w e^2_\mathsf{b}}{\Gamma \vdash \mathsf{ifnz}\ x_\mathsf{a}\ \mathsf{then}\ e^1_\mathsf{a}\ \mathsf{else}\ e^2_\mathsf{a} \precsim^\phi_w \mathsf{ifnz}\ x_\mathsf{b}\ \mathsf{then}\ e^1_\mathsf{b}\ \mathsf{else}\ e^2_\mathsf{b}}$$

$$\frac{x_\mathsf{b} \in Z \quad x_\mathsf{a} = \phi(x_\mathsf{b}) \quad \Gamma, y_\mathsf{b} \vdash e^1_\mathsf{a} \precsim^{\phi'}_w e^1_\mathsf{b} \quad \Gamma, y_\mathsf{b} \vdash e^2_\mathsf{a} \precsim^{\phi'}_w e^2_\mathsf{b} \quad \phi' = \phi[y_\mathsf{a}/y_\mathsf{b}] \quad y_\mathsf{a} \lhd^\phi_Z y_\mathsf{b}}{\Gamma \vdash \mathsf{case}\ x_\mathsf{a}\ (y_\mathsf{a}.\ e^1_\mathsf{a})\ (y_\mathsf{a}.\ e^2_\mathsf{a}) \precsim^\phi_w \mathsf{case}\ x_\mathsf{b}\ (y_\mathsf{b}.\ e^1_\mathsf{b})\ (y_\mathsf{b}.\ e^2_\mathsf{b})}$$

$$\frac{x^1_\mathsf{b}, x^2_\mathsf{b}, k_\mathsf{b} \in Z \quad x^1_\mathsf{a} = \phi(x^1_\mathsf{b}) \quad x^2_\mathsf{a} = \phi(x^2_\mathsf{b}) \quad k_\mathsf{a} = \phi(k_\mathsf{b})}{Z \vdash x^1_\mathsf{a}\ x^2_\mathsf{a}\ k_\mathsf{a} \precsim^\phi_w x^1_\mathsf{b}\ x^2_\mathsf{b}\ k_\mathsf{b}}$$

$$\frac{x_\mathsf{b}, k_\mathsf{b} \in Z \quad x_\mathsf{a} = \phi(x_\mathsf{b}) \quad k_\mathsf{a} = \phi(k_\mathsf{b})}{Z \vdash x_\mathsf{a}\ []\ k_\mathsf{a} \precsim^\phi_w x_\mathsf{b}\ []\ k_\mathsf{b}} \qquad \frac{k_\mathsf{b}, x_\mathsf{b} \in Z \quad k_\mathsf{a} = \phi(k_\mathsf{b}) \quad x_\mathsf{a} = \phi(x_\mathsf{b})}{Z \vdash k_\mathsf{a}\ x_\mathsf{a} \precsim^\phi_w k_\mathsf{b}\ x_\mathsf{b}}$$

$$\frac{x_\mathsf{b} \in Z \quad x_\mathsf{a} = \phi(x_\mathsf{b}) \quad \Gamma, y_\mathsf{b} \vdash e_\mathsf{a} \precsim^{\phi'}_w e_\mathsf{b} \quad \phi' = \phi[y_\mathsf{a}/y_\mathsf{b}] \quad y_\mathsf{a} \lhd^\phi_Z y_\mathsf{b} \quad w \in \mathsf{stable}(\Omega_{\mathcal{II}})}{\Gamma \vdash y_\mathsf{a} \leftarrow \mathsf{ref}\ x_\mathsf{a};\ e_\mathsf{a} \precsim^\phi_w y_\mathsf{b} \leftarrow \mathsf{ref}\ x_\mathsf{b};\ e_\mathsf{b}}$$

$$\frac{x_\mathsf{b} \in Z \quad x_\mathsf{a} = \phi(x_\mathsf{b}) \quad \Gamma, y_\mathsf{b} \vdash e_\mathsf{a} \precsim^{\phi'}_w e_\mathsf{b} \quad \phi' = \phi[y_\mathsf{a}/y_\mathsf{b}] \quad y_\mathsf{a} \lhd^\phi_Z y_\mathsf{b}}{\Gamma \vdash y_\mathsf{a} \leftarrow !x_\mathsf{a};\ e_\mathsf{a} \precsim^\phi_w y_\mathsf{b} \leftarrow !x_\mathsf{b};\ e_\mathsf{b}}$$

$$\frac{x^1_\mathsf{b}, x^2_\mathsf{b} \in Z \quad x^1_\mathsf{a} = \phi(x^1_\mathsf{b}) \quad x^2_\mathsf{a} = \phi(x^2_\mathsf{b}) \quad \Gamma \vdash e_\mathsf{a} \precsim^\phi_w e_\mathsf{b}}{\Gamma \vdash x^1_\mathsf{a} := x^2_\mathsf{a};\ e_\mathsf{a} \precsim^\phi_w x^1_\mathsf{b} := x^2_\mathsf{b};\ e_\mathsf{b}}$$

Figure 5.6: Compatibility lemmas for $\precsim^\phi_w$ (cntd.)

Lemma 70.

$$\frac{\Gamma \vdash a \quad w \in \mathsf{stable}(\Omega_{\mathcal{II}})}{\Gamma \vdash a \precsim_w a} \qquad \frac{\Gamma \vdash e \quad w \in \mathsf{stable}(\Omega_{\mathcal{II}})}{\Gamma \vdash e \precsim_w e}$$

Here we write $\precsim_w$ short for $\precsim_w^{id}$.

With this and a little extra work, we can also lift the identity-renaming versions of the compatibility lemmas in Figures 5.5 and 5.6 from $\precsim_w$ to its reflexive-transitive closure $\precsim_w^*$. These will be used to establish the correctness of the framework presented in the next section. Let us just show and prove one of these rules.

Lemma 71.

$$\frac{\Gamma \vdash a_{\mathsf{a}} \precsim_w^* a_{\mathsf{b}} \quad \Gamma \vdash a_{\mathsf{b}} \quad \Gamma, y \vdash e_{\mathsf{a}} \precsim_w^* e_{\mathsf{b}} \quad \Gamma, y \vdash e_{\mathsf{b}} \quad w \in \mathsf{stable}(\Omega_{\mathcal{II}})}{\Gamma \vdash \mathsf{let}\ y = a_{\mathsf{a}}\ \mathsf{in}\ e_{\mathsf{a}} \precsim_w^* \mathsf{let}\ y = a_{\mathsf{b}}\ \mathsf{in}\ e_{\mathsf{b}}}$$

Proof. We do an induction on the first reflexive-transitive closure in the premise.

Case $a_{\mathsf{a}} = a_{\mathsf{b}}$**.** We do another induction, this time on the second reflexive-transitive closure in the premise.

Case $e_{\mathsf{a}} = e_{\mathsf{b}}$**.** Using the extra conditions on well-formedness and stability, we are done by Lemma 70.

Case $\Gamma^\dagger, k_{\mathsf{b}} \vdash e_{\mathsf{a}} \precsim_w e$ **and** $\Gamma^\dagger, k_{\mathsf{b}} \vdash e \precsim_w^* e_{\mathsf{b}}$**.** By the inner inductive hypothesis we have

$$\Gamma \vdash \mathsf{let}\ y = a_{\mathsf{a}}\ \mathsf{in}\ e \precsim_w^* \mathsf{let}\ y = a_{\mathsf{b}}\ \mathsf{in}\ e_{\mathsf{b}},$$

so it suffices to show

$$\Gamma \vdash \mathsf{let}\ y = a_{\mathsf{a}}\ \mathsf{in}\ e_{\mathsf{a}} \precsim_w \mathsf{let}\ y = a_{\mathsf{a}}\ \mathsf{in}\ e.$$

Since $\Gamma \vdash a_{\mathsf{a}} \precsim_w a_{\mathsf{a}}$ by Lemma 70, this follows by the compatibility lemma for let-bindings.

Case $\Gamma \vdash a_{\mathsf{a}} \precsim_w a$ **and** $\Gamma \vdash a \precsim_w^* a_{\mathsf{b}}$**.** We do a case analysis (not an induction) on $\Gamma, y \vdash e_{\mathsf{a}} \precsim_w^* e_{\mathsf{b}}$.

Case $e_{\mathsf{a}} = e_{\mathsf{b}}$**.** By the outer inductive hypothesis we have

$$\Gamma \vdash \mathsf{let}\ y = a\ \mathsf{in}\ e_{\mathsf{a}} \precsim_w^* \mathsf{let}\ y = a_{\mathsf{b}}\ \mathsf{in}\ e_{\mathsf{b}},$$

so it suffices to show

$$\Gamma \vdash \mathsf{let}\ y = a_{\mathsf{a}}\ \mathsf{in}\ e_{\mathsf{a}} \precsim_w \mathsf{let}\ y = a\ \mathsf{in}\ e_{\mathsf{a}}.$$

Since $e_{\mathsf{a}} = e_{\mathsf{b}}$, we know $\Gamma, y \vdash e_{\mathsf{a}}$ and can use Lemma 70 to get $\Gamma, y \vdash e_{\mathsf{a}} \precsim_w e_{\mathsf{a}}$. The goal follows by the compatibility lemma for let-bindings.

Case $\Gamma, y \vdash e_{\mathsf{a}} \precsim_w e$ **and** $\Gamma, y \vdash e \precsim_w^* e_{\mathsf{b}}$**.** Applying the inductive hypothesis to $\Gamma, y \vdash e \precsim_w^* e_{\mathsf{b}}$ (not to $\Gamma, y \vdash e_{\mathsf{a}} \precsim_w^* e_{\mathsf{b}}$) yields

$$\Gamma \vdash \mathsf{let}\ y = a\ \mathsf{in}\ e \precsim_w^* \mathsf{let}\ y = a_{\mathsf{b}}\ \mathsf{in}\ e_{\mathsf{b}},$$

so it suffices to show

$$\Gamma \vdash \mathsf{let}\ y = a_{\mathsf{a}}\ \mathsf{in}\ e_{\mathsf{a}} \precsim_w \mathsf{let}\ y = a\ \mathsf{in}\ e.$$

This follows immediately by the compatibility lemma for let-bindings.

□

5.3.2 Annotating Expressions With Transformations

In order to simplify the definition and verification of Pilsner's intermediate transformations (transformations from $\mathcal{I}$ to $\mathcal{I}$), we developed a simple framework of *transformations as expression annotations*. The idea is to split module-level transformations into two parts: (1) an analysis that is applied to each top-level function and annotates selected subexpressions with to-be-performed micro-transformations (but does not actually rewrite the code); and (2) the micro-transformations themselves, together with their correctness proofs. Given these, we automatically produce a verified module transformation that analyzes the input module and performs transformations according to the produced annotations in a bottom-up manner.

A micro-transformation is a partial function on expressions—it must fail if the preconditions for its correctness do not hold. It is typically non-recursive. For instance, here is the (only) micro-transformation used in the commute-pass of Pilsner (Section 5.4):

$$\begin{array}{l}\mathsf{commute} \in \mathsf{Exp} \rightharpoonup \mathsf{Exp} \\ \mathsf{commute}(e) := \mathsf{let}\ y_2 = a_2\ \mathsf{in}\ \mathsf{let}\ y_1 = a_1\ \mathsf{in}\ e_0 \\ \qquad \textit{if } e \textit{ is } (\mathsf{let}\ y_1 = a_1\ \mathsf{in}\ \mathsf{let}\ y_2 = a_2\ \mathsf{in}\ e_0) \textit{ and } x \notin \mathrm{fv}(a_2)\end{array}$$

In the case that a micro-transformation fails (for which the analysis is to blame), the subexpression that was being transformed simply stays unchanged, or, alternatively, the whole module transformation—and thus the compiler—fails. In either case, if the module transformation succeeds, the output module is guaranteed to correctly implement the input module. This means that the analysis does not need to be verified—in the worst case, the transformation doesn't optimize the code.

We now turn to the details.

5.3.2.1 Annotations and Transformers

Definition 42 (Annotated expressions and their erasure)**.** We write Exp^A for the set of $\mathcal{I}$ expressions that are annotated with elements l from A, defined recursively

in the obvious way (here we show only two forms but the others are analogous).

$$\begin{aligned} \mathscr{a} &\in \mathsf{BExp}^A ::= \ldots \mid [l]\ \mathsf{fix}\, f(y,k).\, \mathscr{e} \mid \ldots \\ \mathscr{e} &\in \mathsf{Exp}^A ::= \ldots \mid [l]\ \mathsf{let}\ y = \mathscr{a}\ \mathsf{in}\ \mathscr{e} \mid \ldots \end{aligned}$$

(A can be arbitrary. If it is a singleton set, then Exp^A is isomorphic to Exp.)

We write $|\mathscr{e}|$ for the $\mathcal{I}$ expression obtained by erasing all annotations from $\mathscr{e}$.

Definition 43. A *micro-transformation* is any function in $\mathsf{Exp} \rightharpoonup \mathsf{Exp}$. Given a set A of annotations, a *transformer for A* is a function $\beta \in A \to \mathsf{Exp} \rightharpoonup \mathsf{Exp}$, *i.e.,* it maps each annotation l to a micro-transformation $\beta(l)$. An *analysis for A* is a function $\alpha \in \mathsf{Exp} \to \mathsf{Exp}^A$ satisfying $|\alpha(e)| = e$ for any e, *i.e.,* it annotates expressions with elements from A but does not otherwise modify the program.

Given a transformer β and an analysis α for the same set of annotations A, we can generate a module-level transformation tf^β_α as follows. For each top-level function definition a, we run the analysis and thereby annotate the function. Then we recursively transform it according to these annotations with the help of atf^β and etf^β.

Definition 44.

$$\begin{aligned} &tf^\beta_\alpha \in \mathbf{Mod} \to \mathbf{Mod} \\ &tf^\beta_\alpha := \mathit{map}\ (\boldsymbol{\lambda}(F,a).\ (F, atf^\beta_\alpha(\alpha(a)))) \end{aligned}$$

$$\begin{aligned} &atf^\beta \in \mathsf{BExp}^A \to \mathsf{BExp} \\ &atf^\beta([l]\ \mathsf{fix}\, f(y,k).\, \mathscr{e}) := \\ &\quad \mathit{let}\ e' := \mathsf{fix}\, f(y,k).\, etf^\beta(\mathscr{e})\ \mathit{in} \\ &\quad \begin{cases} e' & \text{if } \beta(l)(e') = \bot \\ e'' & \text{if } \beta(l)(e') = e'' \end{cases} \\ &\ldots \end{aligned}$$

$$\begin{aligned} &etf^\beta \in \mathsf{Exp}^A \to \mathsf{Exp} \\ &etf^\beta([l]\ \mathsf{let}\ y = \mathscr{a}\ \mathsf{in}\ \mathscr{e}) := \\ &\quad \mathit{let}\ e' := (\mathsf{let}\ y = atf^\beta(\mathscr{a})\ \mathsf{in}\ etf^\beta(\mathscr{e}))\ \mathit{in} \\ &\quad \begin{cases} e' & \text{if } \beta(l)(e') = \bot \\ e'' & \text{if } \beta(l)(e') = e'' \end{cases} \\ &\ldots \end{aligned}$$

(We are showing only one case each for atf and etf. The others are analogous.)

Let us illustrate how this works by looking at the case of (annotated) let-bindings:

$$etf^\beta([l]\ \mathsf{let}\ y = \mathscr{a}\ \mathsf{in}\ \mathscr{e})$$

First, it recursively transforms the bound term a and the body e and puts them back together in the form of an ordinary (non-annotated) let-binding e'. Now it looks up the micro-transformation for the given annotation l and transforms e' accordingly. If this fails, it simply returns e', in effect not changing the program at all. Otherwise it returns the resulting expression (which may no longer be a let-binding).

Before moving on to the correctness of tf, let us define two convenient lifting operation on transformers.

Definition 45 (Optional Annotations)**.** Given a transformer β for some A, we define a transformer $\beta_\perp$ for $A_\perp$ that behaves in the obvious way: where the annotation is $\perp$, it does nothing; elsewhere it does the same as β.

$$\begin{aligned}
&\beta_\perp \in A_\perp \to \mathsf{Exp} \rightharpoonup \mathsf{Exp} \\
&\beta_\perp(\perp)(e) \;:=\; e \\
&\beta_\perp(l)(e) \;:=\; \begin{cases} e & \text{if } \beta(l)(e) = \perp \\ e' & \text{if } \beta(l)(e) = e' \end{cases}
\end{aligned}$$

Definition 46 (List Annotations)**.** Given a transformer β for some A, we define a transformer $\bar{\beta}$ for $List(A)$. It performs transformations according to the annotations in the given list from right to left until one fails, in which case the input to that failing transformation is returned. This means that $\bar{\beta}$ itself never fails.

$$\begin{aligned}
&\bar{\beta} \in List(A) \to \mathsf{Exp} \rightharpoonup \mathsf{Exp} \\
&\bar{\beta}(\epsilon)(e) \;:=\; e \\
&\bar{\beta}(L, l)(e) \;:=\; \begin{cases} e & \text{if } \beta(l)(e) = \perp \\ \bar{\beta}(L)(e') & \text{if } \beta(l)(e) = e' \end{cases}
\end{aligned}$$

Note that one can imagine similar list transformers with slightly different semantics, *e.g.,* one that fails itself if one of the sub-transformations fails.

5.3.2.2 Correctness

Theorem 19 states the correctness of tf: if a well-formed module M with unique variables is transformed according to a correct transformer β, then the resulting module is equally well-formed, also has unique variables, and, crucially, refines M according to the reflexive-transitive closure of $\precsim_{\mathcal{II}}$.

Theorem 19.

$$\frac{\Gamma \vdash M : \Gamma' \quad \mathit{uniqmod}(M) \quad \mathit{correct}(\beta) \quad \forall e.\ |\alpha(e)| = e}{\Gamma \vdash tf^\beta_\alpha(M) : \Gamma' \quad \mathit{uniqmod}(tf^\beta_\alpha(M)) \quad \Gamma \vdash tf^\beta_\alpha(M) \precsim^*_{\mathcal{II}} M : \Gamma'}$$

Recall that uniqueness is initially provided by the CPS transformation.

Of course, this theorem relies on the transformer β being *correct*. We define this to be the conjunction of two properties:

1. Syntactic correctness: each transformation preserves well-formedness, which includes affinity of continuation variables, and variable uniqueness (see the comment below).

$$\forall l, \Gamma, e, e'.\quad \Gamma \vdash e \wedge \mathit{uniq}(\mathrm{bv}(e), \Gamma) \wedge \beta(l)(e) = e' \implies \Gamma \vdash e' \wedge \mathrm{bv}(e') \subseteq \mathrm{bv}(e)$$

2. Semantic correctness: each transformation preserves the behavior.

$$\forall l, \Gamma, e, e', w.\quad \Gamma \vdash e \wedge \mathit{uniq}(\mathrm{bv}(e), \Gamma) \wedge \beta(l)(e) = e' \wedge w \in \mathsf{stable}(\Omega_{\mathcal{II}}) \implies \Gamma \vdash e' \precsim_w^* e$$

   The relation in the conclusion is the one from the previous section.

(It makes sense to separate these conditions because it is typically convenient to prove them separately.)

Because our variable uniqueness property is not compositional, the syntactic correctness criterion above cannot simply require the preservation of uniqueness. Instead, it requires something much stronger (and compositional), namely that the list of variables bound in e' is a sublist of that of e. This is simple and effective but also quite restrictive, and we will have to relax it later on for one of Pilsner's optimizations (Section 5.9).

The semantic part of $\mathit{correct}(\beta)$ is also fairly strict in that it requires preservation with respect to *any* (stable) local world. This is sufficient for intermediate transformations that do not touch the memory, which are the majority in Pilsner. It is not sufficient, *e.g.,* for dead code elimination (Section 5.7). We leave a suitable generalization of the correctness requirement for future work. Roughly, one would like to turn the universal quantification over w into an existential one, but one must impose some additional constraints in order to be able to find an initial state in the proof of Theorem 19.

For transformers that are correct in the above sense, we can prove the correctness of *etf*.

Lemma 72.

$$\frac{\Gamma \vdash e \quad |\boldsymbol{e}| = e \quad \mathit{uniq}(\mathrm{bv}(e), \Gamma) \quad \mathit{correct}(\beta)}{\Gamma \vdash \mathit{etf}^\beta(\boldsymbol{e}) \precsim_{w^\emptyset}^* e}$$

Proof. By induction on e. □

This lemma is strong enough to let us derive the correctness of the module-level transformation *tf* (Theorem 19).

Finally, recall the transformer liftings defined at the end of the previous section. It should come as no surprise that they preserve correctness:

Theorem 20 (Correctness of transformer liftings)**.**

- $\mathit{correct}(\beta) \implies \mathit{correct}(\beta_\perp)$

- $\mathit{correct}(\beta) \implies \mathit{correct}(\bar{\beta})$

We now move on to discuss Pilsner's individual IL transformations (for pedagogical reasons not in the order in which they are performed).

5.4 The Commute Pass

5.4.1 Transformation

Let us start with the pass that commutes let-expressions in order to enable deduplication. Recall that the goal of the commute pass is to shuffle let-bindings around such that bindings of the same expression end up being "adjacent" (if possible). For instance, in

$$\mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ \mathsf{let}\ z = a\ \mathsf{in}\ e$$

we would like to group the two bindings of a together, because then deduplication (Section 5.5) can get rid of one of them.

We achieve this very naively by partially sorting chains of let-bindings according to the expression to which they assign a name. This is implemented with the help of the annotation framework from Section 5.3 as follows.

First, we have a single and very simple micro-transformation:

$$\begin{aligned}
&\mathit{ecommute} \in \mathsf{Exp} \rightharpoonup \mathsf{Exp} \\
&\mathit{ecommute}(e) := \mathsf{let}\ y = b\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ e_0 \\
&\qquad\qquad \text{if } e = (\mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ e_0) \text{ and } x \notin \mathrm{fv}(b) \\
&\mathit{ecommute}(e) := \mathsf{let}\ k_2 = \mathsf{cont}\ y_2.\, e_2\ \mathsf{in}\ \mathsf{let}\ k_2 = \mathsf{cont}\ y_2.\, e_2\ \mathsf{in}\ e_0 \\
&\qquad\qquad \text{if } e = (\mathsf{let}\ k_1 = \mathsf{cont}\ y_1.\, e_1\ \mathsf{in}\ \mathsf{let}\ k_2 = \mathsf{cont}\ y_2.\, e_2\ \mathsf{in}\ e_0) \text{ and } k_1 \notin \mathrm{fv}(e_2)
\end{aligned}$$

Note that the side condition on x (respectively k_1) is crucial for correctness (we will sketch the proof of correctness in a moment).

