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

Sigurd Schneider

A Verified Compiler for a Linear Imperative/Functional Intermediate Language

April 2018
Saarbrücken

Saarland University

Date of Colloquium . . . . . 16. November 2018
Dean of Faculty . . . . . Prof. Dr. Sebastian Hack
Chairman of Examination Board . . . . . Prof. Dr. Reinhard Wilhelm
Reviewers/Advisors . . . . . Prof. Dr. Sebastian Hack, Prof. Dr. Gert Smolka
Third Reviewer . . . . . Prof. Dr. David Pichardie
Academic Assistant . . . . . Dr. Roland Leißa

Abstract

This thesis describes the design of the verified compiler LVC. LVC's main novelty is the way its first-order, term-based intermediate language IL realizes the advantages of static single assignment (SSA) for verified compilation. IL is a term-based language not based on a control-flow graph (CFG) but defined in terms of an inductively defined syntax with lexically scoped mutually recursive function definitions. IL replaces the usual dominance-based SSA definition found in unverified and verified compilers with the novel notion of *coherence*. The main research question this thesis studies is whether IL with coherence offers a faithful implementation of SSA, and how the design influences the correctness invariants and the proofs in the verified compiler LVC. To study this question, we verify dead code elimination, several SSA-based value optimizations including sparse conditional constant propagation and SSA-based register allocation approach including spilling. In these case studies, IL with coherence provides the usual advantages of SSA and improves modularity of proofs. Furthermore, we propose a novel SSA construction algorithm based on coherence, and leverage the term structure of IL to obtain an inductive proof method for simulation proofs. LVC is implemented and verified with over 50,000 lines of code using the proof assistant Coq. To underline practicability of our approach, we integrate LVC with CompCert to obtain an executable compiler that generates PowerPC assembly code.

Kurzzusammenfassung

Diese Arbeit beschreibt das Design des verifizierten Compilers LVC. Die Hauptneuerung von LVC ist seine term-basierte Zwischensprache IL, die die Vorteile von static single assignment (SSA) für Verifikation nutzbar macht. IL ist eine termbasierte Sprache, die nicht auf einem Kontrollflussgraphen basiert, sondern auf einer induktiv definierten Syntax mit lexikalischen Variablen und verschränkt rekursiven Funktionen. IL ersetzt die übliche, dominanz-basierte SSA-Definition, die man in verifizierten und unverifizierten Compilern gleichermaßen findet, durch das neuartige Konzept der *Kohärenz (coherence)*. Die Hauptforschungsfragen dieser Arbeit sind, ob IL zusammen mit Kohärenz als Implementierung von SSA geeignet ist, und wie ein IL-basiertes Design Korrektheitsinvarianten und Beweise am Beispiel von LVC beeinflusst. Um diese Fragen zu klären verifizieren wir verschiedene SSA-basierte Wertoptimierungen, wie beispielsweise sparse conditional constant propagation, und einen SSA-basierten Registerallokationsansatz mit spilling. In diesen Fallbeispielen bietet IL mit Kohärenz die üblichen Vorteile von SSA und verbessert die Modularität der Beweise. Darüberhinaus schlagen wir einen neuen, kohärenzbasierten SSA Aufbaualgorithmus vor und nutzen die Struktur von IL aus, um ein induktives Beweisverfahren für Simulationsbeweise zu entwickeln. LVC ist mit über 50.000 Zeilen mithilfe des Beweisassistenten Coq implementiert und verifiziert. Um die praktische Anwendbarkeit unseres Ansatzes zu zeigen, integrieren wir LVC mit dem verifizierten Compiler CompCert, wodurch wir einen ausführbaren Compiler erhalten, der PowerPC assembly code generiert.

Acknowledgements

First and foremost I thank my advisers Prof. Gert Smolka and Prof. Sebastian Hack for giving me the opportunity to work on this interesting topic. This thesis necessarily contains both theoretical and practical aspects, and I am grateful to my advisers for appreciating both. I also thank the students that worked with me for their contributions to LVC, most notably Julian Rosemann. Furthermore, I thank Madlen for her love, support, and advice, and my family for being there for me. Finally, I gratefully acknowledge the Google doctoral fellowship which made this thesis possible.