Second, we have an analysis α that marks any let-binding that needs to be swapped with its succeeding one. More formally, it places annotations that are elements of $\mathit{1}_\perp$ and the transformer that consumes them is the lifting $\beta_\perp$ of the following:

$$\begin{aligned}
&\beta \in \mathit{1} \to \mathsf{Exp} \rightharpoonup \mathsf{Exp} \\
&\beta(1)(e) := \mathit{ecommute}(e)
\end{aligned}$$

Accordingly, the annotation $\perp$ means "do nothing here", whereas the annotation 1 means "apply *ecommute* here".

The details of the analysis are not very interesting (and importantly, as far as semantics preservation is concerned, they are irrelevant). The analysis recursively descends into the given function expression, for which the interesting case is, of course, a let-binding. Let us consider a regular let-binding:

$$\mathsf{let}\ y_1 = a_1\ \mathsf{in}\ e_1$$

We recursively analyze a_1 and e_1, yielding $\mathscr{a}_1$ and $\mathscr{e}_1$. Now we want to know whether $etf^{\beta_\perp}(\mathscr{e}_1)$ (*i.e.,* the result of transforming $\mathscr{e}_1$), is another let-binding $\mathsf{let}\ y_2 = a_2\ \mathsf{in}\ e_2$ and if so, produce a $\perp$ or $\mathsf{1}$ annotation depending on whether a_2 is "less"[2] than a_1.

The full commute pass $commute \in \mathbf{Mod} \to \mathbf{Mod}$ is then defined as $commute := tf_\alpha^{\beta_\perp}$.

5.4.2 Verification

In order to establish the correctness of $tf_\alpha^{\beta_\perp}$, it suffices by Theorems 19 and 20 to show the correctness of β. The syntactic aspect of this is straightforward. We sketch the semantic aspect, *i.e.,* the proof of the following:

$$\forall l, \Gamma, e, e', w.\quad \Gamma \vdash e \wedge uniq(\mathrm{bv}(e), \Gamma) \wedge \mathsf{commute}(e) = e' \wedge w \in \mathsf{stable}(\Omega_{\mathcal{II}}) \implies \Gamma \vdash e' \precsim_w^* e$$

Here we consider the case of let-bindings for term variables (the argument for continuations is analogous). So suppose $e = (\mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ e_0)$ with $x \notin \mathrm{fv}(b)$ and $e' = (\mathsf{let}\ y = b\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ e_0)$. It suffices to show[3] $((\emptyset, (\sigma', e')), (\emptyset, (\sigma, e))) \in \mathbf{E}$, where the environments σ' and σ contain related values for variables in Γ.

If the evaluation of a or that of b in the source fails, then the source produces an error and there is nothing more to show. So let us assume that a evaluates in σ to v_a and that b evaluates in $\sigma, x{\mapsto}v_a$ to v_b. By the well-formedness of e we know $\Gamma, x \vdash b$. But since $x \notin \mathrm{fv}(b)$, we can strengthen this to $\Gamma \vdash b$, for which we get $\Gamma \vdash b \precsim_w b$ by Lemma 70. Note that since $x \notin \Gamma$ (by the uniqueness condition), the environments σ' and $\sigma, x{\mapsto}v_a$ are still related for Γ. Thus we learn that the target's evaluation of b in σ' evaluates to some v_b' that is related to v_b, extending σ' to $\sigma', y{\mapsto}v_b'$.

Now we use $\Gamma \vdash a$ to get $\Gamma \vdash a \precsim_w a$. Much as before, note that since $y \notin \Gamma$, we know that $\sigma', y{\mapsto}v_b'$ and σ are related for Γ as well. Hence the target program's evaluation of a in $\sigma', y{\mapsto}v_b'$ evaluates to some v_a' that is related to v_a. Using Lemmas 43, 47, 48, 44, it remains to show the resulting programs related:

$$((\emptyset, ((\sigma', y{\mapsto}v_b', x{\mapsto}v_a'), e_0)), (\emptyset, ((\sigma, x{\mapsto}v_a, y{\mapsto}v_b), e_0))) \in \mathbf{E}$$

It is easy to see that the two environments above are related for Γ, x, y, because σ' and σ are related, v_a' and v_a are related, v_b' and v_b are related, and moreover $x \neq y$ by uniqueness. Therefore we are done by instantiating $\Gamma, x, y \vdash e_0 \precsim_w e_0$, which we get from $\Gamma, x, y \vdash e_0$.

[2] In the Coq development, we do not fix an order on expressions but parameterize over a comparison operator. In the extracted Ocaml code, we implement the comparison with the help of a hash table, thus creating the order on the fly.

[3] Here and in subsequent proof sketches, we often gloss over states, worlds, etc. for the sake of readability.

5.4.3 Alternative Implementation.

There is a somewhat simpler way to implement the commute pass than what we described in Section 5.4.1. The idea is to move the expression comparison into the micro-transformation:

$$\begin{aligned}
&\mathit{ecommute}' \in \mathsf{exp} \rightharpoonup \mathsf{exp} \\
&\mathit{ecommute}'(e) := \mathsf{let}\ x_2 = a_2\ \mathsf{in}\ \mathsf{let}\ x_1 = a_1\ \mathsf{in}\ e_2 \\
&\qquad\qquad \text{if } e = (\mathsf{let}\ x_1 = a_1\ \mathsf{in}\ \mathsf{let}\ x_2 = a_2\ \mathsf{in}\ e_2) \text{ and } x \notin \mathrm{fv}(b) \text{ and } a_2 < a_1
\end{aligned}$$

This is of course still correct, but now the analysis can be trivial: mark *every* expression (so we can use 1 instead of $1_\bot$).

We prefer the first version because it is more general and might be useful in other contexts.

5.5 The Dedup Pass

5.5.1 Transformation

Deduplication is also implemented using the annotation framework. We use the following micro-transformation:

$$\begin{aligned}
&\mathit{ededup} \in \mathsf{Exp} \rightharpoonup \mathsf{Exp} \\
&\mathit{ededup}(e) := \mathsf{let}\ x = a\ \mathsf{in}\ e_0[x/y] \\
&\qquad\qquad \text{if } e = (\mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = a\ \mathsf{in}\ e_0) \\
&\mathit{ededup}(e) := \mathsf{let}\ k_1 = \mathsf{cont}\ y.\, e_1\ \mathsf{in}\ e_0[k_1/k_2] \\
&\qquad\qquad \text{if } e = (\mathsf{let}\ k_1 = \mathsf{cont}\ y.\, e_1\ \mathsf{in}\ \mathsf{let}\ k_2 = \mathsf{cont}\ y.\, e_1\ \mathsf{in}\ e_0)
\end{aligned}$$

As annotations we choose the trivial set (1) and, accordingly, the trivial transformer β and the trivial analysis α. The complete deduplication pass then is defined as $\mathit{dedup} := \mathit{tf}^\beta_\alpha$, *i.e.,* we apply the micro-transformation wherever possible.

5.5.2 Verification

In order to establish the correctness of tf^β_α, it suffices by Theorem 19 to show the correctness of β. The syntactic aspect of this is straightforward. We sketch the semantic aspect, *i.e.,* the proof of the following:

$$\begin{aligned}
\forall l, \Gamma, e, e', w.\quad & \Gamma \vdash e \ \wedge\ \mathit{uniq}(\mathrm{bv}(e), \Gamma) \ \wedge\ \mathit{ededup}(e) = e' \ \wedge\ w \in \mathsf{stable}(\Omega_{\mathcal{II}}) \implies \\
& \Gamma \vdash e' \precsim^*_w e
\end{aligned}$$

Here we consider the case of let-bindings for term variables (the argument for continuations is analogous). So suppose $e = (\mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = a\ \mathsf{in}\ e_0)$ and $e' = (\mathsf{let}\ x = a\ \mathsf{in}\ e_0[x/y])$. It suffices to show $((\emptyset, (\sigma', e')), (\emptyset, (\sigma, e))) \in \mathbf{E}$, where the environments σ' and σ contain related values for variables in Γ.

If the first or second evaluation of a in the source fails, then the source produces an error and there is nothing more to show. So let us assume that a first evaluates in σ to v_1 and then in $\sigma, x{\mapsto}v_1$ to v_2.

From $\Gamma \vdash a$ we get $\Gamma \vdash a \precsim_w a$ by Lemma 70. Since σ' and σ are related for Γ, we thus know that the target evaluation of a in σ' results in a value v' related to v_1. But note that σ' and $\sigma, x{\mapsto}v_1$ are also related for Γ because $x \notin \Gamma$ by uniqueness, hence v' is also related to v_2 (evaluation is deterministic). Using Lemmas 43, 47, 48, 44, it remains to show the resulting programs related:

$$((\emptyset, ((\sigma', x{\mapsto}v'), e_0[x/y])), (\emptyset, ((\sigma, x{\mapsto}v_1, y{\mapsto}v_2), e_0))) \in \mathbf{E}$$

Now we apply Lemma 69 to $\Gamma, x, y \vdash e_0$ (*not* Lemma 70), which yields $\Gamma, x, y \vdash e_0[x/y] \precsim_w^{[x/y]} e_0$. To show the goal, it thus suffices to show the environments above related for Γ, x, y *relative* to the renaming $[x/y]$. This holds essentially because v' is related to both v_1 and v_2.

5.6 The Hoist Pass

5.6.1 Transformation

Hoisting is also implemented using the annotation framework. We define the complete pass as tf_α^β, where—like for deduplication—α is the trivial analysis for *1* and β the trivial transformer $\boldsymbol{\lambda} l.\, \mathsf{hoist}$. The underlying micro-transformation is defined as follows:

$$
\begin{aligned}
&\mathit{ehoist} \in \mathsf{Exp} \rightharpoonup \mathsf{Exp} \\
&\mathit{ehoist}(e) := \mathsf{let}\ y_2 = a\ \mathsf{in}\ \mathsf{let}\ y_1 = (\mathsf{fix}\, f(y_3, k).\, e_1)\ \mathsf{in}\ e_2 \\
&\qquad\qquad \text{if } e = (\mathsf{let}\ y_1 = (\mathsf{fix}\, f(y_3, k).\, \mathsf{let}\ y_2 = a\ \mathsf{in}\ e_1)\ \mathsf{in}\ e_2) \text{ and } f, y_3, k \notin \mathrm{fv}(a) \\
&\qquad\qquad \text{and } a \notin \{x_1 \odot x_2, x_1 == x_2, x.1, x.2\} \\
&\mathit{ehoist}(e) := \mathsf{let}\ y_2 = a\ \mathsf{in}\ \mathsf{let}\ y_1 = (\Lambda k.\, e_1)\ \mathsf{in}\ e_2 \\
&\qquad\qquad \text{if } e = (\mathsf{let}\ y_1 = (\Lambda k.\, \mathsf{let}\ y_2 = a\ \mathsf{in}\ e_1)\ \mathsf{in}\ e_2) \text{ and } k \notin \mathrm{fv}(a) \\
&\qquad\qquad \text{and } a \notin \{x_1 \odot x_2, x_1 == x_2, x.1, x.2\}
\end{aligned}
$$

The restriction of a to certain forms is a syntactic criterion ensuring that the evaluation of a will succeed (yield a value). It is easy to see that *ehoist* would not be correct in general if the evaluation of a could fail.

Here we do not bother to hoist out of continuations because continuations are only executed at most once anyways.

Note that we cannot not hoist continuation definitions (only regular let-bindings) out of functions. This is because the resulting program would generally be ill-formed due to functions not being able to access continuations from a surrounding scope (Section 4.2.4.2).

5.6.2 Verification

In order to establish the correctness of tf^{β}_{α}, it suffices by Theorem 19 to show the correctness of β. The syntactic aspect of this is straightforward. We sketch the semantic aspect, *i.e.,* the proof of the following:

$$\forall l, \Gamma, e, e', w. \quad \Gamma \vdash e \wedge \mathit{uniq}(\mathrm{bv}(e), \Gamma) \wedge \mathit{ehoist}(e) = e' \wedge w \in \mathsf{stable}(\Omega_{\mathcal{II}}) \implies \Gamma \vdash e' \precsim^{*}_{w} e$$

Here we consider the second case in the definition of *ehoist*, *i.e.,* we hoist out of a generalization (the first case is analogous). So suppose that:

$$e = (\mathsf{let}\ y_1 = (\Lambda k.\, \mathsf{let}\ y_2 = a\ \mathsf{in}\ e_1)\ \mathsf{in}\ e_2)$$
$$e' = (\mathsf{let}\ y_2 = a\ \mathsf{in}\ \mathsf{let}\ y_1 = (\Lambda k.\, e_1)\ \mathsf{in}\ e_2)$$

It suffices to show $((\emptyset, (\sigma', e')), (\emptyset, (\sigma, e))) \in \mathbf{E}$ for environments σ' and σ related relative to Γ.

Regarding the source program e, we know that $\Lambda k.\, \mathsf{let}\ y_2 = a\ \mathsf{in}\ e_1$ evaluates in σ to $v_g := \langle \sigma; \Lambda k.\, \mathsf{let}\ y_2 = a\ \mathsf{in}\ e_1 \rangle$. Regarding the target program e', we know that a evaluates in σ' to some v' thanks to the syntactic restriction of a. Afterwards, $\Lambda k.\, e_1$ evaluates to $v'_g := \langle \sigma', y_2 \mapsto v'; \Lambda k.\, e_1 \rangle$. Using Lemmas 43, 47, 48, 44, it remains to show the resulting programs related:

$$((\emptyset, ((\sigma', y_2 \mapsto v', y_1 \mapsto v'_g), e_2)), (\emptyset, ((\sigma, y_1 \mapsto v_g), e_2))) \in \mathbf{E}$$

From $\Gamma, y_1 \vdash e_2$ we get $\Gamma, y_1 \vdash e_2 \precsim_w e_2$ by Lemma 70. To show the goal, it thus suffices to show the environments above related for Γ, y_1. This boils down to showing the closures v'_g and v_g related.

Considering applications of those to continuations v'_k and v_k, respectively, we must show:

$$((\emptyset, ((\sigma', y_2 \mapsto v', k \mapsto v'_k), e_1)), (\emptyset, ((\sigma, k \mapsto v_k), \mathsf{let}\ y_2 = a\ \mathsf{in}\ e_1))) \in \mathbf{E}(v'_k, v_k)$$

By Lemmas 43, 47, 44 this reduces to

$$((\emptyset, ((\sigma', y_2 \mapsto v', k \mapsto v'_k), e_1)), (\emptyset, ((\sigma, k \mapsto v_k, y_2 \mapsto v), e_1))) \in \mathbf{E}(v'_k, v_k)$$

where v is the result of evaluating a in $\sigma, k \mapsto v_k$.

From $\Gamma^{\dagger}, k, y_2 \vdash e_1$ we get $\Gamma^{\dagger}, k, y_2 \vdash e_1 \precsim_w e_1$ by Lemma 70. To show the goal, it thus suffices to show the above environments related for $\Gamma^{\dagger}, k, y_2$ and initial continuations v'_k and v_k. Relying on uniqueness, this means showing the following:

1. σ' and σ are related environments for $\Gamma^{\dagger}$ with initial continuations v'_k, v_k.
2. v' and v are related values.
3. v'_k and v_k are related continuations relative to themselves.

Goal (1) follows from their relatedness for Γ because $\Gamma^\dagger$ does not contain any continuations. Goal (3) holds by Lemma 52. It remains to show (2).

From $\Gamma, k \vdash a$ and $k \notin \mathrm{fv}(a)$ we know $\Gamma \vdash a$ and thus get $\Gamma \vdash a \precsim_w a$ by Lemma 70. Since σ' is related to $\sigma, k{\mapsto}v_k$ for Γ ($k \notin \Gamma$ by uniqueness), we know that v', the result of evaluating a in σ', is related to v.

5.7 The Dead Code Elimination Pass

Pilsner's dead code elimination applies to let-bindings, read operations, and allocations. It removes these operations if the variable that they introduce does not occur in the rest of the program. This is safe because these operations have no observable side effects (even though some of them concern memory).

5.7.1 Transformation

Dead code elimination (DCE) is implemented directly, not using the annotation framework. This is because the framework was developed only after dead code elimination was already finished. In its current form, the framework cannot be used to implement DCE because proving correct the elimination of allocations requires a custom local world (cf. the comment in Section 5.3.2.2).

Figure 5.7 shows the definition of the module-level transformation *dce* in terms of a recursive helper function *mdce* that applies *adce* to each top-level function. *adce* in turn is defined in Figure 5.8 over the structure of pure expressions, in mutual recursion with *edce* for control expressions. The interesting cases are the first four of *edce* (those in Figure 5.7), all others are merely structural.

edce determines whether to perform an elimination by checking if the introduced variable occurs freely in the rest of the program. In order for the transformation to be idempotent, it is crucial, however, that we first transform the rest of the program, *i.e.,* that we do the occurrence check on the free variables of the *transformed* rest of the program (which may be fewer). For efficiency, *adce* and *edce* additionally return (an over-approximation of) this set, so that it doesn't need to be recomputed all the time: if $(e', Z) = \mathit{edce}(e)$, then $\mathrm{fv}(e') \subseteq Z$ (and similarly for *adce*).

5.7.2 Verification

We use the following local world, whose states are heaps on the source side.

$$
\begin{array}{lcl}
w.\mathsf{T}.\mathsf{S} & := & \mathsf{Heap} \\
w.\mathsf{T}.\sqsupseteq & := & \mathsf{Heap} \times \mathsf{Heap} \\
w.\mathsf{T}.\sqsupseteq_{\mathrm{pub}} & := & \mathsf{Heap} \times \mathsf{Heap} \\
w.\mathsf{C}(G)(s_{\mathrm{g}}, h) & := & \{\, (\emptyset, (h, \emptyset)) \,\} \\
w.\mathsf{O} & := & \mathrm{gwf}
\end{array}
$$

The states are only relevant for the elimination of allocations. The idea is very simple: we need to consider any allocation in the source program that got eliminated

$$
\begin{array}{l}
dce \in \mathbf{Mod} \to \mathbf{Mod} \\
dce(\epsilon) \qquad\quad = \epsilon \\
dce((F, a), M) = \mathit{let}\ (a', \_) := adce(a)\ \mathit{in}\ (F, a'), dce(M)
\end{array}
$$

$$
edce \in \mathsf{Exp} \to \mathsf{Exp} \times \mathsf{Var}
$$

$$
\begin{array}{l}
edce(\mathsf{let}\ y = a\ \mathsf{in}\ e) = \\
\quad \mathit{let}\ (e', Z_1) := edce(e)\ \mathit{in} \\
\quad \mathit{if}\ y \in Z_1\ \mathit{then} \\
\quad\quad \mathit{let}\ (a', Z_2) := adce(a)\ \mathit{in} \\
\quad\quad (\mathsf{let}\ y = a'\ \mathsf{in}\ e', Z_1 \cup Z_2) \\
\quad \mathit{else}\ (e', Z_1)
\end{array}
$$

$$
\begin{array}{l}
edce(\mathsf{let}\ k = \mathsf{cont}\ y.\, e_1\ \mathsf{in}\ e_2) = \\
\quad \mathit{let}\ (e_2', Z_1) := edce(e_2)\ \mathit{in} \\
\quad \mathit{if}\ k \in Z_1\ \mathit{then} \\
\quad\quad \mathit{let}\ (e_1', Z_2) := edce(e_1)\ \mathit{in} \\
\quad\quad (\mathsf{let}\ k = \mathsf{cont}\ y.\, e_1'\ \mathsf{in}\ e_2', Z_1 \cup Z_2) \\
\quad \mathit{else}\ (e_2', Z_1)
\end{array}
$$

$$
\begin{array}{l}
edce(y \leftarrow \mathsf{ref}\ x;\ e) = \\
\quad \mathit{let}\ (e', Z) := edce(e)\ \mathit{in} \\
\quad \mathit{if}\ y \in Z\ \mathit{then} \\
\quad\quad (y \leftarrow \mathsf{ref}\ x;\ e', \{x\} \cup Z) \\
\quad \mathit{else}\ (e', Z)
\end{array}
$$

$$
\begin{array}{l}
edce(y \leftarrow !x;\ e) = \\
\quad \mathit{let}\ (e', Z) := edce(e)\ \mathit{in} \\
\quad \mathit{if}\ y \in Z\ \mathit{then} \\
\quad\quad (y \leftarrow !x;\ e', \{x\} \cup Z) \\
\quad \mathit{else}\ (e', Z)
\end{array}
$$

Figure 5.7: DCE transformation (part 1 of 2).

$$
\begin{array}{l}
edce(y \leftarrow \mathsf{input};\ e) = \\
\quad let\ (e', Z) := edce(e)\ in \\
\quad (y \leftarrow \mathsf{input};\ e', Z) \\
\\
edce(\mathsf{output}\ x;\ e) = \\
\quad let\ (e', Z) := edce(e)\ in \\
\quad (\mathsf{output}\ x;\ e', \{x\} \cup Z) \\
\\
edce(\mathsf{ifnz}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) = \\
\quad let\ (e_1', Z_1) := edce(e_1)\ in \\
\quad let\ (e_2', Z_2) := edce(e_2)\ in \\
\quad (\mathsf{ifnz}\ x\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2', \{x\} \cup Z_1 \cup Z_2) \\
\\
edce(\mathsf{case}\ x\ (y.\ e_1)\ (y.\ e_2)) = \\
\quad let\ (e_1', Z_1) := edce(e_1)\ in \\
\quad let\ (e_2', Z_2) := edce(e_2)\ in \\
\quad (\mathsf{case}\ x\ (y.\ e_1')\ (y.\ e_2'), \{x\} \cup Z_1 \cup Z_2) \\
\\
edce(x_1\ x_2\ k) = \\
\quad (x_1\ x_2\ k, \{x_1, x_2, k\}) \\
\\
edce(x\ []\ k) = \\
\quad (x\ []\ k, \{x, k\}) \\
\\
edce(k\ x) = \\
\quad (k\ x, \{k, x\}) \\
\\
edce(x_1 := x_2;\ e) = \\
\quad let\ (e', Z) := edce(e)\ in \\
\quad (x_1 := x_2;\ e', \{x_1, x_2\} \cup Z)
\end{array}
$$

$$
\begin{array}{l}
adce \in \mathsf{BExp} \to \mathsf{BExp} \times \mathsf{Var} \\
adce(\langle\rangle) = (\langle\rangle, \emptyset) \\
adce(n) = (n, \emptyset) \\
adce(x.1) = (x.1, \{x\}) \\
adce(x.2) = (x.2, \{x\}) \\
adce(\mathsf{inl}\ x) = (\mathsf{inl}\ x, \{x\}) \\
adce(\mathsf{inr}\ x) = (\mathsf{inr}\ x, \{x\}) \\
adce(\langle x_1, x_2\rangle) = (\langle x_1, x_2\rangle, \{x_1, x_2\}) \\
adce(x_1 \odot x_2) = (x_1 \odot x_2, \{x_1, x_2\}) \\
adce(x_1 == x_2) = (x_1 == x_2, \{x_1, x_2\}) \\
adce(\mathsf{fix}\, f(y,k).\, e) = let\ (e', Z) := edce(e)\ in\ (\mathsf{fix}\, f(y,k).\, e', Z) \\
adce(\Lambda k.\, e) = let\ (e', Z) := edce(e)\ in\ (\Lambda k.\, e', Z)
\end{array}
$$

Figure 5.8: DCE transformation (part 2 of 2).

in the target program as being local, because otherwise we would be forced to provide a target location corresponding to the freshly allocated source location. For convenience we allow arbitrary transitions between the states—this is okay precisely because the heap locations in these states won't ever be used by the programs.