Contents

1 Introduction

This thesis describes the design of the verified compiler LVC. LVC's main novelty is the way its first-order, term-based intermediate language IL realizes the advantages of static single assignment (SSA) for verified compilation. IL is a term-based language not based on a control-flow graph (CFG) but defined in terms of an inductively defined syntax with lexically scoped mutually recursive function definitions. IL replaces the usual dominance-based SSA definition found in unverified compilers as well as in verified compilers such as CompCertSSA [BDP12] and VeLLVM [Zha+12] with the novel notion of *coherence*. The main research question this thesis studies is whether IL with coherence offers a faithful implementation of SSA, and how it influences the design and the correctness invariants in the verified compiler LVC. To evaluate faithfulness, we focus on the following SSA-specific advantages:

- the reduced asymptotic complexity for SSA-based value analyses
- referential transparency of SSA variables
- the phase separation of spilling and register assignment.

In the LVC project we evaluate whether IL provides these SSA-specific advantages. We show that IL provides reduced asymptotic complexity by implementing and verifying sparse conditional constant propagation [WZ91a] and the associated program analysis in Coq. We develop a program logic that abstracts correctness proofs of value optimizations by leveraging referential transparency. We show that IL allows phase separation of spilling and register assignment by implementing and verifying a SSA-based register allocation approach [HGG06] including spilling. Furthermore, we verify dead code elimination and propose a novel, IL-specific SSA construction algorithm. To show practicability of our approach we realized the entire verified compiler LVC in the proof assistant Coq, which was a major effort with more than 50,000 lines of code. Furthermore, we integrated LVC with CompCert [Ler09b] to obtain an executable compiler that generates PowerPC assembly code.

Static Single Assignment and Coherence

SSA [Cyt+89; Zad09] is an important invariant used in almost every modern compiler at some point during compilation. Essentially, SSA allows to rename apart an imperative program: In SSA, every variable name is assigned at most once and every variable must be defined before use. Formally, "defined before use" is ensured by requiring that every use of a variable is dominated by its definition, which is sufficient to guarantee that the use is only executed if the definition of the variable has been executed before.

Semantically, the SSA invariant entails that imperative variables behave like scoped binders [Kel95; App98] even though the program semantics is still imperative. This phenomenon is often referred to as *referential transparency* of SSA variables in the compiler literature. IL accounts for referential transparency by providing two semantic interpretations: The imperative interpretation IL/I treats variables as imperative locations, and can represent non-SSA programs. The functional[1] interpretation IL/F treats variables as lexically scoped binders, which provides referential transparency in the strong sense that variables can be substituted with their defining expressions. Instead of working with a traditional SSA invariant together with an imperative semantics, LVC uses the functional IL/F without additional invariants to realize SSA. This is a conceptual advantage, because in contrast to other approaches, we do not have to maintain the SSA invariant explicitly, but just work with a functional semantics. In particular, IL/F programs satisfy the "defined before use" invariant, even if they are not renamed apart, because with lexically scoped variables, "use before definition" is impossible: such a use would refer to a *different* variable of the same name. We think that this approach provides a better semantic foundation compared to the state of the art, which uses an imperative semantics together with a dominance-based global invariant. The property that SSA programs are renamed apart is also available in our approach, because α-conversion respects IL/F equivalence and hence ensures that a program can be renamed apart if a transformation needs unique variable names.

The two interpretations of IL are semantically related through our novel notion of coherence. Coherence is a syntactic criterion that identifies a large, decidable fragment of IL programs that have the same meaning in both interpretations. Coherence enables LVC to switch between the two semantic interpretations IL/I and IL/F. Every renamed apart program is coherent.

Renaming Apart Imperative Programs: SSA Construction

Historically, SSA has been introduced as an invariant over languages based on a control-flow graph (CFG). All related work on SSA-based compiler verification uses

[1] *Functional* here only means variables are lexically scoped binders; IL/F programs can have side-effects.

such CFG-based languages. However, such imperative programs cannot simply be renamed apart, but require a special operator, called ϕ-functions, to realize a control-flow dependent copy operation at control-flow join points. The original imperative semantics of the CFG must be extended with the ϕ-function's special semantics. An algorithm of great importance is the SSA construction algorithm [Cyt+91], which places enough ϕ-functions to ensure the SSA-invariant holds. SSA construction algorithms that place ϕ-functions have been verified in the context of CompCertSSA [BDP12] and VeLLVM [ZZ12].