The correctness proof for *dce* boils down to showing

$$\Gamma \vdash e' \wedge \mathit{uniq}(\mathrm{bv}(e), \Gamma) \wedge (e', \_) = \mathit{edce}(e) \implies \Gamma \vdash e' \precsim_w e$$

(and similarly for *adce*). We do this by induction on the structure of the input program. Note that we assume the well-formedness of e' in Γ, not that of e in Γ. This is somewhat unusual but important for the induction to go through, as we will see in a moment.

Let us sketch the allocation case here. The other interesting cases are analogous but a little simpler because they do not involve any important state transitions. The purely structural cases are trivial to handle using the compatibility lemmas.

Suppose $e = (y \leftarrow \mathsf{ref}\ x;\ e_0)$. We must show $\Gamma \vdash e' \precsim_w e$. Let $(e_0', Z_0) := \mathit{edce}(e_0)$. If the allocation is not eliminated ($y \in Z_0$), the goal equals

$$\Gamma \vdash (y \leftarrow \mathsf{ref}\ x;\ e_0') \precsim_w (y \leftarrow \mathsf{ref}\ x;\ e_0)$$

and we are done using the inductive hypothesis and the compatibility lemma for allocation.

Now suppose $y \notin Z_0$ such that the elimination is performed. Then the goal equals $\Gamma \vdash e_0' \precsim_w (y \leftarrow \mathsf{ref}\ x;\ e_0)$. After unfolding the definitions and using Lemma 43, we must show

$$\big((h', (\sigma', e_0')\big), \big(h, (\sigma, (y \leftarrow \mathsf{ref}\ x;\ e_0))\big)\big) \in \mathbf{E}(s_\mathrm{g}, s)$$

where the environments σ' and σ are related for Γ and the heaps consist of a related global part, a related local part, and a frame part. By construction of w, the local part of h' is empty and that of h equals the local state s.

We now use Lemma 47 to take a step on the source side so that it remains to show

$$\big((h', (\sigma', e_0')\big), \big(h \sqcup [l \mapsto \sigma(x)], ((\sigma, y{\mapsto}l), e_0)\big)\big) \in \mathbf{E}(s_\mathrm{g}, s)$$

(for some fresh location l).

Now we use Lemma 50 to advance the world's state (s_g, s) to $(s_\mathrm{g}, s \sqcup [l \mapsto \sigma(x)])$ so that it matches the new heap. That is, we interpret the change to h as a change to its local part (which happens to be s) and pick a new local state accordingly. This enables us to use Lemma 44 such that the goal becomes

$$\big((\emptyset, (\sigma', e_0')\big), \big(\emptyset, ((\sigma, y{\mapsto}l), e_0)\big)\big) \in \mathbf{E}(s_\mathrm{g}, s \sqcup [l \mapsto \sigma(x)]).$$

Since $\Gamma, y \vdash e_0'$ and $y \notin Z_0 \supseteq \mathrm{fv}(e_0')$, we know $\Gamma \vdash e_0'$. The inductive hypothesis then yields $\Gamma \vdash e_0' \precsim_w e_0$. To instantiate this and establish the goal, we must show that σ' is related to $\sigma, y{\mapsto}l$ for Γ, which follows from relatedness of σ' to σ and uniqueness ($y \notin \Gamma$). We are done.

What if we had started out with $\Gamma \vdash e$ instead of $\Gamma \vdash e'$? Then we know $\Gamma, y \vdash e_0$ but we cannot not derive $\Gamma \vdash e_0$ because we only know that y is not free in e'_0—it may very well be free in e_0 (inside dead code). So instead of $\Gamma \vdash e'_0 \precsim_w e_0$, we would only get the weaker $\Gamma, y \vdash e'_0 \precsim_w e_0$, which we would not be able to instantiate because y does not exist in the target program.

5.8 The Inline Pass

5.8.1 Transformation

Pilsner's inlining pass is defined in Figure 5.9. The auxiliary functions *einline* and *ainline* operate on control expressions and basic expressions, respectively. They take a mapping ψ containing the functions that are to be inlined (mapping their identifiers to their code). The more interesting one is *einline*: when it sees a call to one of the functions in ψ, it performs the actual inlining by replacing the call with the function's body and substituting the respective variables (note that this works even for recursive functions). Note that inlining is intentionally not an idempotent transformation.

minline lifts these operations to modules in a straightforward way. In the final transformation, *inline*, we decide for simplicity to just inline all top-level functions of a module by picking $\psi := M$. This is of course somewhat unrealistic but does not reflect a real limitation: the "analysis" that computes ψ can be made arbitrarily complex without affecting the correctness of the transformation—similar to the analyses in the previous transformations. Also, an extension of the transformation and its verification to local functions (functions not defined at the top-level) is expected to be straightforward.

5.8.2 Verification

Unfortunately, the tools from the previous sections do not suffice for verifying the inlining pass. Consider the correctness proof of *einline*. Along the lines of the previous verifications, we would have to show something like

$$\forall e, \psi.\ \ \Gamma \vdash e \implies \Gamma \vdash \mathit{einline}(\psi)(e) \precsim_w^* e$$

But this is clearly not possible. Suppose e gets executed in an environment σ and $\mathit{einline}(\psi)(e)$ gets executed in a related environment σ'. If *einline* inlines a function in e, it does so by replacing functions calls according to ψ. However, there is no connection between this inlining information ψ and the environment σ in which the original program will lookup the function.

To address this problem, we slightly generalize the $\precsim$ relations by considering only environments that match given *descriptions* ψ_{a} and ψ_{b}, respectively (of the same type as ψ above).

$$
\begin{array}{l}
\mathit{inline} \in \mathbf{Mod} \to \mathbf{Mod} \\
\mathit{inline}(M) = \mathit{freshen}(\mathit{minline}(M)(M)) \\
\\
\mathit{minline} \in (\mathsf{Var} \rightharpoonup \mathsf{BExp}) \to \mathbf{Mod} \to \mathbf{Mod} \\
\mathit{minline}(\psi)(\epsilon) \qquad\quad = \epsilon \\
\mathit{minline}(\psi)((F, a), M) = (F, \mathit{ainline}(\psi)(a)), \mathit{minline}(\psi)(M) \\
\\
\mathit{einline} \in (\mathsf{Var} \rightharpoonup \mathsf{BExp}) \to \mathsf{Exp} \to \mathsf{Exp} \\
\mathit{einline}(\psi)(x_1\ x_2\ k) \quad := \begin{cases} e[x_1/f][x_2/y][k/k'] & \text{if } \psi(x_1) = \mathsf{fix}\, f(y, k').\, e \\ x_1\ x_2\ k & \text{otherwise} \end{cases} \\
\mathit{einline}(\psi)(x\ []\ k) \quad := \begin{cases} e[k/k'] & \text{if } \psi(x) = \Lambda k'.\, e \\ x\ []\ k & \text{otherwise} \end{cases} \\
\mathit{einline}(\psi)(\mathsf{let}\ y = a\ \mathsf{in}\ e) := \mathsf{let}\ y = \mathit{ainline}(\psi)(a)\ \mathsf{in}\ \mathit{einline}(\psi)(e) \\
\ldots \\
\\
\mathit{ainline} \in (\mathsf{Var} \rightharpoonup \mathsf{BExp}) \to \mathsf{BExp} \to \mathsf{BExp} \\
\mathit{ainline}(\psi)(\mathsf{fix}\, f(y, k).\, e) := \mathsf{fix}\, f(y, k).\, \mathit{einline}(\psi)(e) \\
\ldots
\end{array}
$$

Figure 5.9: Inlining of top-level functions.

Definition 47.

$$
\begin{array}{l}
\Gamma; \psi_\mathsf{a}; \psi_\mathsf{b} \vdash e_\mathsf{a} \precsim^{\phi}_{w} e_\mathsf{b} := \\
\quad \exists i.\ \forall G, s_0, s, \sigma_\mathsf{a}, \sigma_\mathsf{b}, j, \mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b}. \\
\qquad G \in \mathbf{GK}_{w\uparrow} \wedge s \sqsupseteq_{\mathrm{pub}} s_0 \wedge \mathsf{respects}(\sigma_\mathsf{a}, \psi_\mathsf{a}, \Gamma) \wedge \mathsf{respects}(\sigma_\mathsf{b}, \psi_\mathsf{b}, \Gamma) \wedge \\
\qquad (\forall x \in \Gamma.\ \exists \mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}.\ \sigma_\mathsf{a}(\phi(x)) = \mathbf{v}_\mathsf{a} \wedge \sigma_\mathsf{b}(x) = \mathbf{v}_\mathsf{b} \wedge (\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in \overline{G}(s)) \wedge \\
\qquad (\forall k \in \Gamma.\ \exists \mathbf{k}'_\mathsf{a}, \mathbf{k}'_\mathsf{b}.\ \sigma_\mathsf{a}(\phi(k)) = \mathbf{k}'_\mathsf{a} \wedge \sigma_\mathsf{b}(k) = \mathbf{k}'_\mathsf{b} \wedge \\
\qquad\quad (\mathbf{k}'_\mathsf{a}, \mathbf{k}'_\mathsf{b}) \in \mathbf{K}_{w\uparrow}(j)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)) \\
\quad \implies ((\emptyset, (\sigma_\mathsf{a}, e_\mathsf{a})), (\emptyset, (\sigma_\mathsf{b}, e_\mathsf{b}))) \in \mathbf{E}_{w\uparrow}(i + j)(\mathbf{k}_\mathsf{a}, \mathbf{k}_\mathsf{b})(G)(s_0)(s)(\bot)
\end{array}
$$

Definition 48.

$$
\begin{array}{l}
\Gamma; \psi_\mathsf{a}; \psi_\mathsf{b} \vdash a_\mathsf{a} \precsim^{\phi}_{w} a_\mathsf{b} := \\
\quad \forall G, s, \sigma_\mathsf{a}, \sigma_\mathsf{b}, \mathbf{v}_\mathsf{b}. \\
\qquad G \in \mathbf{GK}_{w\uparrow} \wedge \mathsf{respects}(\sigma_\mathsf{a}, \psi_\mathsf{a}, \Gamma) \wedge \mathsf{respects}(\sigma_\mathsf{b}, \psi_\mathsf{b}, \Gamma) \wedge \\
\qquad (\forall x \in \Gamma.\ \exists \mathbf{v}'_\mathsf{a}, \mathbf{v}'_\mathsf{b}.\ \sigma_\mathsf{a}(\phi(x)) = \mathbf{v}'_\mathsf{a} \wedge \sigma_\mathsf{b}(x) = \mathbf{v}'_\mathsf{b} \wedge (\mathbf{v}'_\mathsf{a}, \mathbf{v}'_\mathsf{b}) \in \overline{G}(s)) \wedge \\
\qquad [\![a_\mathsf{b}]\!]_{\sigma_\mathsf{b}} = \mathbf{v}_\mathsf{b} \\
\quad \implies \exists \mathbf{v}_\mathsf{a}.\ [\![a_\mathsf{a}]\!]_{\sigma_\mathsf{a}} = \mathbf{v}_\mathsf{a} \wedge (\mathbf{v}_\mathsf{a}, \mathbf{v}_\mathsf{b}) \in \overline{G}(s)
\end{array}
$$

The differences to the earlier definitions are highlighted. The ψ_a and ψ_b components can be thought of as static descriptions of (parts of) the runtime environments. Naturally, we obtain the original relations when plugging in empty ψ_a and ψ_b. For

the concrete purpose of verifying inlining, ψ_{a} will be empty but ψ_{b} will be the function mapping used by *einline* and *ainline*. We will see a use case for a non-empty ψ_{a} in Section 5.9.

A runtime environment σ *respects* such a description, written $\mathsf{respects}(\sigma, \psi, \Gamma)$, if it contains matching closures:

Definition 49.

$$\begin{aligned}\mathsf{respects}(\sigma, \psi, \Gamma) := \forall z, e.\ \psi(z) = e \implies \exists \sigma'.\\ z \in \Gamma \wedge \sigma(z) = \langle \sigma'; e \rangle \wedge\\ \forall z' \in \mathsf{defBefore}(z, \Gamma).\ \sigma'(z') = \sigma(z')\end{aligned}$$

This says that whenever ψ maps a variable z to an expression e, the actual runtime environment σ maps z to a closure value whose code component is e. Moreover, the environment σ' stored in that closure must agree with σ on all variables *defined before* z, *i.e.,* those variables that syntactically occur to the left of z in Γ. (In our concrete case of inlining, σ will always be an extension of any such σ'.)

It is easy to see that the new relation generalizes the old one:

Lemma 73.

$$\frac{\Gamma \vdash e_{\mathsf{a}} \precsim^{\phi}_{w} e_{\mathsf{b}}}{\Gamma; \psi_{\mathsf{a}}; \psi_{\mathsf{b}} \vdash e_{\mathsf{a}} \precsim^{\phi}_{w} e_{\mathsf{b}}} \qquad \frac{\Gamma; \emptyset; \emptyset \vdash e_{\mathsf{a}} \precsim^{\phi}_{w} e_{\mathsf{b}}}{\Gamma \vdash e_{\mathsf{a}} \precsim^{\phi}_{w} e_{\mathsf{b}}}$$

With its help we can establish the following correctness statement about the function call case of *einline* (and similarly for $x\ [\!]\ k$).

Lemma 74.

$$\frac{\begin{array}{c} x_1, x_2, k \in \Gamma \\ \psi(x_1) = \mathsf{fix}\, f(y, k').\, e \\ \{x_1, x_2, k\} \cap \mathrm{bv}(e) = \emptyset \\ \mathsf{defBefore}(x_1, \Gamma) \vdash \mathsf{fix}\, f(y, k').\, e \\ \mathsf{stable}(w) \end{array}}{\Gamma; \epsilon; \psi \vdash \mathit{einline}(\psi)(x_1\ x_2\ k) \precsim^{*}_{w} x_1\ x_2\ k}$$

Observe that the inlining information ψ used to transform the call is now connected to whatever runtime environment will be given for the execution of $x_1\ x_2\ k$. Also note that we require the well-formedness of the function not just in Γ but in the restriction $\mathsf{defBefore}(x_1, \Gamma)$. Intuitively, this is the environment in which the function was defined, and using it here ensures compatibility with whatever runtime environment the closure of f will contain (in the source program execution). The proof sketch goes as follows.

Proof. After unfolding the definitions, the goal is to show $(\sigma_{\mathsf{a}}, e[x_1/f][x_2/y][k/k'])$ related to $(\sigma_{\mathsf{b}}, x_1\ x_2\ k)$ by $\mathbf{E}$ (ignoring some uninteresting details here). Since $\mathsf{respects}(\sigma_{\mathsf{b}}, \psi, \Gamma)$, we know that $\sigma_{\mathsf{b}}(x_1) = \langle \sigma'_{\mathsf{b}}; \mathsf{fix}\, f(y, k').\, e \rangle$ for some closure environment σ'_{b} that agrees with σ_{b} on $\mathsf{defBefore}(x_1, \Gamma)$.

Using Lemmas 43, 47, 44, we can take a step on the source side and thus change $(\sigma_\mathsf{b}, x_1\ x_2\ k)$ to $(\sigma'_\mathsf{b}[f, y, k' \mapsto \sigma_\mathsf{b}(x_1), \sigma_\mathsf{b}(x_2), \sigma_\mathsf{b}(k)], e)$ in the goal.

Now we make use of the original (stronger) $\precsim$ relation: we apply Lemma 69 to $\mathsf{defBefore}(x_1, \Gamma)^\dagger, f, y, k' \vdash e$, yielding

$$\mathsf{defBefore}(x_1, \Gamma)^\dagger, f, y, k' \vdash \phi(e) \precsim^\phi_w e$$

for any renaming ϕ.

Note that this immediately leads to the goal if we pick $\phi := [x_1/f][x_2/y][k/k']$ and then show the environments related modulo ϕ. That is, it remains to show the relatedness of $\sigma_\mathsf{a}(\phi(z))$ and $(\sigma'_\mathsf{b}[f, y, k' \mapsto \sigma_\mathsf{b}(x_1), \sigma_\mathsf{b}(x_2), \sigma_\mathsf{b}(k)])(z)$ for any $z \in \mathsf{defBefore}(x_1, \Gamma)^\dagger, f, y, k'$.

For $z = k'$, this reduces to the relatedness of $\sigma_\mathsf{a}(k)$ and $\sigma_\mathsf{b}(k)$, which holds due to the assumed relatedness of σ_a and σ_b for Γ. Similarly for $z = y$ and $z = f$. For z in $\mathsf{defBefore}(x_1, \Gamma)^\dagger$, we must show $\sigma_\mathsf{a}(z)$ related to $\sigma'_\mathsf{b}(z)$. Fortunately we know from $\mathsf{respects}(\sigma_\mathsf{b}, \psi, \Gamma)$ that $\sigma'_\mathsf{b}(z) = \sigma_\mathsf{b}(z)$ for such z, hence we can use the same argument here as well and are done. □

The correctness statements for the purely structural cases (including *ainline*) are simpler, and easy to obtain with the help of compatibility lemmas for the generalized $\precsim$ relations. Unfortunately, these lemmas are currently not generalizations of those for the original $\preceq$ relations (it wasn't obvious how to achieve this due to technical difficulties related to variables). Nevertheless, they look fairly similar and are omitted here. Suffice it to say that for simplicity none of them modifies the ψ_a and ψ_b components. For the purpose of verifying inlining of *local* functions, though, it will most likely be useful to state the lemma for let-bindings of functions such that in the premise the mappings are extended with the newly defined function.

Combining these correctness lemmas with a simple induction leads to the following familiar-looking lemma (omitting some technical side conditions concerning variable freshness and uniqueness of e and the functions inside ψ).

Lemma 75.

$$\frac{\Gamma \vdash e \qquad \mathsf{stable}(w) \qquad \ldots}{\Gamma; \epsilon; \psi \vdash \mathit{einline}(\psi)(e) \precsim^*_w e}$$

Based on this, we can now verify the module-level transformation *inline*. Naturally this is the point where we must demonstrate that the runtime environments that arise from loading a module actually respect the module, so that we can discharge the respective assumptions hidden inside the conclusion of Lemma 75. All this is not very hard but tedious. In the end, we arrive at the following correctness property.

Lemma 76 (Correctness of Inlining)**.**

$$\frac{\Gamma \vdash M : \Gamma' \quad \mathit{uniqmod}(M)}{\Gamma \vdash \mathit{inline}(M) : \Gamma' \quad \Gamma \vdash \mathit{inline}(M) \precsim^*_{\mathcal{II}} M : \Gamma'}$$

Note that we cannot prove $\mathit{uniqmod}(M')$ because it usually doesn't hold—inlining can obviously destroy the uniquess property. For this reason, the inlining pass is immediately followed by a very simple *freshening* pass whose sole purpose is to restore this property.

5.8.3 Freshening

The freshening pass renames all bound variables of a module in order to (re-)establish the variable uniqueness property. It is run after inlining and, as we will see in the next section, after contification as well. As shown in Figure 5.10, freshening is a very simple transformation and easy to verify thanks to $\precsim$'s support of renaming, hence we keep its presentation very short.

The functions *efreshen* and *afreshen* take a renaming (mapping from variables to variables) and an index indicating the next unused variable. The result is the transformed expression and an updated index. *freshen* lifts these functions to the module level, passing through the current index, which is initially 0.

Regarding semantic correctness, the key lemma is the following (and similarly for *afreshen*. This follows easily with the help of the compatibility lemmas from Section 5.3.

$$\frac{\Gamma \vdash e \quad (e', n') = \mathit{efreshen}(\phi)(n)(e) \quad (\forall z \in \Gamma.\ \phi(z) < n) \quad \mathsf{stable}(w)}{\Gamma \vdash e' \precsim_w e}$$

On top of that, we can derive *freshen*'s semantics preservation. The overall correctness statement includes preservation of well-formedness and—the reason for defining freshening in the first place—enforcement of the uniqueness property that other passes rely on.

$$\frac{\Gamma \vdash M : \Gamma' \quad M' = \mathit{freshen}(M)}{\Gamma \vdash M' : \Gamma' \quad \mathit{uniqmod}(M') \quad \Gamma \vdash M' \precsim^*_{\mathcal{II}} M : \Gamma'}$$

5.9 The Contify Pass

Pilsner includes a (very simplistic) contification pass. Contification [24] is an optimization that turns a function into a continuation when the function is only ever called with the same continuation argument. This makes control flow more explicit, thus potentially enabling subsequent optimizations. In Pilsner, it has the additional benefit that continuations don't need to be heap-allocated.

5.9.1 Transformation

Contification in Pilsner uses a slightly generalized version of the expression transformation framework from Section 5.3.2, *i.e.,* it first runs an untrusted analysis that

$$
\begin{array}{l}
\mathit{freshen} \in \mathbf{Mod} \to \mathbf{Mod} \\
\mathit{freshen}(M) = \mathit{mfreshen}(0)(M) \\
\\
\mathit{mfreshen} \in \mathbb{N} \to \mathbf{Mod} \to \mathbf{Mod} \\
\mathit{mfreshen}(n)(\epsilon) = \epsilon \\
\mathit{mfreshen}(n)((F, a), M) = \\
\quad \mathit{let}\ (a', n') := \mathit{afreshen}(\mathrm{id})(n)\ \mathit{in} \\
\quad (F, a'), \mathit{mfreshen}(n')(M) \\
\\
\mathit{afreshen} \in (\mathsf{Var} \to \mathsf{Var}) \to \mathbb{N} \to \mathsf{BExp} \to \mathsf{BExp} \times \mathbb{N} \\
\mathit{afreshen}(\phi)(n)(\Lambda k.\, e) = \\
\quad \mathit{let}\ (e', n') := \mathit{efreshen}(\phi[\mathrm{kvar}(n)/k])(n+1)(e)\ \mathit{in} \\
\quad (\Lambda\, \mathrm{kvar}(n).\, e', n') \\
\ldots \\
\\
\mathit{efreshen} \in (\mathsf{Var} \to \mathsf{Var}) \to \mathbb{N} \to \mathsf{Exp} \to \mathsf{Exp} \times \mathbb{N} \\
\mathit{efreshen}(\phi)(n)(y \leftarrow !x;\ e) = \\
\quad \mathit{let}\ (e', n') := \mathit{efreshen}(\phi[\mathrm{tvar}(n)/y])(n+1)(e)\ \mathit{in} \\
\quad (\mathrm{tvar}(n) \leftarrow !\phi(x);\ e', n') \\
\ldots
\end{array}
$$

Figure 5.10: Freshening.

annotates places in the module where the expression-level contification should happen. This saves a lot of work because only the expression-level contification needs to be verified.

Expression-level contification consists of two steps. Consider a non-recursive function binding

$$\mathsf{let}\ y = (\mathsf{fix}\ \_(y_1, k).\, e')\ \mathsf{in}\ e$$

that is contifiable, *i.e.,* where all invocations of y in e are done using the same continuation k'. We first extend the binding with the definition of a new continuation, namely the contification of y (w.r.t. to k'):

$$
\begin{array}{l}
\mathsf{let}\ y = (\mathsf{fix}\ \_(y_1, k).\, e')\ \mathsf{in} \\
\mathsf{let}\ k_y = \mathsf{cont}\ y_2.\, e'[y_2/y_1][k'/k]e\ \mathsf{in}
\end{array}
$$

In the second step, all invocations of y in e are turned into calls of the newly added continuation k_y (*e.g.,* $y\ x\ k' \rightsquigarrow k_y\ x$). Note that contification does not purge the definition of y, but leaves this to the dead code elimination pass. We exclude recursive functions from contification because our language does not support recursive continuations.

Our analysis is fairly simple. Whenever it finds a function binding that is contifiable (according to the conservative syntactic criterion above), it annotates the

binding as follows:

$$\alpha(\mathsf{let}\ y = (\mathsf{fix}\, f(y', k).\, e')\ \mathsf{in}\ e)\ :=\ [\mathsf{Subst}_{k'}, \mathsf{Add}_{k'}]\mathsf{let}\ y = ([\epsilon]\mathsf{fix}\, f(y', k).\, \alpha(e'))\ \mathsf{in}\ \alpha(e) \quad (f \notin \mathrm{fv}(e') \wedge \mathsf{contifiable}(y, e) = k')$$

Here, the annotation $\mathsf{Add}_{k'}$ corresponds to the operation of adding a new contification of the function with fixed continuation argument k' (step 1), and $\mathsf{Subst}_{k'}$ corresponds to the operation of replacing all calls of y that use k' with corresponding continuation calls (step 2). By having k' be part of both annotations, the implementations of the respective operations do not need to re-inspect the code in order to find out k'. The analysis uses the list lifting of annotations from Section 5.3.2, which performs transformations from right to left.

Non-contifiable functions as well as other expression forms receive an empty list annotation (if annotatable):

$$\begin{aligned}
&\alpha(\mathsf{let}\ y = (\mathsf{fix}\, f(y', k).\, e')\ \mathsf{in}\ e) :=\\
&\quad [\epsilon]\mathsf{let}\ y = ([\epsilon]\mathsf{fix}\, f(y', k).\, \alpha(e'))\ \mathsf{in}\ \alpha(e) \qquad (f \in \mathrm{fv}(e') \vee \mathsf{contifiable}(y, e) = \bot)\\
&\alpha(y \leftarrow !x;\ e) :=\\
&\quad y \leftarrow !x;\ \alpha(e)\\
&\ldots
\end{aligned}$$

If our analysis (or a different one) is incorrect, then either the expression-level contification will fail, or it will succeed but the later dead code elimination won't be able to remove the original function binding. For instance, the first situation would arise if the analysis were to mark as contifiable anything other than function bindings, and the secound situation would arise if the analysis were to mark as contifiable a function that is actually called with different continuation arguments. Neither situation can would result in a violation of semantics preservation.

A note on variable uniqueness. Contification, as presented above, basically duplicates existing code (the function body). Clearly this does not preserve the uniqueness property of variables, which was painfully established by CPS and which later compiler passes rely on. The obvious solution to this issue lies in freshening the new copy of the function body with the help of *efreshen* from Section 5.8.3, in the implementation of the Add_k transformer. This immediately raises two further issues, though: (i) We can't tell which variables *efreshen* is allowed to use (*i.e.,* which n to pass), and (ii) the transformation framework's concrete syntactic correctness criterion actually forbids the introduction of new bound variables (Section 5.3.2.2).

For this reason, Pilsner's contification is based on a slight generalization of the transformation framework from Section 5.3.2. In this version, an expression transformation additionally takes the index n of the next unused variable as an argument (which, for instance, it can pass on to *efreshen*). Moreover, the restriction on bound variables is relaxed: a transformation may introduce new bound variables, as long as their numeric values lie between the given n and some n' that the transformation

must output (of course each of them must be bound only once). In this way, the framework can provide an appropriate n for each transformation and ensure that the newly introduced variables are disjoint from any existing ones and from any that get introduced by other transformations.

5.9.2 Verification

The first transformation, the transformer for Add_k is very easy to prove correct since all it does is introduce an unused binding (the reverse of dead code elimination). Recall the expressions from the beginning of this section. All we have to do is basically apply the compatibility lemmas for let and fix, take a step on the target side (binding the continuation k_y), stutter on the source side, and finally apply reflexivity of e (Lemma 70).

The proof of the second transformation, the transformer for Subst_k, mainly boils down to a lemma relating the new continuation calls to the old function calls, which looks roughly as follows:

$$\frac{\psi_{\mathsf{a}}(k_y) = \mathsf{cont}\ y_2.\, e[y_2/y_1][k'/k] \qquad \psi_{\mathsf{b}}(y) = \mathsf{fix}\, f(y_1, k).\, e}{\Gamma; \psi_{\mathsf{a}}; \psi_{\mathsf{b}} \vdash k_y\ x \precsim_w^* y\ x\ k'}$$

Here we make crucial use of the "context-sensitive" version of $\precsim$ that we introduced in Section 5.8.2. It lets us express the connection between the two programs's environments via the static descriptions ψ_{a} and ψ_{b}: the code of k_y in the transformed program must be precisely the contification of the code of y in the original program (glossing over the aforementioned freshening). Without such an assumption, the connection would be lost and there would be no hope of proving the calls related.

5.10 The Codegen Pass

Code generation, Pilsner's final pass, translates $\mathcal{I}$ code to the machine language $\mathcal{T}$.

5.10.1 Transformation

5.10.1.1 Compiling Variable Lookups

We first explain how variables are looked up at run-time. Recall that there are three kinds of "variables" in $\mathcal{I}$: labels F, term variables y, and continuation variables k. For each we define a helper function that generates the code for lookups, as shown in Figure 5.11. In either case, the lookup is based on the variable's position in the well-formedness context Γ of the expression being compiled (we do not store any variable names in memory). Hence these functions (and thus code generation itself) all take Γ as an input. Let us look at the details. (In an attempt to improve readability, we will often omit parentheses around arguments to meta-level functions here.)

$$
\begin{array}{l}
\mathit{lookup}_{\mathsf{Lbl}} \in \mathsf{Lbl} \to \mathsf{Word} \to [\mathsf{Var}] \to \mathsf{Reg} \to [\mathsf{Instr}] \\
\mathit{lookup}_{\mathsf{Lbl}}\, F\, n\, \Gamma\, r := \\
\quad \mathit{let}\ i := \mathit{index}\, F\, (\Gamma|_{\mathsf{Lbl}})\ \mathit{in} \\
\quad \mathsf{lpc}\ r, \\
\quad \mathsf{bop}\ (-)\ r\ r\ n, \\
\quad \mathsf{ld}\ r\ [r + i] \\
\\
\mathit{lookup}_{\mathrm{TVar}} \in \mathrm{TVar} \to [\mathsf{Var}] \to \mathsf{Reg} \to [\mathsf{Instr}] \\
\mathit{lookup}_{\mathrm{TVar}}\, y\, \Gamma\, r := \\
\quad \mathit{let}\ i := \mathit{index}\, y\, (\Gamma|_{\mathrm{TVar}})\ \mathit{in} \\
\quad \mathsf{ld}\ r\ \mathsf{env}, \\
\quad \mathit{repeat}\, i\, (\mathsf{ld}\ r\ [r + 1]), \\
\quad \mathsf{ld}\ r\ [r + 0] \\
\\
\mathit{lookup}_{\mathrm{KVar}} \in \mathrm{KVar} \to [\mathsf{Var}] \to [\mathsf{Instr}] \\
\mathit{lookup}_{\mathrm{KVar}}\, k\, \Gamma := \\
\quad \mathit{let}\ i := 2 * \mathit{index}\, k\, (\Gamma|_{\mathrm{KVar}}) + 2\ \mathit{in} \\
\quad \mathsf{bop}\ (-)\ \mathsf{sp}\ r\ i, \\
\quad \mathsf{ld}\ \mathsf{ret}\ \langle \mathsf{sp} + 0 \rangle, \\
\quad \mathsf{ld}\ \mathsf{env}\ \langle \mathsf{sp} + 1 \rangle
\end{array}
$$

Figure 5.11: Code generated for variable lookups.

Labels are looked up in the static label environment that the loader places as part of the group header right before the data block in the heap (recall the memory layout from Figure 4.10 and the goal of position-independent code). In order for this to work, the various code generating functions keep track of the label environment's distance n relative to the code being produced.

lookup$_{\mathsf{Lbl}}$ emits the code for label lookups. Each lookup consists of three instructions. It first gets its own code address (in the heap) and—using the distance n—calculates the absolute address of the label environment. Then it indexes the environment according to F's position[4] in Γ (restricted to labels) and loads the correct value into the given register r.

Term variables are looked up in a singly-linked list in the heap, which is immutable except that it can grow. Code emitted for expressions expects the env register to point to this list.

The lookup code, generated by *lookup*$_{\mathrm{TVar}}$, is straightforward: it skips the first i elements in the list and then loads the next one into r, where i is the position of y in Γ.

Continuation variables are looked up on the stack. Each continuation takes up two slots: the first points to the continuation's code and the second to its term variable environment (both in the heap). Code emitted for expressions expects that the topmost stack cells correspond to the continuation variables in Γ. (Pilsner code does not use the stack for anything else, but imported functions that were not compiled by Pilsner may of course store arbitrary data there. For instance, Zwickel compiled code uses it to store intermediate values.)

lookup$_{\mathrm{KVar}}$ calculates k's position in the stack and then loads its code address into register ret and its environment address into register env (we will see later why it makes sense to always use these registers). Subsequently, it pops the continuation and all above it (more recently defined ones) from the stack by decrementing the stack pointer sp. This makes the lookup a destructive operation; it is safe because $\mathcal{I}$'s affinity property (cf. Section 4.2.4) ensures that using k now means that k won't be needed later and neither do any continuations that got defined after k (in the same lexical scope).

5.10.1.2 Compiling Expressions

Figures 5.12 and 5.13 show *acodegen* and *ecodegen*, which are responsible for translating basic expressions $a \in \mathsf{BExp}$ and control expressions $e \in \mathsf{Exp}$, respectively. Like above, the argument n is the size of the code emitted so far.

In the case of basic expressions only, the generated code writes the result value into the heap, at the address indicated by the arg register. This is always the last thing it does, so the execution will run into whatever instructions follow. On the

[4]The *index* function counts from 0.

other hand, code for control expressions will always jump to another function or continuation. Let us take a closer look at some representative examples.

Sums. The code generated for inl x first uses *lookup* to load the value of x into register ret (which is used as a temporary register here). It then allocates two heap cells, one for the tag of the sum (0 indicates "in left"), and one for the value of x. The code ends after writing the sum value itself, represented by the address of the first cell, into the result location (see above). Note that x may be either a label or a term variable. Here, and below, we write $lookup_{\mathsf{Lbl} \cup \mathrm{TVar}}$ for the obvious combination of $lookup_{\mathsf{Lbl}}$ and $lookup_{\mathrm{TVar}}$.

Let bindings. The code emitted for let $y = a$ in e (by *ecodegen*) is equally straightforward. It allocates a new list node (two heap cells) and sets its "next" pointer (the second cell) to the current environment. Then it executes c_1, the code for a. Since it does so when arg points to the new node, a's result will automatically end up in the right place, namely in the node's first cell. At this point the node represents the extension of the environment list with the value of a. The code sets env to this extended list and executes the code for e.

Note that we must be careful to provide the correct offsets when using *acodegen* and *ecodegen* to generate code for the subexpressions. For instance, we pass $3 + n$ to *acodegen* because c_1 will be preceded by three instructions.

Function calls. When compiling a function call, we want to produce the same code no matter what kind of function we are dealing with (imported from an unknown module vs. exported by the current module vs. defined locally in the current module). Hence when we compile a function, we always make it follow the regular calling convention (introduced in Section 4.2.5). This makes it very simple to define *ecodegen* for function calls. The code generated for $x_2\ x_1\ k$ first looks up the value of the argument x_1 and stores it into register arg. Next, it looks up the value of the function x_2 and stores it into register clo. Finally, it looks up the continuation variable k and performs the prescribed indirect jump to the function.

Recall that looking up a continuation has the effect of writing the continuation's code address into ret and its environment into env. This fits very well, because a function expects its return address in ret. Moreoever, since env is a callee-save register, the continuation can be sure to find its environment intact when it eventually gets control.

Continuation calls. The code emitted by *ecodegen* for continuation calls $k\ x$ loads the argument value into register arg, loads the continuation and performs a direct jump to its code.

Functions. As far as the calling convention is concerned, a function value is just an address on which to perform an indirect jump, *i.e.,* the address of a heap cell that

$$
\begin{array}{l}
\mathit{acodegen} \in \mathsf{BExp} \to \mathsf{Word} \to [\mathsf{Var}] \to [\mathsf{Instr}] \\
\\
\mathit{acodegen}\,(\mathsf{inl}\ x)\,n\,\Gamma := \\
\quad \mathit{lookup}_{\mathsf{Lbl} \cup \mathrm{TVar}}\,x\,n\,\Gamma\ \mathsf{ret}, \\
\quad \mathsf{ld}\ \mathsf{aux}_1\ 2, \\
\quad \mathsf{alloc}\ \mathsf{clo}\ \mathsf{aux}_1, \\
\quad \mathsf{ld}\ \mathsf{aux}_1\ 0, \\
\quad \mathsf{sto}\ [\mathsf{clo} + 0]\ \mathsf{aux}_1, \\
\quad \mathsf{sto}\ [\mathsf{clo} + 1]\ \mathsf{ret}, \\
\quad \mathsf{sto}\ [\mathsf{arg} + 0]\ \mathsf{clo} \\
\\
\mathit{acodegen}\,(\mathsf{fix}\,f\,(y, k).\,e)\,n\,\Gamma := \\
\quad \mathit{let}\ c := \mathit{ecodegen}\,e\,(22 + n)\,(\Gamma, f, y, k)\ \mathit{in} \\
\quad \left.\begin{array}{l}
\mathsf{ld}\ \mathsf{aux}_1\ 2, \\
\mathsf{alloc}\ \mathsf{clo}\ \mathsf{aux}_1, \\
\mathsf{lpc}\ \mathsf{aux}_1, \\
\mathsf{bop}\ (+)\ \mathsf{aux}_1\ \mathsf{aux}_1\ 7, \\
\mathsf{sto}\ [\mathsf{clo} + 0]\ \mathsf{aux}_1, \\
\mathsf{sto}\ [\mathsf{clo} + 1]\ \mathsf{env}, \\
\mathsf{sto}\ [\mathsf{arg} + 0]\ \mathsf{clo}, \\
\mathsf{bop}\ (+)\ \mathsf{clo}\ \mathsf{aux}_1\ (13 + \mathit{length}\,c), \\
\mathsf{jmp}\ \mathsf{clo},
\end{array}\right\}\ \text{active part} \\
\quad \left.\begin{array}{l}
\mathsf{sto}\ \langle \mathsf{sp} + 0 \rangle\ \mathsf{ret}, \\
\mathsf{sto}\ \langle \mathsf{sp} + 1 \rangle\ \mathsf{env}, \\
\mathsf{bop}\ (+)\ \mathsf{sp}\ \mathsf{sp}\ 2, \\
\mathsf{ld}\ \mathsf{aux}_1\ 2, \\
\mathsf{alloc}\ \mathsf{env}\ \mathsf{aux}_1, \\
\mathsf{sto}\ [\mathsf{env} + 0]\ \mathsf{clo}, \\
\mathsf{ld}\ \mathsf{aux}_1\ [\mathsf{clo} + 1], \\
\mathsf{sto}\ [\mathsf{env} + 1]\ \mathsf{aux}_1, \\
\mathsf{ld}\ \mathsf{aux}_1\ 2, \\
\mathsf{alloc}\ \mathsf{clo}\ \mathsf{aux}_1, \\
\mathsf{sto}\ [\mathsf{clo} + 0]\ \mathsf{arg}, \\
\mathsf{sto}\ [\mathsf{clo} + 1]\ \mathsf{env}, \\
\mathsf{ld}\ \mathsf{env}\ \mathsf{clo}, \\
c
\end{array}\right\}\ \text{inactive part}
\end{array}
$$

Figure 5.12: Code generation for basic expressions.

$$
\begin{array}{l}
\mathit{ecodegen} \in \mathsf{Exp} \to \mathsf{Word} \to [\mathsf{Var}] \to [\mathsf{Instr}] \\
\\
\mathit{ecodegen}\,(\mathsf{let}\ y = a\ \mathsf{in}\ e)\,n\,\Gamma := \\
\quad \mathit{let}\ c_1 := \mathit{acodegen}\,a\,(3+n)\,\Gamma)\ \mathit{in} \\
\quad \mathit{let}\ c_2 := \mathit{ecodegen}\,e\,(4 + \mathsf{length}\,c_1 + n)\,(\Gamma, y)\ \mathit{in} \\
\quad \mathsf{ld}\ \mathsf{aux}_1\ 2, \\
\quad \mathsf{alloc}\ \mathsf{arg}\ \mathsf{aux}_1, \\
\quad \mathsf{sto}\ [\mathsf{arg}+1]\ \mathsf{env}, \\
\quad c_1, \\
\quad \mathsf{ld}\ \mathsf{env}\ \mathsf{arg}, \\
\quad c_2 \\
\\
\mathit{ecodegen}\,(\mathsf{let}\ k = \mathsf{cont}\ y.\,e_2\ \mathsf{in}\ e_1)\,n\,\Gamma := \\
\quad \mathit{let}\ c_1 := \mathit{ecodegen}\,e_1\,(5+n)\,(\Gamma, k)\ \mathit{in} \\
\quad \mathit{let}\ c_2 := \mathit{ecodegen}\,e_2\,(10 + \mathsf{length}\,c_1 + n)\,(\Gamma, y)\ \mathit{in} \\
\quad \mathsf{lpc}\ \mathsf{aux}_1, \\
\quad \mathsf{bop}\ (+)\ \mathsf{aux}_1\ \mathsf{aux}_1\ (5 + \mathsf{length}\,c_1), \\
\quad \mathsf{sto}\ \langle \mathsf{sp}+0 \rangle\ \mathsf{aux}_1, \\
\quad \mathsf{sto}\ \langle \mathsf{sp}+1 \rangle\ \mathsf{env}, \\
\quad \mathsf{bop}\ (+)\ \mathsf{sp}\ \mathsf{sp}\ 2, \\
\quad c_1, \\
\quad \mathsf{ld}\ \mathsf{clo}\ \mathsf{env}, \\
\quad \mathsf{ld}\ \mathsf{aux}_1\ 2, \\
\quad \mathsf{alloc}\ \mathsf{env}\ \mathsf{aux}_1, \\
\quad \mathsf{sto}\ [\mathsf{env}+0]\ \mathsf{arg}, \\
\quad \mathsf{sto}\ [\mathsf{env}+1]\ \mathsf{clo}, \\
\quad c_2 \\
\\
\mathit{ecodegen}\,(x_2\ x_1\ k)\,n\,\Gamma := \\
\quad \mathit{let}\ c_1 := \mathit{lookup}_{\mathsf{Lbl} \cup \mathrm{TVar}}\,x_1\,n\,\Gamma\,\mathsf{arg}\ \mathit{in} \\
\quad \mathit{let}\ c_2 := \mathit{lookup}_{\mathsf{Lbl} \cup \mathrm{TVar}}\,x_2\,(\mathit{length}\,c_1 + n)\,\Gamma\,\mathsf{clo}\ \mathit{in} \\
\quad \mathit{let}\ c_3 := \mathit{lookup}_{\mathrm{KVar}}\,k\,\Gamma\ \mathit{in} \\
\quad c_1, c_2, c_3, \\
\quad \mathsf{jmp}\ [\mathsf{clo}+0] \\
\\
\mathit{ecodegen}\,(k\ x)\,n\,\Gamma := \\
\quad \mathit{let}\ c_1 := \mathit{lookup}_{\mathsf{Lbl} \cup \mathrm{TVar}}\,x\,n\,\Gamma\,\mathsf{arg}\ \mathit{in} \\
\quad \mathit{let}\ c_2 := \mathit{lookup}_{\mathrm{KVar}}\,k\,\Gamma\ \mathit{in} \\
\quad c_1, c_2, \\
\quad \mathsf{jmp}\ \mathsf{ret}
\end{array}
$$

Figure 5.13: Code generation for control expressions.

contains the address of the function's first instruction. For top-level functions, which are closed (modulo labels), this is all we need. In fact, the loader already creates these values as part of the group header, so besides producing the function code, we just need to fill these values with the correct code pointers.