Our coherence-based approach results in a novel SSA construction algorithm. Our approach does not require additional syntactic constructs such as ϕ-functions, but SSA construction on IL can simply add additional function parameters. SSA construction realizes a semantics-preserving transformation from an IL/F program to a coherent IL/F program. We verify a SSA construction algorithm that uses liveness information to determine a set of additional parameters for each function which is large enough to ensure the resulting program is coherent. The set of additional parameters is small in the sense that it only contains live variables, hence our construction algorithm produces guarantees similar to pruned SSA form [CCF91]. In contrast to all other SSA construction algorithms, our algorithm only needs to establish coherence and does not have to rename apart the program. This is a conceptional simplification witnessed by the fact that liveness information remains the same after applying our SSA construction algorithm.

Value Optimizations and Referential Transparency

SSA simplifies optimizations both conceptually and in practice. Many of these advantages stem from the fact that every definition can be uniquely identified by the corresponding variable name. In particular, static analyses can represent the result of a program analysis as one global mapping from variable names to analysis information instead of maintaining such a mapping per program point. This decreases the asymptotic size of program analysis information by a factor proportional to the program size, and was one of the original motivations for the development of SSA [RWZ88; AWZ88]. LVC realizes several SSA-based value optimizations in the middle end on renamed apart IL/F programs and obtains the same reduction in asymptotic analysis information size as SSA with our design based on coherence.

We reformulate and prove correct copy propagation and sparse conditional constant propagation (SCCP) [WZ91a] on renamed-apart IL/F programs. The verification of the optimizations is done on a renamed apart, and hence coherent, IL/F program, and the SCCP analysis relies on uniqueness of variable names. The correctness proofs exploit referential transparency of SSA variables and can be argued correct locally: We must argue that replacing a simple expression e with a concrete value is justified given certain information about the context, which amounts to a substitu-

tion property. This is a reasoning principle that occurs in the correctness proofs for value optimizations in general, and we develop a program logic that encapsulates it and can be used whenever the correctness of an optimization essentially follows from referential transparency. The program logic isolates the correctness proof of a transformation from the semantics of the intermediate language and the notion of simulation-based semantic equivalence. When using the program logic, only the transformations of expressions must be justified, and a general soundness lemma for the logic lifts this argument to the correctness of recursive IL functions. In this way, the logic provides a form of modularization for the correctness proofs.

Register Allocation

Register allocation transforms a program in such a way that it only uses a fixed number of registers. While non-SSA-based register allocation has been verified in the context of two different compiler verification projects [Tan+16; BRA10], we know of no other verified SSA-based register allocator. A property unique to SSA is that register allocation can be organized in two separate phases [HGG06]. The first phase is spilling, which lowers the number of maximal simultaneously live variables to the register bound by transferring values to memory. The second phase is the register assignment phase, which then only requires polynomial time to rename the program in such a way that it only uses a fixed number of registers that is bounded by the number of maximal simultaneously live variables. LVC realizes this SSA-specific advantage with its intermediate language IL and coherence, which is another indication that our coherence-based approach can serve as a faithful implementation of SSA. In the correctness proof of register allocation, all advantages of coherence come together. We switch to IL/I for the verification of spilling, and back to IL/F for register assignment. The correctness of register assignment follows from the fact that α-conversion respects IL/F semantics and the observation that the register assignment realizes an α-renaming. A interesting finding in our work is that while the original SSA-based register allocation algorithm [HGG06] requires traversal of the CFG in dominance order, it suffices to traverse the IL program recursively. Clearly, this traversal order is compatible with the dominance ordering, but possibly coarser. Although the quality of the register allocation may improve by using the dominance ordering, its correctness only depends on the fact that the traversal order is compatible with the dominance ordering.

Liveness, Reachability, and Dead Code Elimination

The notion of liveness and reachability are central in LVC. Coherence is defined relative to liveness information. Register allocation relies on liveness information and yields better results if no dead variables are present. Furthermore, we show that register assignment realizes an α-renaming on programs without dead code.