For local functions, there's more to do: they need to be converted to proper closures, *i.e.,* pairs of code pointer and term variable environment, for which we must dynamically allocate heap storage. We represent closures as two consecutive cells, where the first one holds the code pointer and the second one the environment pointer. The function value itself then is the address of the first cell. This still matches the calling convention: an indirect jump on the function value will reach the first instruction of its code.

So let us look at the code emitted for local functions $\mathsf{fix}\, f(y,k).\, e$ by *acodegen*. It consists of an active and an inactive part. The active part, executed immediately, creates and returns the function closure; the inactive part is the code that the closure references, *i.e.,* that gets executed when the function is called.

- The active part starts off by allocating two cells of heap storage for the closure. It then computes the absolute address of the beginning of the inactive part and writes it into the first cell. Next, it writes the current environment pointer into the second cell (recall that the values in this environment are read-only). Having created and populated the closure, it stores the function value, *i.e.,* the address of the first closure cell, at the designated result location in the heap (see above). Finally, it computes the address of the last instruction in the inactive part and jumps one behind it, *i.e.,* it skips over the inactive part and jumps to whatever code follows.

- The inactive part consists of a prologue and of c, the code for the function body e. The prologue first pushes the contents of registers ret and env on the stack. Since the stack acts as the continuation environment, pushing ret and env (in this order) means extending the continuation environment with an entry for the function's continuation argument k. If the function gets called from Pilsner-compiled code, then this makes perfect sense as ret will be the continuation's code address and env its term environment (see the compilation of function calls above). But even when the function gets called from arbitrary code, this makes perfect sense: the calling convention requires that the function returns to the address in ret and preserves the contents of env. Treating ret plus env as a Pilsner continuation will have exactly this effect.

  After pushing k on the stack, the code sets up the term variable environment for e by extending the function's own environment with entries for f and y. Concretely, it first creates a list node for f in which it writes the function value itself (stored in clo as per the calling convention) to it. It attaches this node to the function's existing term environment, which we know resides in $[\mathsf{clo}+1]$ (the second cell of the closure). Next, it creates a list node for y, writes the contents of arg to it, attaches it to the previous node, and writes its address

$$
\begin{array}{l}
\mathit{codegen} \in \mathbf{Mod}_{\mathcal{I}} \to [\mathsf{Lbl}] \to \mathbf{Mod}_{\mathcal{T}} \\
\mathit{codegen} M\,\Gamma := (\mathit{mcodegen}\,M\,(\mathit{length}\,\Gamma + 2 * \mathit{length}\,M)\,0\,\Gamma) \\
\\
\mathit{mcodegen} \in \mathbf{Mod}_{\mathcal{I}} \to \mathsf{Word} \to \mathsf{Word} \to [\mathsf{Lbl}] \to \mathbf{Mod}_{\mathcal{T}} \\
\mathit{mcodegen}\,\epsilon\,m\,n\,\Gamma := (\epsilon, \epsilon) \\
\mathit{mcodegen}\,(F{=}a, M)\,m\,n\,\Gamma := \\
\quad \mathit{let}\ c := \mathit{tcodegen}\,a\,(m+n)\,\Gamma\ \mathit{in} \\
\quad \mathit{let}\ g := \mathit{mcodegen}\,m\,(n + \mathit{length}\,c)\,(\Gamma, F)\ \mathit{in} \\
\quad ((F, n), g.\mathit{cptrs}, (\mathit{map}\,\mathbb{E}\,c, g.\mathit{data})) \\
\\
\mathit{tcodegen}\,(\mathsf{fix}\,f(y,k).\,e)\,n\,\Gamma := \\
\quad \mathit{let}\ c := \mathit{ecodegen}\,e\,(10+n)\,(\Gamma, f, y, k)\ \mathit{in} \\
\quad \mathsf{sto}\ \langle \mathsf{sp} + 0 \rangle\ \mathsf{ret}, \\
\quad \mathsf{sto}\ \langle \mathsf{sp} + 1 \rangle\ \mathsf{env}, \\
\quad \mathsf{bop}\ (+)\ \mathsf{sp}\ \mathsf{sp}\ 2, \\
\quad \mathsf{ld}\ \mathsf{aux}_1\ 2, \\
\quad \mathsf{alloc}\ \mathsf{aux}_1\ \mathsf{aux}_1, \\
\quad \mathsf{sto}\ [\mathsf{aux}_1 + 0]\ \mathsf{clo}, \\
\quad \mathsf{ld}\ \mathsf{env}\ 2, \\
\quad \mathsf{alloc}\ \mathsf{env}\ \mathsf{env}, \\
\quad \mathsf{sto}\ [\mathsf{env} + 0]\ \mathsf{arg}, \\
\quad \mathsf{sto}\ [\mathsf{env} + 1]\ \mathsf{aux}_1, \\
\quad c
\end{array}
$$

Figure 5.14: Code generation for modules.

into env. This concludes the function prologue and c is executed next.

Continuation bindings. As the final example, consider continuation bindings $\mathsf{let}\ k = \mathsf{cont}\ y.\,e_2\ \mathsf{in}\ e_1$. The code emitted by *ecodegen* first gets its own address so that it can calculate the absolute address of the continuation code (starting with the first load instruction). It pushes this address onto the stack, together with the current environment pointer, thus extending the continuation environment with an entry for k. Next, the code for the binding's body e_1 is executed. If and when this eventually leads to the continuation being invoked (directly or indirectly), the continuation can be sure to find its term environment and input value in env and arg, respectively. It then extends its environment with the input value in order for e_2's code, which it executes next, to find it there.

5.10.1.3 Compiling Modules

Code generation for modules is shown in Figure 5.14. The final transformation is *codegen*, which takes an $\mathcal{I}$ module with a matching wellformedness context Γ (*i.e.,*

the list of imports), and produces a $\mathcal{T}$ module. This module consists of a single group (recall the structure of machine modules from Section 4.2.5), which is generated by the helper function *mcodegen*. For each function definition $F{=}a$ in M, *mcodegen* adds an entry for F to the group's code pointer table and adds the code produced for a to the group's data block (after encoding the instructions as machine words). In order to create the correct code pointer table entries, *mcodegen* maintains the size n of the code generated so far (for the functions "to the left"). Initially this is 0, so the first function will be mapped to 0 as this is the offset in the data block where its code starts. When using *tcodegen* to compile the functions, it also passes the correct offset along, which is always the current n plus the fixed size m of the group header.

The helper *tcodegen* translates top-level functions, and top-level functions only (Figure 5.14 shows the fix case but omits the analogous Λ case). As discussed previously, top-level functions are somewhat simpler than local functions. The code generated by *tcodegen* corresponds to the inactive part that *acodegen* would generate, except that the prologue that sets up the term environment doesn't need to load and extend any existing environment.

5.10.2 Verification

Code generation is the most radical and low-level transformation in Pilsner, and so it comes as no surprise that its proof is also the longest and most tedious. Here we can only give an overview.

The main goal is to show the following theorem, which states the semantics preservation of *codegen* in terms of our canonical $\precsim_{\mathcal{TI}}$ relation.

Theorem 21.

$$\frac{\Gamma \vdash M : \Gamma'}{\Gamma \vdash \mathit{codegen}\, M\, \Gamma \precsim_{\mathcal{TI}} M : \Gamma'}$$

Analogous to earlier proofs, a key part of this is finding a relation $\preccurlyeq_{\mathcal{TI}}$ that formalizes when some machine code implements an open $\mathcal{I}$ expressions. For this relation we must prove something along the lines of

$$\frac{\Gamma \vdash e}{\Gamma \vdash \mathit{ecodegen}\, e \preccurlyeq_{\mathcal{TI}} e}$$

(and similarly for *acodegen*).

In $\preccurlyeq_{\mathcal{TI}}$, we want to ultimately say that the two programs are similar according to **E**. But clearly the generated code makes many assumptions about its environment, *e.g.,* where term variables can be looked up, how continuations are laid out on the stack, where temporary results are placed, etc. In order to restrict the environments in which the code is placed, we must therefore express parts of the compiler-internal protocol that the code follows. Naturally, the choice of the local world in Theorem 21 plays an important role in this.

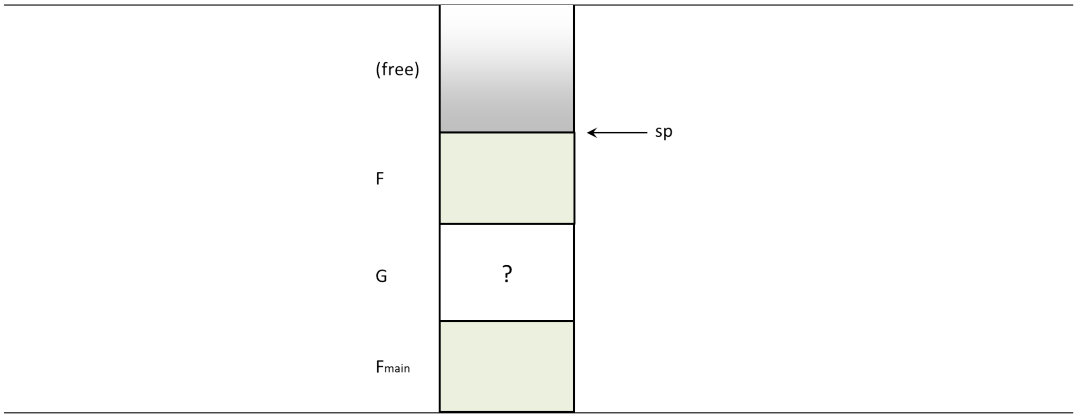


Figure 5.15: Example of Pilsner's local stack.

5.10.2.1 The Local World

We define a local world w for the verification of code generation such that a state s consists of a stack (fragment) and a heap (fragment). They represent the memory used *internally* by the machine code generated for module M. The configuration relation $w.\mathsf{C}$ ensures that $\mathcal{T}$'s local heap is precisely the one given by the state, and similarly for the stack (so from now on we use the terms local and internal interchangeably).

Let us review the internal memory that Pilsner-generated code maintains.

Stack. The local stack $s.q$ consists of the current continuation environment. Note that this may not be one consecutive chunk due to calls to imported functions that haven't returned yet.

To illustrate a bit better what the local stack typically looks like, let's assume that the module for which we generate code provides the "main" function F_{main}. And that F_{main} calls an imported higher-order function G, passing as argument another function F from our module. After G calls F, the full machine stack looks as shown in Figure 5.15. The parts that constitute the local stack $s.q$ are shown in green.

The first chunk is part of our local stack and contains the continuations that F_{main}'s code allocated before it called G. The second chunk is not part of our local stack, but part of the frame configuration that the **E** relation quantifies over. It contains whatever G had pushed before it called F. We have no idea what that is (if anything at all) because G may not have been compiled by Pilsner. The third chunk is again part of our local stack. It contains the continuations that F's code just allocated. The final chunk of the stack is free and infinite. It is owned by the global world, as discussed in Section 4.4.2.3.

Heap. As is standard, the local heap $s.h$ contains the label environment. Of interest is what else it consists of: all the module's code and the singly linked lists representing

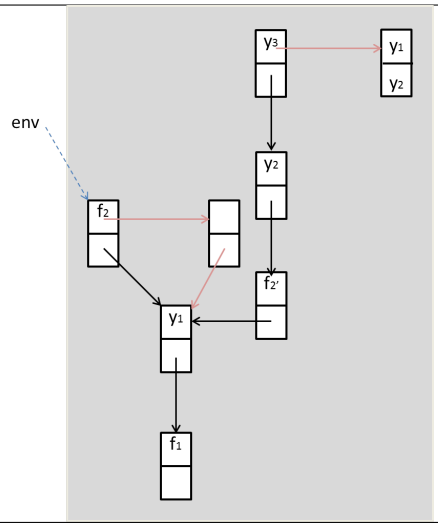


Figure 5.16: Example of Pilsner's local heap.

term variable environments that have been allocated so far (typically, many of them share nodes).

For instance, consider a module consisting of a single function F:

$$\begin{aligned} F = {} & \mathsf{fix}\, f_1(y_1, k_1). \\ & \quad \mathsf{let}\ f_2 = (\mathsf{fix}\, f_2'(y_2, k_2).\, \mathsf{let}\ y_3 = \langle y_1, y_2 \rangle\ \mathsf{in}\ k_2\ y_3)\ \mathsf{in} \\ & \quad k_1\ f_2 \end{aligned}$$

Right before the execution of the main function, the local heap consists solely of the label environment and the code for F (which, of course, includes the code for the local function f_2). After executing the main function's prologue, the local heap additionally contains a term environment that provides the values for f_1 and y_1 (pointed to by register env). Executing the let-binding has two effects: (i) it allocates a new closure and (ii) it extends the environment with a new list node containing the closure (the value for f_2). Step (i) affects the local heap only partially: the second cell of the closure, which points to the initial term environment, is part of the local heap. However, the first cell, which contains the code pointer and represents the value of the function, is part of the global heap instead. This is enforced by our global world $\Omega_{\mathcal{TI}}$: if we want f_2 to be recognized as a proper function (*e.g.,* so that we can pass it to the unknown k_1) then we must register it as such in the global state's value registry. This registry, as we have seen in Section 4.4.2.3, is tied to the global heap.

The next time the local heap changes is when k_1 calls f_2 (assuming such a call is made, and assuming k_1 does not happen to call F itself before). The prologue of f_2 extends f_2's environment with nodes for f_2' and y_2. Then the let-binding (i) allocates a new pair and (ii) further extends the environment with a node for y_3. Step (i) does not affect the local heap at all, because the representation of a pair as the address of the first of two heap cells is nothing internal but rather part the official "interface" of

values that every module must agree with. $\Omega_{\mathcal{TI}}$ enforces via the value registry that both cells of the pair are exposed.

As a summary, Figure 5.16 depicts the shape of the local heap when execution reaches the call to k_2 inside of f_2. Besides the group environment and the code, it contains f_2's closure term environment, the cell of f_2's closure that points to this environment, and an extension of that environment matching the current execution of f_2. The black arrows denote the "next" pointers of the singly-linked environments.

The formal definition of w is quite simple:

Definition 50 (Local world w for the verification of code generation)**.**

$$
\begin{array}{lcl}
w \in \mathrm{LWorld}^{\Omega_{\mathcal{IS}}.\mathsf{T}} & & \\
w.\mathsf{T}.\mathsf{S} & := & \mathsf{Stack} \times \mathsf{Heap} \\
w.\mathsf{T}.\sqsupseteq & := & \{\, ((q', h'), (q, h)) \mid h' \supseteq h \,\} \\
w.\mathsf{T}.\sqsupseteq_{\mathrm{pub}} & := & \{\, ((q', h'), (q, h)) \mid h' \supseteq h \wedge q' = q \,\} \\
w.\mathsf{C}(G)(s_{\mathrm{g}}, (q, h)) & := & \{\, (c_{\mathsf{a}}, c_{\mathsf{b}}) \mid c_{\mathsf{a}} = (\emptyset, \emptyset, q, h) \wedge c_{\mathsf{b}} = (\emptyset, \emptyset) \,\} \\
w.\mathsf{O} & := & \mathrm{gwf}
\end{array}
$$

Note that a state does not record any information about the $\mathcal{I}$ program's memory. $\mathcal{I}$'s heap is only used for references, which we can all treat as global in this verification. Hence the $\mathcal{I}$ configurations in $w.\mathsf{C}$ are empty.

Both transition relations allow the local heap to grow but not to be modified otherwise, which matches the use in Pilsner code. Regarding the local stack, the full transition relation allows arbitrary changes while the public one allows no changes at all. This reflects the calling convention: a function may modify the stack arbitrarily as long as it restores it again before returning.

We pick w's initial state analogous to the example proof in Section 4.6. In particular, we choose its local heap to be exactly the local heap given by **cload**. This heap consists of two continuous chunks: (i) the label environment of size $\mathit{length}\,\Gamma + \mathit{length}\,\Gamma'$, starting at whatever load address $\Psi_{\mathcal{T}}$ we are given, and (ii) the code, starting at address $\Psi_{\mathcal{T}} + \mathit{length}\,\Gamma + 2 * \mathit{length}\,\Gamma'$. The initial local stack is empty.

5.10.2.2 Relating Machine Code to Open Expressions

We now define the $\preccurlyeq_{\mathcal{TI}}$ relation, which is of the form $\Gamma \vdash f \preccurlyeq_{\mathcal{TI}} e$ and $\Gamma \vdash f \preccurlyeq_{\mathcal{TI}} a$. But what is f? It doesn't make sense to directly relate machine code (instruction sequences) to control expressions or basic expressions, because the code that Pilsner generates depends on its distance n to the label environment. We model this by relating a meta-level function $f \in \mathsf{Word} \to [\mathsf{Word}]$ instead, which maps n to (encoded) instructions. In practice, f will always be $\boldsymbol{\lambda} n.\ \mathit{map}\,\mathbb{E}\,(\mathit{ecodegen}\,e\,n\,\Gamma)$ and $\boldsymbol{\lambda} n.\ \mathit{map}\,\mathbb{E}\,(\mathit{acodegen}\,a\,n\,\Gamma)$, respectively.

In the case of control expressions e, $\preccurlyeq_{\mathcal{TI}}$ is defined as follows.

Definition 51.

$$\begin{aligned}
\boxed{\Gamma \vdash f \preccurlyeq_{\mathcal{TI}} e} := \forall i.\ \exists i'.\ \forall G, s_0, s, K_\mathsf{a}, K_\mathsf{b}, \psi_\mathsf{a}, \sigma_\mathsf{a}, \sigma_\mathsf{b}, m, \mathit{cur}, \mathit{old}.\\
\quad G \in \mathbf{GK}_{w\uparrow} \wedge\\
\quad s \sqsupseteq s_0 \wedge\\
\quad m \neq 0 \wedge\\
\quad (\exists n.\ s.h \supseteq [m \longmapsto f\, n] \sqcup [m - n \longmapsto \mathit{map}\ \mathit{snd}\ \psi_\mathsf{a}]) \wedge\\
\quad (\forall F \in \Gamma.\ \exists v_\mathsf{a}, v_\mathsf{b}.\ \psi_\mathsf{a}(F) = v_\mathsf{a} \wedge \sigma_\mathsf{b}(F) = v_\mathsf{b} \wedge (v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s)) \wedge\\
\quad \Gamma|_{\mathsf{Lbl}} = \mathit{map}\ \mathit{fst}\ \psi_\mathsf{a} \wedge\\
\quad \mathit{repr}\ s.h\ (s.R(\mathsf{env}))\ (\mathit{map}\ \mathit{snd}\ \sigma_\mathsf{a}) \wedge\\
\quad (\forall y \in \Gamma.\ \exists v_\mathsf{a}, v_\mathsf{b}.\ \sigma_\mathsf{a}(y) = v_\mathsf{a} \wedge \sigma_\mathsf{b}(y) = v_\mathsf{b} \wedge (v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s)) \wedge\\
\quad \Gamma|_{\mathrm{TVar}} = \mathit{map}\ \mathit{fst}\ \sigma_\mathsf{a} \wedge\\
\quad s.q = [s_0.R(\mathsf{sp}) \longmapsto \mathit{cur}] \sqcup \mathit{old} \wedge\\
\quad s.R(\mathsf{sp}) = s_0.R(\mathsf{sp}) + \mathit{length}\ \mathit{cur} \wedge\\
\quad \mathit{length}\ \mathit{cur} = 2 * \mathit{length}\ \Gamma|_{\mathrm{KVar}} \wedge\\
\quad \mathsf{K}\, G\, s_0\, s\, K_\mathsf{a}\, K_\mathsf{b}\, i\, \sigma_\mathsf{b}\, \Gamma\\
\implies ((m, \emptyset, \emptyset, \emptyset), (\emptyset, (\sigma_\mathsf{b}, e))) \in \mathbf{E}_{w\uparrow}(i' + i)(K_\mathsf{a}, K_\mathsf{b})(G)(s_0)(s)(\bot)
\end{aligned}$$

Definition 52.

$$\boxed{\mathit{repr}\ h\ v\ d} \qquad \frac{}{\mathit{repr}\ h\ v\ \epsilon} \qquad \frac{h(v) = w \quad h(v+1) = v' \quad \mathit{repr}\ h\ v'\ W}{\mathit{repr}\ h\ v\ (w, W)}$$

The conclusion of the implication states that a $\mathcal{T}$ configuration consisting of a program counter m is $\mathbf{E}$-related in state s of $w\uparrow$ to an $\mathcal{I}$ configuration consisting of e and some closing environment σ_b. Let us walk through the assumptions under which this must hold.

The first larger condition connects the program counter m to f: it says that the current local heap contains the sequence of words given by $f\, n$ (think: the code), starting at address m. The distance n is arbitrary but the condition also tells us that the local heap contains some data starting at address $m - n$.