LVC realizes dead code elimination (DCE) with two separate transformations. The first transformation removes unreachable code based on a verified reachability analysis. We show that this transformation is complete relative to a definition of reachability that evaluates constant conditions. The second transformation eliminates unused variables and uses information from a liveness analysis.

The soundness criterion for liveness information differs between IL/I and IL/F. The register allocation phase needs to switch between the semantic interpretations IL/I and IL/F. This is only possible because we show that on coherent programs without unreachable code, liveness information that is sound for IL/I is also sound for IL/F.

Translation to Assembly and Integration with CompCert

IL's two semantic interpretations also enable compilation to be realized as "transformation to a fragment" as proposed by Kelsey [KH89]. His observation is that compilation can be seen as transformation into successively more restricted subsets of the same language, such that the final subset is very easy to translate to assembly. This idea is in contrast to the approaches taken by CompCert [Ler09b], which uses 9, and CakeML [Tan+16], which uses 12 very different intermediate languages. We think this strategy is a good fit for verified compilation in general, but especially so for LVC, which is centered around IL/I and IL/F: We need to specify drastically fewer semantics than related approaches (only 2: IL/I and IL/F), and can reuse proof methods for IL/I and IL/F in every transformation.

An additional feature that makes Kelsey's idea work so well for LVC is that IL/I is a rather low-level language: IL/I without function parameters already behaves like a register transfer language, in particular, function application corresponds to `goto`s. We hence view the compilation task as an endo-translation on IL, thereby leveraging coherence to switch between IL/F and IL/I when useful, and ultimately ending in the argument-less fragment of IL/I. We then translate the argument-less fragment to CompCert's intermediate language "Linear", that is, to machine code where only the layout of the stack frame is missing.

Semantic Equivalence and System Calls

All transformation steps are verified in a bisimulation-based framework for semantic equivalence we developed. The main challenge for semantic equivalence is the formalization of coinductive proofs in Coq. IL/F and IL/I both feature external calls which behave non-deterministically and serve as a benchmark for our semantic techniques. For the correctness proofs of the optimizations, the treatment of external calls is largely orthogonal to the techniques we develop for coherence. We develop an inductive proof method for simulation that supports stutter-steps and proof modularization better than the methods applied in CompCert and better than parametric coinduction alone [Hur+13].

Translation Validation

Translation validation [PSS98] is a technique that avoids proving correctness properties of a function, and instead uses a validated decider to test whether the result of the function has the required properties after each application of the function. Translation validation was successfully applied in CompCert [Ler09b], because designing and proving correct a decider for a certain property is often much simpler than showing that a complex algorithm establishes the property.

LVC explicates the specification of many kinds of analysis information as inductive predicates. Examples are the correctness of liveness and reachability information, coherence of a program, the correctness of spilling decisions and register assignments. We give constructive proofs that all these specifications are efficiently decidable, which realizes extractable translation validators for these components. In joint work with Julian Rosemann [RSH17], we developed an novel approach that we call "translation-validation with repair" and which extends translation validation.

All transformations currently in LVC are verified and do not rely on translation validated input; however, spilling and spill slot coalescing (i.e. the problem which spill slots can share the same stack slot) accept unverified suggestions that are then validated and repaired if necessary by LVC. LVC could be changed with minor effort to allow external procedures also for register allocation itself, but this is not included in the thesis. For analysis information, such as liveness and reachability information, translation validation does not make sense, because our verified analysis algorithm is already precise. A major advantage of this approach is the following implementation strategy: First prototype an unverified algorithm and integrate it using translation validation (with repair), and once satisfied with the design, prove it correct once and for all.

Realization in Coq

LVC is realized in the proof assistant Coq, which is based on constructive type theory, but remains compatible with classical assumptions (such as excluded middle). The entire LVC compiler has been realized and verified in Coq. We obtain an executable compiler from the Coq development via extraction to OCaml code. In the Coq development, we avoid additional assumptions via axioms (such as excluded middle) whenever possible. We only use excluded middle and informed excluded middle for two proofs in the meta-theory of program equivalence. Two libraries we use require Uniqueness of Identity Proofs (UIP) and functional extensionality. Realizing the project in Coq turned out to be a major effort, the LVC development is approximately 50k LoC. More information about the Coq implementation can be found in §17. The Coq development is available online on GitHub[2] under MIT license.