The next condition gives meaning to this data: it is actually the label environment. For any label F in Γ, the data chunk contains a value that is related to whatever value F has in e's environment σ_b. The subsequent condition says that the values in the data chunk are ordered according to Γ. In effect, these conditions tell us that looking up a label will yield the expected result (see also Lemma 78 below).

The next three conditions concern term variables. With the help of the predicate from Definition 52, the first of these states that the current value of register env (exposed via the global state) points to a singly-linked list in the local heap containing certain elements (think: the term environment). The subsequent condition tells us something about these elements: for each term variable y in Γ, there is an element that is related to whatever value y has in e's environment σ_b. And yet another condition says that these elements are ordered according to Γ. In effect, these three conditions tell us that looking up a term variable will yield the expected result (see also Lemma 78 below).

The remaining four conditions concern continuation variables. The first says that the local stack can be split into a "current" and an "old" part, and that the current part starts where the stack ended in state s_0. Next, that the current part is actually the top part of the machine stack right now. The last two conditions intuitively say that the current part is the continuation environment matching Γ (recall that each continuation there takes up two slots) and that the stored continuations are related to the corresponding ones in e's environment σ_{b} (see also Lemma 78 below).

This last condition is itself somewhat complex, as it must in turn describe the assumptions that code generated for continuations makes:

Definition 53.

$$
\begin{aligned}
&\boxed{\mathsf{K}\, G\, s_0\, s\, K_{\mathsf{a}}\, K_{\mathsf{b}}\, i\, \sigma_{\mathsf{b}}\, \Gamma} := \forall k \in \Gamma.\ \forall j, \mathbf{k}_{\mathsf{a}}, w, \mathbf{k}_{\mathsf{b}}.\ j = \mathit{index}\, k\, \Gamma|_{\mathrm{KVar}} \implies \\
&\quad s.q(s.R(\mathsf{sp}) - 2 * (1 + j)) = \mathbf{k}_{\mathsf{a}} \wedge \\
&\quad s.q(s.R(\mathsf{sp}) - 2 * (1 + j) + 1) = w \wedge \\
&\quad \sigma_{\mathsf{b}}(k) = \mathbf{k}_{\mathsf{b}} \wedge \\
&\quad (\forall G', s', v_{\mathsf{a}}, v_{\mathsf{b}}. \\
&\qquad G' \supseteq G \wedge s' \sqsupseteq s \wedge s'.R(\mathsf{sp}) = s.R(\mathsf{sp}) - 2 * (1 + j) \wedge \\
&\qquad (\exists p \in [\mathsf{Word}].\ \mathit{length}\, p = 2 * (1 + j) \wedge s.q = s'.q \sqcup [s'.R(\mathsf{sp}) \longmapsto p]) \wedge \\
&\qquad s'.R(\mathsf{env}) = w \wedge (v_{\mathsf{a}}, v_{\mathsf{b}}) \in \overline{G'}(s') \\
&\qquad \implies \\
&\qquad w{\uparrow}.\mathsf{cqh}_{\mathsf{a}}(s')(\mathsf{ret}\, v_{\mathsf{a}}\, \mathbf{k}_{\mathsf{a}}) \times w{\uparrow}.\mathsf{cqh}_{\mathsf{b}}(s')(\mathsf{ret}\, v_{\mathsf{b}}\, \mathbf{k}_{\mathsf{b}}) \subseteq \\
&\qquad \mathbf{E}_{w\uparrow}(i)(K_{\mathsf{a}}, K_{\mathsf{b}})(G')(s_0)(s')(\bot))
\end{aligned}
$$

For each continuation variable k in Γ, it expresses the following: The first three conditions state that, based on k's position[5] j in Γ, there are two adjacent slots associated with it in the local stack. The first contains $\mathbf{k}_{\mathsf{a}}$ (think: code pointer) and the second contains w (think: environment pointer). (k having position j means that there are j continuations above it on the stack). The next condition says that k has value $\mathbf{k}_{\mathsf{b}}$ in e's environment σ_{b}.

The last condition then makes the connection between the $\mathcal{T}$ continuation and the $\mathcal{I}$ continuation for k. Roughly, when they are used in some future state s' with related input values v_{a}, v_{b}, then they must yield **E**-related computations as long as the following preconditions are met: k itself and all continuations that were above it in the stack have been popped (p represents the popped part) and apart from that, the stack is unchanged; furthermore, k's environment must have been loaded into register env. It is easy to see that the code generated by *ecodegen* for continuation calls establishes these conditions.

Let us now consider the definition of $\preccurlyeq_{\mathcal{TI}}$ for basic expressions $a \in \mathsf{BExp}$. Its purpose is to capture the contract that *acodegen*-generated code adheres to.

[5] A variable that occurs multiple times is only considered once, as *index* returns the index of the only relevant occurrence.

Definition 54.

$$
\begin{aligned}
&\boxed{\Gamma \vdash f \preccurlyeq_{\mathcal{TI}} a} := \forall i, G, s_0, s, K_\mathsf{a}, K_\mathsf{b}, \psi_\mathsf{a}, \sigma_\mathsf{a}, \sigma_\mathsf{b}, m, n, v, i', x, e, c_\mathsf{a}, c_\mathsf{b}, \eta_\mathsf{a}, \eta_\mathsf{b}.\\
&\qquad G \in \mathbf{GK}_{w\uparrow} \wedge\\
&\qquad s \sqsupseteq s_0 \wedge\\
&\qquad m \neq 0 \wedge\\
&\qquad (\exists h.\ s.h = [m \longmapsto f\ n] \sqcup [m - n \longmapsto \mathit{map}\ \mathit{snd}\ \psi_\mathsf{a}] \sqcup h) \wedge\\
&\qquad (\forall F \in \Gamma.\ \exists v_\mathsf{a}, v_\mathsf{b}.\ \psi_\mathsf{a}(F) = v_\mathsf{a} \wedge \sigma_\mathsf{b}(F) = v_\mathsf{b} \wedge (v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s)) \wedge\\
&\qquad \Gamma|_{\mathsf{Lbl}} = \mathit{map}\ \mathit{fst}\ \psi_\mathsf{a} \wedge\\
&\qquad \mathit{repr}\ s.h\ (s.R(\mathsf{env}))\ (\mathit{map}\ \mathit{snd}\ \sigma_\mathsf{a}) \wedge\\
&\qquad (\forall y \in \Gamma.\ \exists v_\mathsf{a}, v_\mathsf{b}.\ \sigma_\mathsf{a}(y) = v_\mathsf{a} \wedge \sigma_\mathsf{b}(y) = v_\mathsf{b} \wedge (v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s)) \wedge\\
&\qquad \Gamma|_{\mathrm{TVar}} = \mathit{map}\ \mathit{fst}\ \sigma_\mathsf{a} \wedge\\
&\qquad (\forall s' \sqsupseteq_{\mathrm{pub}} s.\ \forall v_\mathsf{a}, v_\mathsf{b}.\\
&\qquad\quad s'.R(\mathsf{arg}) = s.R(\mathsf{arg}) \wedge\\
&\qquad\quad s'.h(s'.R(\mathsf{arg})) = v_\mathsf{a} \wedge\\
&\qquad\quad (v_\mathsf{a}, v_\mathsf{b}) \in \overline{G}(s')\\
&\qquad\quad \Longrightarrow\ ((m + \mathit{length}(f\ n), \emptyset, \emptyset, \emptyset),\\
&\qquad\qquad\quad (\emptyset, (\sigma_\mathsf{b}[x \mapsto v_\mathsf{b}], e))) \in \mathbf{E}_{w\uparrow}(i')(K_\mathsf{a}, K_\mathsf{b})(G)(s_0)(s')(\bot)) \wedge\\
&\qquad (c_\mathsf{a}, c_\mathsf{b}) \in w\uparrow.\mathsf{C}(G)(s)\\
&\Longrightarrow\ ((m, \emptyset, \emptyset, [s.R(\mathsf{arg}) \mapsto v]) \cdot c_\mathsf{a} \cdot \eta_\mathsf{a},\\
&\qquad (\emptyset, (\sigma_\mathsf{b}, \mathsf{let}\ x = a\ \mathsf{in}\ e)) \cdot c_\mathsf{b} \cdot \eta_\mathsf{b}) \in \mathbf{E}_{w\uparrow}(i)(K_\mathsf{a}, K_\mathsf{b})(G)(s_0)(s)(\eta_\mathsf{a}, \eta_\mathsf{b})
\end{aligned}
$$

Several parts here look very much the same as for control expressions. Let us highlight the important differences.

1. Since we cannot directly relate a basic expression using **E** (in the implication's conclusion), we wrap a with a let-binding. This does not restrict the applicability, because—excluding top-level functions—a let-binding is the only way a basic expression can be used. Top-level functions are not compiled using *acodegen*, so we do not need to consider them here.

2. Instead of using **E** in the default mode, we use it in fixed mode (cf. Section 4.5.2) and then explicitly assume that parts of the configurations are world-related (namely c_a and c_b) and other parts are the frames (η_a and η_b). This would be equivalent to just using the default mode and hiding these configurations if it weren't for $[s.R(\mathsf{arg}) \mapsto v]$. Here is the deal: We must assert ownership of the heap cell initially pointed to by register arg, because this is where the code is going to write its result (cf. Section 5.10.1.2). However, we cannot consider this cell part of the local heap before the result has actually been written, because w allows no mutation of the local heap. Concretely, if we were to state that $s.h$ contains $[s.R(\mathsf{arg}) \mapsto v]$, where v is the arbitrary and irrelevant value currently stored in that cell, then we would get stuck when reasoning about the final write of the result: there would be no future state matching the final heap.

   As an alternative to this definition's use of the fixed mode, we could probably have used a more complicated local world that allows for some cells of the

internal heap to initially be in an "uncommitted" state, where their contents can still change. The current approach seems simpler, though.

3. The last clause in the implication's premise connects the two big unknowns: the code following f's code in memory (which the execution of f's code will eventually run into), and e, the expression that we had to introduce as part of the let-binding.

   It says that they must be **E**-related in any public future state s' in which the result cell is part of the local heap and contains a value v_{a} related to the one that x is mapped to in e's environment. Note that now that the result cell is actually in the local heap, we can use **E** in its default mode again.

   The reason for requiring s' to be a *public* extension is that the local stack cannot have changed (and it is important to know that). In fact, only registers and the value of the result cell can have changed, which is why there is no need to consider a future global knowledge. This is in contrast to control expressions, whose evaluation can change global references and call external functions.

   The reason for requiring that the contents of register arg hasn't changed is merely that the code of let-bindings relies on this (this could easily be changed, though).

4. Nowhere do we say anything about continuation variables. This is fine, because basic expressions cannot refer to any continuations coming from an outer lexical scope (cf. Section 4.2.4.2).

5.10.2.3 Key Lemma

We are now able to formulate the desired key lemma.

Lemma 77.

$$\frac{\Gamma \vdash e}{\Gamma \vdash (\boldsymbol{\lambda} n.\ \mathit{map}\,\mathbb{E}\,(\mathit{ecodegen}\,e\,n\,\Gamma)) \preccurlyeq_{\mathcal{TI}} e}$$

Proof. Reminiscent of compatibility lemmas (cf. Section 3.7.3), we prove one lemma for each form of control expression and basic expression. For instance, the statement for functions—as usual, the only one proven using coinduction—reads as follows:

$$\frac{\Gamma^\dagger, f, y, k \vdash (\boldsymbol{\lambda} n.\ \mathit{map}\,\mathbb{E}\,(\mathit{ecodegen}\,e\,n\,(\Gamma^\dagger, f, y, k))) \preccurlyeq_{\mathcal{TI}} e}{\Gamma \vdash (\boldsymbol{\lambda} n.\ \mathit{map}\,\mathbb{E}\,(\mathit{acodegen}\,(\mathsf{fix}\,f(y,k).\,e)\,n\,\Gamma)) \preccurlyeq_{\mathcal{TI}} (\mathsf{fix}\,f(y,k).\,e)}$$

As another example, here is the statement for applications:

$$\frac{x_1 \in \Gamma \quad x_2 \in \Gamma \quad k \in \Gamma}{\Gamma \vdash (\boldsymbol{\lambda} n.\ \mathit{map}\,\mathbb{E}\,(\mathit{ecodegen}\,(x_1\ x_2\ k)\,n\,\Gamma)) \preccurlyeq_{\mathcal{TI}} (x_1\ x_2\ k)}$$

With an easy induction, these properties then yield the goal. □

The various sub-proofs of Lemma 77 employ Lemmas 78 and 79. These provide Hoare-style reasoning principles for the variable lookup code generated by the *lookup* functions that we saw earlier. In particular the first one is tremendously useful, as code generation makes use of $lookup_{\mathsf{Lbl}\cup\mathrm{TVar}}$ in over twenty places.

Lemma 78 (Label or term variable lookup)**.** If

1. $[pc - n \longmapsto map\ snd\ \psi_{\mathsf{a}}] \subseteq h\ \wedge\ \Gamma|_{\mathsf{Lbl}} = map\ fst\ \psi_{\mathsf{a}}$
2. $repr\ h\ (R(\mathsf{env}))\ (map\ snd\ \sigma_{\mathsf{a}})\ \wedge\ \Gamma|_{\mathrm{TVar}} = map\ fst\ \sigma_{\mathsf{a}}$
3. $[pc \longmapsto map\ \mathbb{E}\ c] \subseteq h\ \wedge\ c = lookup_{\mathsf{Lbl}\cup\mathrm{TVar}}\ x\ n\ \Gamma\ r\ \wedge\ pc \neq 0$
4. $\psi_{\mathsf{a}}(x) = v\ \vee\ \sigma_{\mathsf{a}}(x) = v$
5. $pc' = pc + length\ c\ \wedge\ R' = R[r \mapsto v]\ \wedge\ i' <^{length\ c} i$
6. $((pc', R', q, h), c_{\mathsf{b}}) \in \mathbf{E}_{w\uparrow}(i')(K_{\mathsf{a}}, K_{\mathsf{b}})(G)(s_0)(s)(\eta_{\mathsf{a}}, \eta_{\mathsf{b}})$

then $((pc, R, q, h), c_{\mathsf{b}}) \in \mathbf{E}_{w\uparrow}(i)(K_{\mathsf{a}}, K_{\mathsf{b}})(G)(s_0)(s)(\eta_{\mathsf{a}}, \eta_{\mathsf{b}})$.

The lemma describes the precise effects of executing the *lookup* code and can be read as follows: When reasoning about a $\mathcal{T}$ machine whose program counter points to the first instruction of that code, it suffices to reason about the machine *after* executing that code, which is obtained by incrementing the program counter accordingly and by setting the r register to whatever the requested value is. (The requested value is determined by condition (4) depending on whether x is a label or a term variable.) Note how conditions (1) and (2) match the definition of $\preccurlyeq_{\mathcal{TI}}$.

Lemma 79 is analogous:

Lemma 79 (Continuation variable lookup)**.** If

1. $[pc \longmapsto map\ \mathbb{E}\ c] \subseteq h\ \wedge\ c = lookup_{\mathrm{KVar}}\ k\ \Gamma\ \wedge\ pc \neq 0$
2. $j = index\ k\ \Gamma|_{\mathrm{KVar}}\ \wedge\ R(\mathsf{sp}) = n + 2 * (1 + j)\ \wedge\ q(n) = v\ \wedge\ q(n+1) = w$
3. $pc' = pc + length\ c\ \wedge\ R' = R[\mathsf{sp} \mapsto n][\mathsf{ret} \mapsto v][\mathsf{env} \mapsto w]\ \wedge\ i' <^{length\ c} i$
4. $((pc', R', q, h), c_{\mathsf{b}}) \in \mathbf{E}_{w\uparrow}(i')(K_{\mathsf{a}}, K_{\mathsf{b}})(G)(s_0)(s)(\eta_{\mathsf{a}}, \eta_{\mathsf{b}})$

then $((pc, R, q, h), c_{\mathsf{b}}) \in \mathbf{E}_{w\uparrow}(i)(K_{\mathsf{a}}, K_{\mathsf{b}})(G)(s_0)(s)(\eta_{\mathsf{a}}, \eta_{\mathsf{b}})$.

5.11 The Full Pilsner Compiler

At this point, we have verified each of Pilsner's passes separately. Let us summarize the results:

- The first pass:

$$\frac{\Gamma \vdash M : \Gamma'}{\Gamma \vdash cps\ M \precsim_{\mathcal{IS}} M : \Gamma' \quad |\Gamma| \vdash cps\ M : |\Gamma'| \quad uniqmod(cps\ M)}$$

- The intermediate passes:

$$\frac{\Gamma \vdash M : \Gamma' \quad \mathit{uniqmod}(M)}{\Gamma \vdash f\ M \precsim^*_{\mathcal{II}} M : \Gamma' \quad \Gamma \vdash f\ M : \Gamma' \quad \mathit{uniqmod}(f\ M)}$$

  for $f \in \{\, \mathit{inline}, \mathit{contify}, \mathit{dce}, \mathit{hoist}, \mathit{commute}, \mathit{dedup} \,\}$

- The final pass:

$$\frac{\Gamma \vdash M : \Gamma' \quad \mathit{uniqmod}(M)}{\Gamma \vdash \mathit{codegen}\ M\ \Gamma \precsim_{\mathcal{TI}} M : \Gamma' \quad \vdash \mathit{codegen}\ M\ \Gamma : \Gamma'}$$

Thanks to transitivity of PILS (Theorem 15), the composition of all these transformations is correct w.r.t. $\precsim_{\mathcal{TS}}$.

Note that it doesn't actually matter which of the intermediate transformations are applied. So we can easily allow the user of Pilsner to selectively disable some optimizations[6]. Formally, we define Pilsner as follows. It takes a sequence of Boolean flags, each enabling or disabling one intermediate transformation.

Definition 55 (Pilsner with selection)**.**

$$\begin{aligned}
&\mathit{pilsner} \in \mathbb{B}^6 \to [\mathsf{Lbl}] \to \mathbf{Mod} \to \mathbf{Mod} \\
&\mathit{pilsner}\ \mathit{flags}\ \Gamma := \\
&\quad (\boldsymbol{\lambda} M.\ \mathit{codegen}\ M\ \Gamma) \circ \\
&\quad (\mathit{if}\ \mathit{flags}.6\ \mathit{then}\ \mathit{dedup}\ \mathit{else}\ \mathrm{id}) \circ \\
&\quad (\mathit{if}\ \mathit{flags}.5\ \mathit{then}\ \mathit{commute}\ \mathit{else}\ \mathrm{id}) \circ \\
&\quad (\mathit{if}\ \mathit{flags}.4\ \mathit{then}\ \mathit{hoist}\ \mathit{else}\ \mathrm{id}) \circ \\
&\quad (\mathit{if}\ \mathit{flags}.3\ \mathit{then}\ \mathit{dce}\ \mathit{else}\ \mathrm{id}) \circ \\
&\quad (\mathit{if}\ \mathit{flags}.2\ \mathit{then}\ \mathit{contify}\ \mathit{else}\ \mathrm{id}) \circ \\
&\quad (\mathit{if}\ \mathit{flags}.1\ \mathit{then}\ \mathit{inline}\ \mathit{else}\ \mathrm{id}) \circ \\
&\quad \mathit{cps}
\end{aligned}$$

No matter which flags the user chooses, Pilsner is always correct:

Theorem 22 (Correctness of Pilsner)**.**

$$\frac{\Gamma \vdash M : \Gamma'}{\Gamma \vdash \mathit{pilsner}\ \mathit{flags}\ |\Gamma|\ M \precsim_{\mathcal{TS}} M : \Gamma' \qquad \vdash \mathit{pilsner}\ \mathit{flags}\ |\Gamma|\ M : |\Gamma'|}$$

5.12 The Zwickel Compiler

The Zwickel compiler is a straightforward compiler from $\mathcal{S}$ directly down to $\mathcal{T}$, not involving any intermediate transformations. As such, it is rather different from Pilsner, and being able to verify it using the PILS system developed in Chapter 4 provides some evidence of the flexibility of our approach.

[6] We could even let the user choose the order in which optimizations are performed, but that may not be very useful as optimizations are typically written with a certain order in mind.

```
lookup_Lbl ∈ Lbl → Word → [Var] → [Instr]
lookup_Lbl F n Γ :=
    let i := index F (Γ|_Lbl) in
    lpc aux1,
    bop (−) aux1 aux1 n,
    ld aux1 [aux1 + i],
    bop (+) sp sp 1,
    sto ⟨sp − 1⟩ aux1

lookup_TVar ∈ TVar → [Var] → [Instr]
lookup_TVar y Γ :=
    let i := index y (Γ|_TVar) in
    sto clo env
    repeat i (ld clo [clo + 1]),
    bop (+) sp sp 1,
    ld clo [clo + 0],
    sto ⟨sp − 1⟩ clo
```

Figure 5.17: Code generated for variable lookups.

5.12.1 Transformation

Despite the big difference to Pilsner as a whole, Zwickel is somewhat similar to Pilsner's *code generation* pass (Section 5.10). We can therefore keep this section relatively short by focussing on the differences.

Recall that Pilsner's codegen compiles $\mathcal{I}$ programs, not $\mathcal{S}$ programs. The code that it generates for a pure expression writes the result value into a new environment slot in the heap (because pure expressions can only occur in let-bindings, which extend the environment); control expressions, on the other hand, do not directly produce a value because they are in CPS. In $\mathcal{S}$, there is no distinction between pure and control expressions, and every expression produces a value (if it terminates). For Zwickel, we thus follow the convention that the code for an expression e pushes e's value on the stack. We could have used a fixed register instead, but using the stack makes Zwickel more different from Pilsner, where the stack is reserved for $\mathcal{I}$'s continuations instead.