[2] https://github.com/sigurdschneider/lvc

1.1 Related Work

We give a brief overview over the most important differences to related work here, and postpone an in-depth discussion to §18.

1.1.1 Term-based SSA instead of CFG with ϕ-functions

LVC uses the term-based language IL instead of a control-flow graph (CFG) with ϕ-functions and an explicit SSA-invariant, which is the basis for all other verified SSA approaches [BDP12; Zha+12]. LVC uses the functional IL/F to represent SSA programs, which makes ϕ-functions and an explicit SSA invariant unnecessary. The notion of coherence supports translating in and out of SSA, for correctness it suffices to maintain program equivalence, but not necessarily coherence. To rename a program apart, the functional semantics can be chosen, and the program can be α-converted if a renamed apart program is required. In contrast to related work, which must maintain the SSA invariant at all times, transformations in LVC leverage that neither coherence nor renamed apartness have to be maintained: For example, our spilling phase requires the input program to be coherent and renamed part, and breaks both invariants, thereby greatly simplifying the introduction of spill slots. Without breaking coherence, the spilling phase would have to perform an SSA construction algorithm; we re-use the SSA construction algorithm LVC contains anyway to obtain a coherent program after spilling. We think that coherence here helps to better factor concerns. Another example is register assignment, requires the input program to be renamed apart, but produces an imperative program that is, in general, not renamed apart program, but coherent, which accommodates the facts that one the one hand registers are imperative variables, but on the other hand correctness of register assignment follows from soundness of α-conversion (which only holds in the functional interpretation).

1.1.2 IL Replaces Many Intermediate Languages

LVC uses the intermediate language IL with its two semantic interpretations IL/I and IL/F, instead of related approaches [Ler09b; Owe+16] which use up to 12 different intermediate languages, essentially one per translation pass. We think having fewer languages is an advantage, because the correctness methods we develop for IL can be reused in the correctness proofs of the optimizations. Furthermore, we only have to specify the semantics for IL/I and IL/F, instead of dealing with a multitude of intermediate languages.

IL is very flexible: IL/F can represent SSA-programs, IL/I can express low-level programs and its parameter-less fragment is close to assembly. IL/I supports the idea of "transformation to a fragment" by Kelsey [KH89], and allows us to stay in IL/I until we translate to a list of machine instructions.

1.1.3 Invariance as Semantic Foundation for SSA

We developed coherence as a sufficient criterion for invariance, i.e. such that coherence suffices for a program to have the same meaning in IL/I and IL/F. Finding a sufficient criterion for invariance is a well-posed problem with a semantic foundation that does not depend on dominance or other complicated notions. This is in stark contrast to other SSA formalizations, which take the definition of SSA from literature [Cyt+91] as ground truth and make no attempt to characterize SSA semantically. As a result, LVC's coherence-based framework is more flexible than SSA, requires fewer invariants, and makes explicit which translation passes require the program to be renamed apart, and which just require a functional semantics.

1.1.4 SSA-based Register Allocation

Register allocation has been verified in the context of CompCert [Ler09a; BRA10], LambdaTamer [Chl10] and CakeML [Tan+16]. To our knowledge, LVC features the first verified SSA-based register allocator. CompCertSSA [BDP12] uses CompCert's non-SSA allocator after doing a generic out-of-SSA translation. The approach followed in LambdaTamer is effectively block-local (see also §18.11.4). The approaches verified in the context of CompCert [BRA10] and CakeML [Tan+16] consider the iterated register coalescing algorithm [GA96]. Register allocation in CompCert is not verified, but translation validated to obtain higher code quality [RL10]. We think that our SSA-based approach [HGG06] has the potential to produce register allocations of high quality, just as the non-verified implementations of the approach.

1.1.5 Inductive Correctness Methods

We propose an inductive proof method for simulation-based compiler correctness. This method is based on compatibility lemmas to modularize the proof. Our method has better modularity properties than the simulation proofs used in CompCert and derivatives [Ler09b; Sev+13] and the parametric bisimulation approach [Hur+13].