Figures 5.17–5.19 below roughly correspond to Figures 5.11–5.13. Like Pilsner's codegen, the Zwickel code for looking up labels F—shown in Figure 5.17—reads from the loader environment whose start address is known relative to the current instruction. Similarly, term variables x are looked up in the linked-list environment pointed to by register env. Since labels and term variables in $\mathcal{S}$ are syntactically expressions, however, the found value is in both cases then pushed on the stack. (This is why the *lookup* functions don't take a register argument, as they did in Pilsner.) Also, recall that there are no continuation variables in $\mathcal{S}$.

$$
\begin{array}{l}
\mathit{ezwickel} \in \mathsf{Exp} \to \mathsf{Word} \to [\mathsf{Var}] \to [\mathsf{Instr}] \\
\\
\mathit{ezwickel}\, x\, n\, \Gamma := \\
\quad \mathit{lookup}_{\mathrm{TVar}}\, x\, n\, \Gamma \\
\\
\mathit{ezwickel}\, F\, n\, \Gamma := \\
\quad \mathit{lookup}_{\mathsf{Lbl}}\, F\, n\, \Gamma \\
\\
\mathit{ezwickel}\, (\mathsf{inl}\ e)\, n\, \Gamma := \\
\quad \mathit{ezwickel}\, e\, n\, \Gamma, \\
\quad \mathsf{ld\ clo\ 2}, \\
\quad \mathsf{alloc\ clo\ clo}, \\
\quad \mathsf{ld\ aux_1\ 0}, \\
\quad \mathsf{sto\ [clo + 0]\ aux_1}, \\
\quad \mathsf{ld\ aux_1\ \langle sp - 1 \rangle}, \\
\quad \mathsf{sto\ [clo + 1]\ aux_1}, \\
\quad \mathsf{sto\ \langle sp - 1 \rangle\ clo}
\end{array}
$$

Figure 5.18: Code generation for expressions.

Figures 5.18 and 5.19 show excerpts of Zwickel's expression compilation *ezwickel*, which takes the same arguments as *ecodegen* and *acodegen* did in Pilsner. Let us here just focus on compiling functions and function calls. Functions now have an epilogue (the last four instructions) that moves the body's value, found on the top of the stack, into the function call's result register arg, as required by the calling convention that all modules adhere to. The epilogue also restores the callee-save register env that the function stored on the stack as well. The code for an application does not simply pass through a given return address, but provides its own such that it can push the result value back onto the stack (an application is an expression). Of course it must then also restore the original return address in ret, which it saved on the stack.

The module-level compilation $\mathit{zwickel} \in \mathbf{Mod}_{\mathcal{S}} \to [\mathsf{Lbl}] \to \mathbf{Mod}_{\mathcal{T}}$ is defined analogous to that in Pilsner and thus not shown here.

5.12.2 Verification

The verification of Zwickel follows roughly that of Pilsner's codegen (Section 5.10). In fact, we use essentially the same local world w.

$ezwickel\,(\mathsf{fix}\,f(x).\,e)\,n\,\Gamma :=$
$\quad let\ c := ezwickel\,e\,(22+n)\,(\Gamma, f, x)\ in$
$\quad \mathsf{ld\ clo}\ 2,$
$\quad \mathsf{alloc\ clo\ clo},$
$\quad \mathsf{lpc\ arg},$
$\quad \mathsf{bop}\ (+)\ \mathsf{arg\ arg}\ 8,$
$\quad \mathsf{sto}\ [\mathsf{clo}+0]\ \mathsf{arg},$
$\quad \mathsf{sto}\ [\mathsf{clo}+1]\ \mathsf{env},$
$\quad \mathsf{bop}\ (+)\ \mathsf{sp\ sp}\ 1,$
$\quad \mathsf{sto}\ \langle \mathsf{sp}-1\rangle\ \mathsf{clo},$
$\quad \mathsf{bop}\ (+)\ \mathsf{arg\ arg}\ (16 + length\,c),$
$\quad \mathsf{jmp\ arg},$
} active part

$\quad \mathsf{bop}\ (+)\ \mathsf{sp\ sp}\ 1,$
$\quad \mathsf{sto}\ \langle \mathsf{sp}-1\rangle\ \mathsf{env},$
$\quad \mathsf{ld\ env}\ 2,$
$\quad \mathsf{alloc\ env\ env},$
$\quad \mathsf{sto}\ [\mathsf{env}+0]\ \mathsf{clo},$
$\quad \mathsf{ld}\ \mathsf{aux}_1\ [\mathsf{clo}+1],$
$\quad \mathsf{sto}\ [\mathsf{env}+1]\ \mathsf{aux}_1,$
$\quad \mathsf{ld\ clo}\ 2,$
$\quad \mathsf{alloc\ clo\ clo},$
$\quad \mathsf{sto}\ [\mathsf{clo}+0]\ \mathsf{arg},$
$\quad \mathsf{sto}\ [\mathsf{clo}+1]\ \mathsf{env},$
$\quad \mathsf{ld\ env\ clo},$
$\quad c,$
$\quad \mathsf{ld\ arg}\ \langle \mathsf{sp}-1\rangle,$
$\quad \mathsf{bop}\ (-)\ \mathsf{sp\ sp}\ 2,$
$\quad \mathsf{ld\ env}\ \langle \mathsf{sp}+0\rangle,$
$\quad \mathsf{jmp\ ret}$
} inactive part

$ezwickel\,(e_1\ e_2)\,n\,\Gamma :=$
$\quad let\ c_1 := ezwickel\,e_1\,n\,\Gamma\ in$
$\quad let\ c_2 := ezwickel\,e_2\,(length\,c_1 + n)\ \Gamma\ in$
$\quad c_1, c_2,$
$\quad \mathsf{bop}\ (-)\ \mathsf{sp\ sp}\ 1,$
$\quad \mathsf{ld\ clo}\ \langle \mathsf{sp}-1\rangle,$
$\quad \mathsf{sto}\ \langle \mathsf{sp}-1\rangle\ \mathsf{ret},$
$\quad \mathsf{ld\ arg}\ \langle \mathsf{sp}+0\rangle,$
$\quad \mathsf{lpc\ ret},$
$\quad \mathsf{bop}\ (+)\ \mathsf{ret\ ret}\ 3,$
$\quad \mathsf{jmp}\ [\mathsf{clo}+0],$
$\quad \mathsf{ld\ ret}\ \langle \mathsf{sp}-1\rangle,$
$\quad \mathsf{sto}\ \langle \mathsf{sp}-1\rangle\ \mathsf{arg}$

Figure 5.19: Code generation for expressions (continued).

Definition 56 (Local world w for the verification of Zwickel)**.**

$$
\begin{array}{lcl}
w \in \mathrm{LWorld}^{\Omega_{\mathcal{TS}}.\mathsf{T}} & & \\
w.\mathsf{T}.\mathsf{S} & := & \mathsf{Stack} \times \mathsf{Heap} \\
w.\mathsf{T}.\sqsupseteq & := & \{\, ((q',h'),(q,h)) \mid h' \supseteq h \,\} \\
w.\mathsf{T}.\sqsupseteq_{\mathrm{pub}} & := & \{\, ((q',h'),(q,h)) \mid h' \supseteq h \wedge q' = q \,\} \\
w.\mathsf{C}(G)(s_{\mathrm{g}},(q,h)) & := & \{\, (c_{\mathsf{a}},c_{\mathsf{b}}) \mid c_{\mathsf{a}} = (\emptyset,\emptyset,q,h) \wedge c_{\mathsf{b}} = (\emptyset,\emptyset,\emptyset) \,\} \\
w.\mathsf{N}.\mathsf{NS} & := & \emptyset \\
w.\mathsf{N}.\mathsf{NR} & := & \emptyset \\
w.\mathsf{O} & := & \mathrm{gwf}
\end{array}
$$

The only differences are due to this being a local world for $\precsim_{\mathcal{TS}}$ rather than for $\precsim_{\mathcal{TI}}$. In particular, since we are now working in the typed version of PILS, we must provide a set of type names $w.\mathsf{N}.\mathsf{NS}$ and a relational interpretation $w.\mathsf{N}.\mathsf{NR}$ for them—we choose the empty set. Otherwise, the structure of w is the same as before, although the contents of the stack $s.q$ will look rather different in our Zwickel proofs than they did for Pilsner.

Analogous to Definition 51 in Section 5.10, we define an auxiliary relation $\preccurlyeq_{\mathcal{TS}}$ between machine code parameterized over its load address (f) and its source expression (e), relative to a type and typing context. Similar to before, m is the address at which we load the code $f\,m$, and the only f in which we are interested is $\boldsymbol{\lambda} n.\ \mathit{map}\,\mathbb{E}\,(\mathit{ezwickel}\,e\,n\,|\Gamma|)$.

Definition 57.

$$\begin{aligned}
&\boxed{\Gamma \vdash f \preccurlyeq_{\mathcal{TS}} e : \tau} := \exists i.\ \forall G, s_0, s, \delta, \psi_{\mathsf{a}}, \sigma_{\mathsf{a}}, \sigma_{\mathsf{b}}, m, n, j, \tau'.\\
&\qquad G \in \mathbf{GK}_{w\uparrow} \wedge\\
&\qquad s \sqsupseteq s_0 \wedge\\
&\qquad m \neq 0 \wedge\\
&\qquad (\exists h.\ s.h = [m \longmapsto f\ n] \sqcup [m - n \longmapsto \mathit{map}\ \mathit{snd}\ \psi_{\mathsf{a}}] \sqcup h) \wedge\\
&\qquad (\forall F{:}\tau' \in \Gamma.\ \exists v_{\mathsf{a}}, v_{\mathsf{b}}.\ \psi_{\mathsf{a}}(F) = v_{\mathsf{a}} \wedge \sigma_{\mathsf{b}}(F) = v_{\mathsf{b}} \wedge (v_{\mathsf{a}}, v_{\mathsf{b}}) \in \overline{G}(s)(\delta\tau') \wedge\\
&\qquad \mathit{map}\ \mathit{fst}\ \Gamma|_{\mathsf{Lbl}} = \mathit{map}\ \mathit{fst}\ \psi_{\mathsf{a}} \wedge\\
&\qquad \mathit{repr}\ s.h\ (s.R(\mathsf{env}))\ (\mathit{map}\ \mathit{snd}\ \sigma_{\mathsf{a}}) \wedge\\
&\qquad (\forall y{:}\tau' \in \Gamma.\ \exists v_{\mathsf{a}}, v_{\mathsf{b}}.\ \sigma_{\mathsf{a}}(y) = v_{\mathsf{a}} \wedge \sigma_{\mathsf{b}}(y) = v_{\mathsf{b}} \wedge (v_{\mathsf{a}}, v_{\mathsf{b}}) \in \overline{G}(s)(\delta\tau')) \wedge\\
&\qquad \mathit{map}\ \mathit{fst}\ \Gamma|_{\mathrm{TVar}} = \mathit{map}\ \mathit{fst}\ \sigma_{\mathsf{a}} \wedge\\
&\qquad \mathsf{K}\, G\, s_0\, s\, K_{\mathsf{a}}\, K_{\mathsf{b}}\, j\, (\delta\tau)\, \tau'\, (\mathit{length}\ (f\ n) + m)\, K\\
&\quad \Longrightarrow\ ((m, \emptyset, \emptyset, \emptyset), (\emptyset, \emptyset, K[\sigma_{\mathsf{b}} e])) \in \mathbf{E}_{w\uparrow}(i + j)(K_{\mathsf{a}}, K_{\mathsf{b}})(G)(s_0)(s)(\bot)(\tau')
\end{aligned}$$

$$\begin{aligned}
&\boxed{\mathsf{K}\, G\, s_0\, s\, K_{\mathsf{a}}\, K_{\mathsf{b}}\, j\, \tau\, \tau'\, m\, K} = \forall G', s', v_{\mathsf{a}}, v_{\mathsf{b}}.\\
&\qquad G' \supseteq G \wedge\\
&\qquad G' \in \mathbf{GK}_{w\uparrow} \wedge\\
&\qquad s' \sqsupseteq s \wedge\\
&\qquad s'.R(\mathsf{env}) = s.R(\mathsf{env}) \wedge\\
&\qquad s'.R(\mathsf{ret}) = s.R(\mathsf{ret}) \wedge\\
&\qquad s'.R(\mathsf{sp}) = s.R(\mathsf{sp}) + 1 \wedge\\
&\qquad s'.q = [s.R(\mathsf{sp}) \mapsto v_{\mathsf{a}}] \sqcup s.q \wedge\\
&\qquad (v_{\mathsf{a}}, v_{\mathsf{b}}) \in \overline{G'}(s')(\tau)\\
&\quad \Longrightarrow\ ((m, \emptyset, \emptyset, \emptyset), (\emptyset, \emptyset, K[v_{\mathsf{b}}])) \in \mathbf{E}_{w\uparrow}(j)(K_{\mathsf{a}}, K_{\mathsf{b}})(G')(s_0)(s')(\bot)(\tau')
\end{aligned}$$

The key difference to $\preccurlyeq_{\mathcal{TI}}$ concerns the last condition, which, earlier, described how the top-most stack cells correspond to whatever σ_{b} provides for e's continuation variables. Here, however, K relates the evaluation context K (in which a closed instance of e is being executed) to the machine code that gets executed after the code for $f\ n$. This code starts at address $m + \mathit{length}\ (f\ n)$, *i.e.,* it is simply whatever comes next in memory (cf. the definition of *ezwickel*). K's definition states that, under some assumptions, this code is $\mathbf{E}$-related to the filling of K with some value v_{b}. It quantifies over an extended global knowledge and a future state s', whose register file agrees with s on ret and env. This is important, for instance, in the case of functions, whose code relies on ret having retained its original value when the function epilogue jumps to it (cf. Figure 5.19). Moreover, K assumes that the sp register has been increased by one and, correspondingly, the stack extended with a new value v_{a}. Finally, this value must be related to v_{b} at s'.

Lemma 80.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\boldsymbol{\lambda} n.\ \mathit{map}\, \mathbb{E}\, (\mathit{ezwickel}\, e\, n\, |\Gamma|)) \preccurlyeq_{\mathcal{TS}} e : \tau}$$

Theorem 23 (Correctness of Zwickel)**.**

$$\frac{\Gamma \vdash M : \Gamma'}{\Gamma \vdash \mathit{zwickel}\, M\, |\Gamma| \precsim_{\mathcal{TS}} M : \Gamma' \qquad \vdash \mathit{zwickel}\, M\, |\Gamma| : |\Gamma'|}$$

5.13 The Self-Modifying Awkward Example

As we have now seen, the PILS system from Chapter 4 can be used to verify multiple different compilers with the same source and target languages (namely at least Pilsner and Zwickel). In this section we further demonstrate PILS' flexibility by reporting on the proof of the challenging refinement from Hur and Dreyer [32] mentioned in Section 1.3, which relies on tricky manipulations of local state, far more involved than those of any traditional compiler.

This example is based on Pitts and Stark's "awkward" example, which have visited in Section 2.2 and repeat here for convenience:

$$\begin{array}{lcl} \tau & = & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int} \\ e_1 & = & \lambda f.\, (f\ \langle\rangle; 1) \\ e_3 & = & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda f.\, (x := 1; f\ \langle\rangle; !x) \end{array}$$

Recall that both expressions evaluate to higher-order functions that, when applied, call the argument "callback" function f and then return a number. In e_1 this number is simply 1. In e_3 it is the result of dereferencing a local (private) reference x, which is initialized to 0. Notice, though, that when e_3 is called for the first time, it immediately writes 1 to x. Since there are no other writes, the value of x returned at the end will always be 1 as well.

Hur and Dreyer [32] adapt this example by substituting for e_1 a tricky self-modifying machine program that implements the same behavior, but in a rather baroque way. Figure 5.20 shows what our version[7] of this program looks like in memory. It is parameterized by the load address n, and $\mathbb{E}(-)$ denotes the encoding of an instruction as a machine word. Notice that part of the code has been "encrypted" by adding 666 to its encoding. We briefly explain how the code works.

The first few lines allocate a new function closure with empty environment and code pointer $n + 5$, and return it to the context. When this function gets called the first time, it starts out by decrypting the encrypted instructions (offsets 5–11), thus replacing the encrypted code in memory. Subsequently (offsets 12–14), it replaces its first instruction by a direct jump in order to skip over the decryption loop in future executions. The remaining code (offsets 15–23), which is also the target of that jump, simply performs the callback function call and then returns 1.

Hur and Dreyer showed that this contrived implementation refines the high-level program e_3 as a demonstration that their KLR approach is flexible enough to reason about semantically involved "transformations", even ones whose correctness relies on low-level internal state changes that clearly have no high-level counterpart. By

[7] We have slightly modified the original one to account for a difference in calling convention.

offset	*word*	*comment*
$n+0$	$\mathbb{E}(\mathsf{ld\ arg\ 1})$	create and return closure
	$\mathbb{E}(\mathsf{alloc\ arg\ arg})$	
	$\mathbb{E}(\mathsf{ld\ aux_1}\ \underline{n+5})$	
	$\mathbb{E}(\mathsf{sto}\ [\mathsf{arg}+0]\ \mathsf{aux_1})$	
	$\mathbb{E}(\mathsf{jmp\ ret})$	
$n+5$	$\mathbb{E}(\mathsf{ld\ aux_2}\ \underline{n+11})$	(closure begins here)
	$\mathbb{E}(\mathsf{ld\ aux_1}\ [\mathsf{aux_2}+0])$	decrypt in loop
	$\mathbb{E}(\mathsf{bop\ -\ aux_1\ aux_1}\ 666)$	
	$\mathbb{E}(\mathsf{sto}\ [\mathsf{aux_2}+0]\ \mathsf{aux_1})$	
	$\mathbb{E}(\mathsf{bop\ +\ aux_2\ aux_2}\ 1)$	
$n+10$	$\mathbb{E}(\mathsf{bop\ -\ aux_1}\ \underline{n+24}\ \mathsf{aux_2})$	
	$666+\mathbb{E}(\mathsf{jnz\ aux_1}\ \underline{n+6})$	
	$666+\mathbb{E}(\mathsf{ld\ aux_1}\ \underline{\mathbb{E}(\mathsf{jmp}\ \underline{n+15})})$	update code
	$666+\mathbb{E}(\mathsf{ld\ aux_2}\ \underline{n+5})$	
	$666+\mathbb{E}(\mathsf{sto}\ [\mathsf{aux_2}+0]\ \mathsf{aux_1})$	
$n+15$	$666+\mathbb{E}(\mathsf{sto}\ \langle\mathsf{sp}+0\rangle\ \mathsf{ret})$	save and set return address
	$666+\mathbb{E}(\mathsf{bop\ +\ sp\ sp}\ 1)$	
	$666+\mathbb{E}(\mathsf{ld\ ret}\ \underline{n+20})$	
	$666+\mathbb{E}(\mathsf{sto\ clo\ arg})$	call function
	$666+\mathbb{E}(\mathsf{jmp}\ [\mathsf{clo}+0])$	
$n+20$	$666+\mathbb{E}(\mathsf{bop\ -\ sp\ \ sp1})$	clean up and return 1
	$666+\mathbb{E}(\mathsf{ld\ ret}\ \langle\mathsf{sp}+0\rangle)$	
	$666+\mathbb{E}(\mathsf{ld\ arg}\ 1)$	
	$666+\mathbb{E}(\mathsf{jmp\ ret})$	

Figure 5.20: Self-modifying "awkward" example

verifying the same example (with respect to $\precsim_{\mathcal{TS}}$), we aim to demonstrate that PILS are equally flexible.

Fortunately (and, as should be clear by now, not coincidentally), the high-level structure of our proof closely follows that of Hur and Dreyer's proof. Let us here merely sketch the local world that we use. Full details can be found in the Coq development.

The $\mathcal{S}$ module that we are interested in is a singleton module whose only function is a thunked e_3, *i.e.,* $\lambda\_.\, e_3$. The $\mathcal{T}$ module under consideration is a singleton module whose function has the "code" shown in the figure. A state s of our local world consists of the following:

1. The load address of the machine module. Given how loading works, this address determines the function value (*i.e.,* the heap address at which the code pointer is stored), the code address itself, and the precise memory contents at that address (*i.e.,* the code). In order for the latter to be uniquely determined, the state also records...

2. ... whether or not the code has already been decrypted. This has an all-or-nothing semantics, meaning that we do not have states representing the intermediate stages of the memory where some instructions have been decrypted but others still remain encrypted. Such states are not necessary for the proof since no functions get called during decryption.

3. The part of the machine stack occupied by not-yet returned instances of the self-modifying function. It consists entirely of return addresses, one per invocation.

4. Concerning the $\mathcal{S}$ module, the state records the allocated memory, *i.e.,* all references x (created by calling the exported function) and their current values (either 0 or 1)—in other words, the local heap.

Let's look at the possible state transitions. The load address (of the machine module) remains the same in all future states. The encryption status can change, but only from encrypted from unencrypted, after which it will remain unencrypted in all future states. The local machine stack can change arbitrarily in private, but must remain unchanged in public. Finally, the local source heap can change in two dimensions: new references can be added (when the exported wrapper function gets called), and existing references can change their contents from 0 to 1 (when one of closures gets called for the first time).

5.14 Mechanization and Extraction

Even though having our model be somewhat language-generic helped us avoid a lot of duplicate work, the mechanization effort was huge and so is the actual Coq code, consisting of roughly 13,000 lines of definitions and 24,000 lines of proofs.

In general, proofs done directly in the model tend to be very tedious—in particular when the assembly language is involved, such as the example sketch in Section 4.6. There is certainly room for more sophisticated automation here, but ultimately we would like to develop better abstractions and reasoning principles on top of the model.

From the start we formalized our work on PILS in Coq. This was critical for gaining high confidence in the results. Given the amount of technical details involved, we consider it impractical to carefully write and check many of the proofs by hand. Of course it is also immensely helpful that, after having modified some definition, Coq points one to all places that need to be updated.

Besides obviously being a very time-demanding task, the formalization also posed several other challenges. The compilation (including proof checking) of the Coq development takes very long (over 1 hour for the whole project). This seems to be not only due to the large number of definitions and proofs but also due to the large size of some proof terms. In fact we had to break up some large proofs in unnatural ways just to avoid intolerable slowdowns.

To work more productively, we wrote a script that would first compile all files with (most) proofs erased, in order to more quickly produce the ".vo" files needed for interactive development. After that, a regular build would run. We welcome the recent work on parallelizing compilation in Coq, which implements a similar idea and other features aimed at speeding up compilation and workflow [86].

Our Coq development contains a script that extracts Pilsner, Zwickel, and the linker as OCaml code, and couples the compilers with code for parsing command-line arguments as well as a lexer and parser for the source language. In order to execute target machine code, we have implemented a single-step interpretation function in Coq and proved that it conforms to the operational semantics. This function is also extracted to OCaml and wrapped in a loop. Please see the "README.txt" file for further details and instructions on how to build and run the programs.