1.1.6 Leveraging Referential Transparency of SSA Variables

LVC supports a program logic that leverages referential transparency for the verification of value optimizations. We think that using a program logic as abstraction for correctness proofs of SSA-based value optimization is an effective way to modularize correctness proofs for the classical SSA-based optmizations, of which we verify SCCP as example.

1.2 Motivation

The main motiviation for this work was to study the design of a compiler with a term-based intermediate language with binding to realize SSA, and the impact of the design on optimizations and their correctness proofs on functional SSA. A second goal was to create an extensible research compiler that is realistic in the sense that extraction of a compiler with at least polynomial runtime is possible.

During this research, we discovered the notion of coherence and that it connects standard notions from compiler construction with standard notions from programming language theory. Of particular interest for us are the interaction of coherence with the notion of liveness and its application for the verification of register allocation. We think that this work is a step towards bridging the gap between practical compiler construction, where the imperative view with CFGs and ϕ-functions prevails, and compiler and programming languages research, where we understand SSA as form the functional perspective as binding. Bringing these two worlds together provides interesting insights. For example, coherence allows formally justify the correctness of register assignment by showing that it is, in fact, an α-renaming. Coherence also provides a better factoring for spilling: We do not need to perform SSA construction in the spilling phase, because in LVC, coherence is only established when it is useful, but not always maintained as an invariant.

1.3 Outline

We start with some preliminary definitions in §3, which presents standard concepts and explains how we formalized them given the constraints of extractability to OCaml. We introduce the intermediate language IL in §4 together with its two semantic interpretations. In §5 we explore semantic equivalence and the implementation relation we use in LVC. We give different trace based and several bisimulation-based characterizations, and explain the tools required to show some basic properties of bisimilarity. In §8, we discuss compatibility results for bisimilarity, including a general compatibility result for function definitions which enables an inductive proof method for correctness proofs on IL.

§2 provides provides an overview with examples over LVC and outlines §9 to §16, which discuss each phases of LVC in detail.

In §17 we discuss technical details of the Coq development, such as our custom plugins and tactics, as well as the number of lines of code and the module structure. In §18 provide an in-depth discussion of related work. We conclude in §19.

1.4 Contributions

- We are the first to explore term-based SSA for verified compilation on our intermediate language IL.
- We propose the notion of coherence, which generalizes the notion of SSA and helps formally connect notions from compiler construction with notions from programming language theory.
- We verify SSA-based register allocation on IL with the help of coherence. For this purpose, we developed a framework for the verification of spilling algorithms, and a framework for the verification of register assignment algorithms.
- We propose a novel, coherence-based SSA construction algorithm for IL that is conceptually simpler than the standard SSA construction algorithm [Cyt+91] because establishing coherence does not require to rename apart the program and does not require dominance information.
- We develop a framework for the verification of value optimizations, and apply it to copy propagation and sparse conditional constant propagation. The framework includes a program logic especially tailored to decouple the verification process from the simulation equivalence.
- We develop a framework for program analyses on IL and use it to verify correctness of liveness analysis, reachability analysis, and sparse conditional constant propagation analysis.
- We explore several formulations of bisimulative equivalences in Coq and evaluate them based on their support for proof modularization, in particular whether they easily accommodate stutter steps, and whether transitivity of the simulation relation can be leveraged directly in coinductive proofs.
- We integrate out approach in a rudimentary way with CompCert, and in this way show that IL can indeed be used as intermediate language in the back-end of a realistic compiler.

1.4.1 Published Results

During his time as a PhD student at Saarland University, the author of this thesis submitted 6 papers, of which three were accepted. This thesis builds on the following peer-reviewed papers:

- Sigurd Schneider, Gert Smolka, Sebastian Hack:
  “A Linear First-Order Functional Intermediate Language for Verified Compilers.”
  Iteractive Theorem Proving (2015).

- Steven Schäfer, Sigurd Schneider, Gert Smolka:
  "Axiomatic semantics for compiler verification."
  Certified Programs and Proofs (2016).

- Julian Rosemann, Sigurd Schneider, Sebastian Hack:
  "Verified Spilling and Translation Validation with Repair."
  Iteractive Theorem Proving (2017).

Additionally, a technical report is available:

- Sigurd Schneider, Gert Smolka, Sebastian Hack:
  "An Inductive Proof Method for Simulation-based Compiler Correctness."
  CoRR abs/1611.09606 (2016).

2 Compiler Overview

In this section, we informally introduce the language IL, and give an overview over the three main parts of the compiler: the front-end, the middle-end, and the back-end. Each of the three parts contains multiple compilation phases.

2.1 The Intermediate Language IL

see §4

This section gives a quick and informal introduction to the intermediate language IL and its two semantic interpretations IL/I and IL/F, and how they are connected via coherence. We also discuss the semantic impact of system calls.

2.1.1 IL/F and IL/I

see §4

Consider the program in Figure 2.1, which behaves exactly like one would expect from a function program: In line 2, a closure is created that saves the value of x such that the program ultimately yields 7.

```
1 let x = 7 in
2 fun f () = x in
3 let x = 5 in f ()
```

Figure 2.1: An IL/F program that yields 7.

The other semantic interpretation IL/I interprets the program in Figure 2.1 as is indicated by the different notation in listing Figure 2.2. IL/I treats x as a location, and under IL/I interpretation the program yields 5 as result. Note that we used the stylized notation `:=` to emphasize that the variables in IL/I behave like imperative locations, but use the `let`-syntax for both interpretations throughout the thesis: In fact, Figure 2.2 and Figure 2.1 are the same program (i.e. identical abstract syntax tree), and IL/F and IL/I provide two different semantic interpretations.

```
x := 7;
fun f () = x in
x := 5; f ()
```

Figure 2.2: The program from Figure 2.1 stylized to hint at its IL/I interpretation, which yields 5.

2.1.2 Coherence

see §10

The main observation behind coherence is that some IL programs have the same meaning in both interpretations. For example, each program in Figure 2.3 has the same meaning in both IL/I and IL/F: The program on the left yields 5 in both IL/F and IL/I, and the program on the right yields 7 in both IL/F and IL/I.

```
let x = 7 in
fun f (x) = x in
let x = 5 in f (x)
```

```
let x = 7 in
fun f () = x in
let y = 5 in f ()
```

Figure 2.3: Two coherent programs: Both programs have the same meaning in both IL/F and IL/I.

The program on the left has the same meaning as the IL/I interpretation of Figure 2.1, and can be obtained from it by adding the parameter x to f. Similarly, the program on the right has the same meaning as the IL/F interpretation of Figure 2.1, and can be obtained by renaming apart Figure 2.1.

Coherence is a syntactic criterion sufficient to ensure that IL/I and IL/F agree on the meaning of a program. Both programs in Figure 2.3 are coherent. The simplest way to establish coherence while preserving IL/I semantics is to add all variables occurring in the program as parameters to every function. We discuss a more clever approach in detail in §12. The simplest way to establish coherence while preserving IL/F semantics is to simply rename apart the program. A more clever approach is performing register assignment, which we discuss in detail in §13.4.

2.1.3 System Calls

IL features external events, which are an abstract concept that subsumes system calls and interactions with memory. System calls in IL take the form

$$\texttt{let x = } \alpha\, \overline{e} \texttt{ in } \ldots$$

where α is a event identifier from a countably infinite alphabet. In the system call metaphor, α is the name of the system call. The expressions $\overline{e}$ are evaluated, and their values are made visible externally. The system call may then return a value, which may or may not depend on its arguments. For this purpose, we allow non-determinism in the semantics, that is, an external event may produce any value.

The treatment of external events is orthogonal to the correctness arguments for the approach developed in this thesis. Their presence, however, witnesses that our techniques scale to extensions of IL with memory and proper system calls. In this way, the presence of external events in IL serves as a benchmark for our semantic techniques.

2.2 The Front-End

The front-end starts with an IL program from the parser, which is interpreted as an IL/I program. The parser is not verified. This IL/I program still uses explicit names for functions instead of De-Bruijn indices. The front-end consists of the following three phases, which are verified and ultimately translate to a coherent program.