5.15 Putting It All Together

In Section 5.11 we have established the correctness of Pilsner w.r.t. $\precsim_{\mathcal{TS}}$, and in Section 5.12 also that of Zwickel. As discussed in Section 5.13, we have moreover proven the correctness of a manual "translation" of the awkward example to a self-modifying $\mathcal{T}$ machine program.

Thanks to the modularity of PILS, specifically that of $\precsim_{\mathcal{TS}}$ (Theorem 14), this immediately means that we can link together the self-modifying machine program module with any Pilsner-produced and/or any Zwickel-produced code, and that doing so results in a $\mathcal{T}$ program that refines the linking of the corresponding sources.

Our Coq development contains one simple example of this, involving a Zwickel-compiled main function that acts as client of the awkward function and of a Pilsner-compiled factorial function. Please see the "README.txt" file for details.

Chapter 6

Related Work

Contents

6.1 Logical Relations

The body of work on applying and extending logical relations is vast. We can only give a few pointers here.

Early days The idea of logical relations is over 50 years old. Often quoted as one of the earliest use of the technique is Tait's proof of strong normalization for the typed lambda calculus [84]. Other early work includes that of Plotkin and Statman, who investigated lambda-definability [66, 67, 76].

Parametricity Girard [26, 27, 25] and Reynolds [69] generalized logical relations to the polymorphic lambda calculus, a key insight being the quantification over relational interpretations of a program's abstract types. Reynolds introduced *parametricity*, which led to further work on reasoning about abstract types [53, 54, 90, 36, 60].

State Although they were originally geared toward reasoning about pure λ-calculi, logical relations have been successfully generalized to reason about state. In Pitts and Stark's seminal work on *Kripke logical relations (KLRs)* [65], logical relations are indexed by *possible worlds*, which characterize the runtime environment (*e.g.,* the assumptions about heaps) under which two programs are considered to be equivalent.

These early KLRs for reasoning about local state imposed serious restrictions on memory contents by allowing references only to integers [65] or, say, to references of

integers [8, 68]. The first to deal with full higher-order store were Birkedal *et al.* [16, 14].

In more recent work on KLRs, Dreyer *et al.* [4, 22] showed how to generalize Pitts and Stark's technique to reason about (1) modules whose correctness proofs require fine-grained control over how local state evolves over time, and (2) ML-like languages with higher-order state. W.r.t. point (1), they model possible worlds as *state transition systems (STSs)*, as we have reviewed in Chapter 2. PBs (and PILS) adopt Dreyer *et al.*'s STS technique directly, and thus it is relatively straightforward to port all the $\mathsf{F}^{\mu!}$ equivalence proofs given in their papers from using KLRs to using PBs.

W.r.t. point (2), the challenge of supporting higher-order state in Kripke logical relations is that a naive attempt to construct a model of general reference types leads to a circularity. Intuitively, ℓ_1 and ℓ_2 are related at $\mathsf{ref}\,\tau$ under a possible world W iff W encodes the invariant that the heaps of the two programs map ℓ_1 and ℓ_2 to values v_1 and v_2 that are logically related at type τ. But how can the logical relation be indexed by a possible world W, which itself is defined in terms of the logical relation? If τ is restricted to base type (*e.g.,* int), there's no issue because the logical relation at int is simply the identity relation, but at higher type we have a problem.

Dreyer *et al.* handle higher-order state by means of Appel, McAllester, and Ahmed's technique of *step-indexed logical relations (SILRs)* [5, 2]. That is, they cut the aforementioned semantic circularity by indexing the model by a natural number ("step index") k, which represents the number of steps left on "the clock" and which gets decremented every time around the cycle between logical relations and possible worlds. As discussed in Chapter 2, it seems fundamentally difficult to compose SILR proofs transitively.

PBs and PILS employ the idea of global knowledge in order to avoid the need for step-indexing in modeling higher-order state. Specifically, by parameterizing the heap relations in our worlds over the global knowledge G, we give heap invariants a way of referring to the global value equivalence, which is essentially what the step-indexed stratification of Kripke worlds is trying to achieve as well. Our method does not bake in any syntactic typing assumptions.

Dreyer *et al.* [22] also proposed the distinction between private and public transitions to prove equivalences that only hold in the absence of control features such as call/cc. Orthogonally, they showed how equivalences that only hold when restricting store to first order can be handled by allowing a kind of backtracking in the STSs.

Our notions of logical reduction steps and stuttering from Sections 3.10 and 4.5.1 are reminiscent of Svendsen *et al.*'s *transfinite step-indexing* [83], which relaxes the coupling of a logical relation's step-index to the physical reduction steps of related programs. This enables proofs where one can decrease the step-index by an arbitrary finite amount even when there is only a single corresponding physical reduction step.

Logics Several logics based on logical relations models have been developed [23, 21, 87], with increasing power and complexity. They attempt to abstract away tech-

nical details of the models (*e.g.,* the step-indexing) and offer higher-level reasoning principles.

6.2 Bisimulations

Aside from their general coinductive flavor, PBs and PILS are closely related to two different bisimulation techniques.

From *normal form* (or *open*) bisimulations [71, 43, 78, 44, 45], we take the idea of treating unknown equivalent functions as black boxes. In particular, our expression equivalence relation $\mathbf{E}$, which deals explicitly with the possibility (in its third disjunct) that related terms may get stuck by calling unknown functions, is *highly* reminiscent of the formulation of normal form bisimulations. The main difference is that we express the notion of "stuckness" semantically, via the global knowledge parameter G, whereas normal form bisimulations express it syntactically by requiring related stuck terms to share a common head variable.

Normal form bisimulations draw much inspiration from game-semantics models [57], and our distinction between global and local knowledge has a seemingly gamey flavor as well. We leave a deeper study of the connection to game semantics to future work.

Sumii *et al.*'s *environmental* bisimulations (aka "relation-sets bisimulations") are perhaps the most powerful form of bisimulation yet developed for ML-like languages [63, 81, 41, 74, 79]. As the latter name suggests, these bisimulations are not term relations, but *sets* $\mathcal{X}$ whose elements are themselves term relations R (possibly paired with some additional environmental information, such as knowledge about the state of the heap). In essence, each $R \in \mathcal{X}$ defines some piece of "local knowledge" (following our terminology) about program equivalence. In order to show $\mathcal{X}$ to be a bisimulation, one must check that for all $R \in \mathcal{X}$, uses of terms related by R will never result in observably different outcomes and will always produce values that are related by some $R' \in \mathcal{X}$ s.t. $R' \supseteq R$.

Viewed in terms of PBs, one can understand an environmental bisimulation $\mathcal{X}$ as effectively defining an abstract state space, with each $R \in \mathcal{X}$ as a distinct state. However, the accessibility (transition) relation between these states is essentially baked in: roughly speaking, a term relation R' is (publicly) accessible from another term relation R if $R' \supseteq R$. Thus, environmental bisimulations provide less control over the structure of the transition system than PBs do, and they do not support anything directly analogous to the distinction between public and private transitions.

As a consequence, environmental bisimulations are most effective at proving equivalences that require transition systems with only public transitions (*e.g.,* the twin abstraction example), and their proofs for examples where private transitions are required (*e.g.,* the well-bracketed state change example) are comparatively "brute-force". It is an open question whether environmental bisimulations can be generalized to support the full power of PBs and PILS with both public and private transitions.

Our approach to reasoning about parametricity of ADTs, by populating the local

knowledge of a world with relations at abstract *type names*, is inspired directly by Sumii and Pierce [81].

Well-Founded Bisimulations Our technique of logical reduction is inspired by Namjoshi's *well-founded* [58, 48] bisimulations, which were developed as an alternative formulation of *stuttering* bisimulations [17] that can be checked by only reasoning about single transitions instead of infinite computations. In order to support finite but unbounded stuttering, well-founded bisimulations employ a "rank function" mapping states to some well-founded ordering, and insist (roughly) that, for states related in a bisimulation, either both make physical transitions to related states or else one side makes a transition while the rank of the pair of states decreases. In our model, we use the stutter budget to effectively bake a particular rank function into our bisimulations, which is sufficient for our purposes and convenient to work with. As far as we aware, this is the first time that the idea of well-founded bisimulations has been adapted for use in reasoning about open programs in a higher-order language setting.

6.3 Compositional Compiler Correctness

Research on compiler correctness has a long history. Here we focus only on compositionality, referring to Dave's extensive bibliography [20] for the broader area.

Using Logical Relations Benton and Hur [6] proposed the idea of using logical relations to define compositional semantics preservation. In addition to being inherently modular, logical relations are highly flexible, having been used in the past as an effective technique for proving correctness of a wide variety of program transformations in a wide variety of languages [4, 22]. Moreover, unlike contextual refinement, logical relations can be used to relate different source and target languages.

Hur and Dreyer [32] developed this idea further by formalizing the compositional correctness of a simple, single-pass compiler from an ML-like source language to an idealized assembly language. They additionally demonstrated the flexibility of their inter-language logical relations by using them to verify a contrived but illustrative example, wherein a higher-order ML function was implemented in a rather baroque way by some tricky hand-written *self-modifying* assembly code. Thanks to the modularity of their logical relations method, this highly non-standard assembly code could nonetheless be safely linked with assembly modules produced by their verified compiler, with the resulting assembly program guaranteed to preserve the semantics of the corresponding linked source modules.

Unfortunately, it is not clear how to scale Hur *et al.*'s approach from single- to multi-pass compilers because, although logical relations are modular and flexible, they are not typically transitive.

Our language-generic style of defining PILS is very much inspired by Hur and Dreyer's generic KLR setup [32]. Their point, however, was mainly to "simplify and

clarify the formal presentation"—they only ever instantiated their definitions with a single language pair and proved only a few very basic properties about the generic model; all the rest was proven about the concrete instantiation.

Multi-language semantics. Motivated by the goal of supporting compiler verification for programs that interoperate between different languages, Perconti and Ahmed [62] propose an approach based on *multi-language semantics* [49]. In particular, they define a "big-tent" language that comprises the source, target, and intermediate languages of a compiler, and provides "wrapping" operations for embedding terms of each language within the others. They then use logical relations to prove that every source module is contextually equivalent to a suitably wrapped version of the target module to which it is compiled. In this way, their method synthesizes the benefits of logical relations (modularity and different source and target languages) and contextual equivalence (transitivity).

One downside of their approach is that the intermediate languages (ILs) used in a compiler show up explicitly in the statement of compiler correctness. This leads to a loss of flexibility: the semantics of source-level linking is not preserved when linking the results of compilers that have different ILs. Another limitation with respect to flexibility is that their approach seems to be restricted to compilers that use *typed* intermediate and assembly languages, and has only so far been applied to a purely functional source language. On the other hand, Perconti and Ahmed are more flexible than we are with respect to multi-language interoperation. One of their explicit goals is to reason about the linking of ML code with *arbitrary* typed assembly code, whereas we only support verified linking with assembly modules that refine *some* source-level counterpart. As we observed in footnote 1, we do not believe this is a fundamental limitation of our approach: it should in principle be possible to develop PILS for a different source language in which high- and low-level modules may interoperate, in which case the "source"-level specification of a "target"-level module could be the target-level module itself.

In the above work, the source and intermediate languages are purely functional, and the target language is still very high-level. In recent follow-up work, Patterson *et al.* [61] develop a multi-language semantics that provides safe interoperability between the former source language and a truly low-level typed assembly language. They also construct a logical relation that supports compositional reasoning, but they do not define a compiler.

Compositional verified compilation for C. Motivated by the goal of compositional compiler verification, Beringer *et al.* [9, 77] propose an adaptation of the CompCert framework based on a novel "interaction" semantics that differentiates between internal (intra-module) and external (inter-module) function calls. They introduce a notion of "structured simulation" that assumes little about the memory transformations performed by external function calls.

Beringer *et al.*'s approach is transitive, and like Perconti and Ahmed's (but unlike

ours), it supports verified compilation of multi-language programs—in this case, programs that link C and assembly modules. However, also like Perconti and Ahmed's approach, Beringer *et al.*'s is somewhat lacking in flexibility. It depends on compiler passes only performing a restricted set of memory transformations—additional transformations could potentially break the transitivity property. In addition, their method appears to be geared specifically toward compilers à la CompCert, which employ a uniform memory model across source, intermediate, and target languages. It is not clear how to generalize their technique to support richer (*e.g.,* ML-like) source languages, or compilers whose source and target languages have different memory models.

In recent work, Kang *et al.* [37] adapt CompCert such that it supports separate compilation when all modules are compiled with the same compiler. This is clearly weaker than our horizontal compositionality. However, the restriction to a single compiler makes it possible to phrase the compositional verification in terms of the existing whole-program verification and thus reuse most of the proofs. While carrying out the work, Kang *et al.* discovered two bugs in CompCert: an incorrect axiom and an analysis that becomes invalid in the presence of linking.

Wang *et al.* [91] have also recently explored compositional compiler verification for a restricted C-like language called Cito. Their approach embeds the verification statement within a Hoare logic for partial correctness of assembly modules, thus enabling support for verified cross-language linking, but without guaranteeing preservation of termination behavior. Further work is needed to better understand the relationship between this approach and traditional refinement-based compiler verification.

6.4 Miscellaneous

Large vs. Small Worlds While PBs (and PILS) build very closely on the state transition systems in Dreyer *et al.*'s KLRs [4, 22], there is a big difference between them, which we like to think of in terms of *large* vs. *small* worlds.

Under Dreyer *et al.*'s approach, in order to demonstrate the equivalence of functions f_1 and f_2 under a "possible world" W, one proves that they behave the same when passed arguments that are related under any "future world" W' of W, which may contain arbitrary new invariants concerning the local state of other modules in the program. One can really think of the "future world" relation (*i.e.,* the Kripke structure) as defining its own transition system (or *large* world), with the possible worlds W as its states.

In contrast, our PBs rely only on *small* worlds. For us, worlds W are static entities that contain only the local invariants relevant to the module we are reasoning about, and nothing about any invariants for other parts of the program. In proving equivalence of functions f_1 and f_2 under W, we never quantify over any future worlds that extend W. Of course, in order to support compositional reasoning—*i.e.,* in order to show that consistency of worlds is preserved under separating conjunction—we must show that f_1 and f_2 behave the same when applied to arguments drawn from

some larger relation than just W's local knowledge; but for that purpose we quantify over the global knowledge G, which is not a world, but rather an arbitrary extension of W's local knowledge.

These different accounts of worlds are strongly reminiscent of the different techniques that have been proposed for modeling resource invariants in logics of storable locks. Gotsman *et al.* [30] and Hobor *et al.* [31] presented, roughly contemporaneously, two different models of a concurrent separation logic for local reasoning about programs that dynamically allocate locks and store them in the heap. The central challenge in developing such a model is in dealing with the semantic circularity that arises when accounting for locks whose resource invariants are essentially recursive.

Gotsman *et al.* deal with this circularity syntactically, by assuming a static set of named "sorts" of resource invariants, which includes not all possible invariants, but only those needed for reasoning about a particular program. In contrast, Hobor *et al.* (and more recently, Buisse *et al.* [18]) deal with the circularity head-on, defining once and for all what recursive resource invariants *mean* using step-indexing. The latter is analogous to Dreyer *et al.*'s "large worlds" approach, which defines the space of all possible heap invariants, while the former is analogous to our "small world" approach of defining only the heap invariants needed within the module we are reasoning about.

Our small-world relations seem easier to compose transitively, precisely because we make no assumption whatsoever about the relatedness of functions defined outside of whatever module we are reasoning about. That is, the global knowledge G that we quantify over (*e.g.,* when proving world consistency) could include complete garbage, and our transitivity proofs rely in a fundamental way on the surgical insertion of contentful garbage into the global knowledge.

Bibliography

[1] Samson Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. 1990.

[2] Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[3] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

[4] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *POPL*, 2009.

[5] Andrew Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

[6] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[7] Nick Benton and Vasileios Koutavas. A mechanized bisimulation for the nu-calculus. *Journal of Higher Order and Symbolic Computation*, 2013.

[8] Nick Benton and Benjamin Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, 2005.

[9] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In *ESOP*, 2014.

[10] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *LMCS*, 2(5:1), 2006.

[11] Lars Birkedal and Aleš Bizjak. A note on the transitivity of step-indexed logical relations. Manuscript, November 2012.

[12] Lars Birkedal and Robert W. Harper. Constructing interpretations of recursive types in an operational setting. *Information and Computation*, 155:3–63, 1999.

[13] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. Realizability semantics of parametric polymorphism, general references, and recursive types. In *FOSSACS*, 2009.

[14] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. Relational parametricity for references and recursive types. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 91–104, New York, NY, USA, 2009. ACM.

[15] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[16] Nina Bohr. *Advances in Reasoning Principles for Contextual Equivalence and Termination*. PhD thesis, IT University of Copenhagen, 2007.

[17] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2):115–131, July 1988.

[18] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. A step-indexed Kripke model of separation logic for storable locks. In *MFPS*, 2011.

[19] The Coq Proof Assistant. `http://coq.inria.fr/`.

[20] Maulik A. Dave. Compiler verification: A bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, November 2003.

[21] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *LMCS*, 7(2:16):1–37, June 2011.

[22] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *JFP*, 22(4-5), 2012.

[23] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, 2010.

[24] Matthew Fluet and Stephen Weeks. Contification using dominators. In *ICFP*, 2001.

[25] Jean H Gallier. On Girard's "candidats de reductibilité". Manuscript, 1989.

[26] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[27] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[28] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, October 1999.

[29] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 386–395, New York, NY, USA, 1996. ACM.

[30] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning about storable locks and threads. In *APLAS*, 2007.

[31] Aquinas Hobor, Andrew Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[32] Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. In *POPL*, 2011.

[33] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, 2012.

[34] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *POPL*, 2013.

[35] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. A logical step forward in parametric bisimulations. Technical Report MPI-SWS-2014-003, MPI-SWS, 2014.

[36] Patricia Johann and Janis Voigtländer. The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.

[37] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 178–190, New York, NY, USA, 2016. ACM.

[38] Andrew Kennedy. Relational parametricity and units of measure. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *POPL*, pages 442–455. ACM Press, 1997.

[39] Andrew Kennedy. Compiling with continuations, continued. In *ICFP*, 2007.

[40] Vasileios Koutavas, Paul Blain Levy, and Eijiro Sumii. From applicative to environmental bisimulation. In *MFPS*, 2011.

[41] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, 2006.

[42] Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL*, 2014.

[43] Soren Lassen. Eager normal form bisimulation. In *LICS*, 2005.

[44] Soren B. Lassen and Paul Blain Levy. Typed normal form bisimulation. In *CSL*, 2007.

[45] Soren B. Lassen and Paul Blain Levy. Typed normal form bisimulation for parametric polymorphism. In *LICS*, 2008.

[46] Vu Le, Mehrdad Afshari, and Zhengdong Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[47] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[48] Panagiotis Manolios. *Mechanical Verification of Reactive Systems*. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin, August 2001.

[49] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *POPL*, 2007.

[50] John Mccarthy and James Painter. Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society, 1967.

[51] Paul-André Melliès and Jérôme Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *LICS*, 2005.

[52] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[53] John C. Mitchell. Representation independence and data abstraction. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 263–276, New York, NY, USA, 1986. ACM.

[54] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[55] J. Strother Moore. A mechanically verified language implementation. *J. Autom. Reason.*, 5(4):461–492, November 1989.

[56] Lawrence S. Moss. Parametric corecursion. *Theor. Comput. Sci.*, 260(1-2):139–163, June 2001.

[57] Andrzej S. Murawski and Nikos Tzevelekos. Game semantics for good general references. In *LICS*, 2011.

[58] Kedar S. Namjoshi. A simple characterization of stuttering bisimulation. In *FSTTCS*, pages 284–296, 1997.

[59] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, 2010.

[60] Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. *JFP*, 21(4&5):497–562, 2011.

[61] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably mixing a functional language with assembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 495–509, New York, NY, USA, 2017. ACM.

[62] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, 2014.

[63] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–586, 2000.

[64] Andrew Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7. MIT Press, 2005.

[65] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[66] Gordon Plotkin. Lambda-definability and logical relations. Technical Report SAI-RM-4, Univ. of Edinburgh, School of Artificial Intelligence, 1973.

[67] Gordon Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *Combinatory Logic, Lambda Calculus, and Formalism (Curry Festschrift)*, pages 363–373. Academic Press, Amsterdam, 1980.

[68] Uday S. Reddy and Hongseok Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, March 2004.

[69] John C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, 1983.

[70] Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *Journal of Functional Programming*, 24(5):529607, 2014.

[71] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994.

[72] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2012.

[73] Davide Sangiorgi. Origins of bisimulation and coinduction. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction.* Cambridge University Press, 2012.

[74] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *TOPLAS*, 33(1), 2011.

[75] Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM (JACM)*, 60(3):22, 2013.

[76] Richard Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65(2/3):85–97, 1985.

[77] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *POPL*, 2015.

[78] Kristian Støvring and Soren Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *POPL*, 2007.

[79] Eijiro Sumii. A complete characterization of observational equivalence in polymorphic λ-calculus with general references. In *CSL*, 2009.

[80] Eijiro Sumii and Benjamin Pierce. A bisimulation for type abstraction and recursion. *JACM*, 54(5):1–43, 2007.

[81] Eijiro Sumii and Benjamin Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54(5):1–43, 2007.

[82] Eijiro Sumii and Benjamin C. Pierce. Logical relation for encryption. *J. Comput. Secur.*, 11(4):521–554, July 2003.

[83] Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. Transfinite step-indexing: Decoupling concrete and logical steps. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, pages 727–751, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[84] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[85] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 60–73, New York, NY, USA, 2016. ACM.

[86] Enrico Tassi. Coq reference manual: Asynchronous and parallel proof processing. `https://coq.inria.fr/refman/async-proofs.html`.

[87] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 377–390, New York, NY, USA, 2013. ACM.

[88] Viktor Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, 2011.

[89] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

[90] Philip Wadler. Theorems for free! In *FPCA*, 1989.

[91] Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. In *OOPSLA*, 2014.

[92] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.