1. convert function names to De-Bruijn indices
2. dead code elimination
3. establish coherence (i.e. SSA)

Converting a program with explicit names to De-Bruijn can be done by a simple recursive traversal while maintaining a mapping from function names to De-Bruijn indices. The correctness proof is in the formal development, but we will not mention it any further.

2.2.1 Dead Code Elimination

Dead code elimination (DCE) consists of unreachable code elimination (UCE) and dead variable elimination (DVE), which are run in this order.

The unreachable code elimination phase relies on a reachability analysis. Unreachable code arises from constant expressions in conditionals and function definitions that are never called. Figure 2.4 shows a program where UCE removes unused function definitions.

Dead variable elimination (DVE) relies on a liveness analysis to determine which let-bindings and parameters are unused, and removes them.

Intuitively, a variable is live if it contributes to the behavior of the rest of the program. For example in Figure 2.5, liveness analysis finds out that the parameter y

```
fun f (x,y) =
 if (9 > 3) then 1
 else g (x+1, y)
and fun g (x,y) =
  f(x+1,y/10)
in f (3,2)
```

```
fun f (x,y) = 1


in f (3,2)
```

Figure 2.4: An example program before (left) and after (right) unreachable code elimination.

```
fun f (x,y) =          {}
 if (x > 9) then 1     {x}
 else f (x+1, y)       {x}
in let a = 5 + b       {}
in f (3,a)             {}
```

```
fun f (x) =
 if (x > 9) then 1
 else f (x+1)

in f (3)
```

Figure 2.5: An example program before (left) and after (right) dead variable elimination. Each line in the left-hand side program is annotated with the set of live variables before the respective statement.

in Figure 2.5 does not contribute to the behavior of the program, and can hence be removed. In turn, the values of a, b also do not contribute to the behavior of the program, and are removed as well.

2.2.2 SSA Construction

see §12

It is easy to update liveness information to take the effect of DVE into account and obtain valid liveness information for the resulting program. This liveness information is then used in the SSA construction algorithm. The SSA construction algorithm adds just enough parameters to function definitions for the program to become coherent. This means that in our setting, SSA construction does not rename apart the program, but instead ensures that in the resulting program variable definitions behave live lexically scoped binders.

SSA construction introduces parameters to make dataflow explicit that is already implicitly present in the program. In Figure 2.6, the function f reads a variable x, and x might have changed since between the definition of f and an application of f, so x is added as a parameter to f. SSA construction inserts a parameter with the same name to the function. This process, however, does not change liveness

```
1 let x = 7 in
2 let a = 4 in
3 fun f () = x+y in
4 let x = a+5 in f ()
```

```
1 let x = 7 in
2 let a = 4 in
3 fun f (x) = x+y in
4 let x = a+5 in f (x)
```

Figure 2.6: An example program before (left) and after (right) SSA construction. Note that the program on the right is coherent, but not renamed apart. Note no argument needs to be introduced for the parameter y, because it does not change between the definition of f and the application of f.

information. Liveness information before SSA construction is also valid after SSA construction.

2.3 The Middle-End

The middle end only features two optimizations: copy propagation and sparse conditional constant propagation. Both optimizations require the program to be renamed apart. The middle end performs the following translation phases:

1. rename apart the program
2. copy propagation
3. sparse conditional constant propagation

2.3.1 Copy Propagation

see §14

After renaming apart, which we discussed earlier, copy propagation is performed. Our version of copy propagation is realized by a simple recursive traversal, and does not require a fixed-point analysis.

2.3.2 Sparse Conditional Constant Propagation

see §14

Afterwards, the sparse conditional constant propagation SCCP is performed. SCCP evaluates constant expressions and replaces them by values. SCCP relies on a fixed-point analysis to find constant function parameters.

Note that in LVC neither copy propagation nor SCCP remove variable bindings. Instead, both optimizations rely on a DCE pass to be performed later. In the examples above, we have already included the effect of the DCE pass for the sake of better illustration.

```
fun f (x,y) =
  if (x+y > 10) then
    let b = y in
    1
  else f (x, y+2)
in let x = a
in f (x,0)
```

```
fun f (x,y) =
  if (x+y > 10) then

    1
  else f (x, y+2)

in f (a,0)
```

Figure 2.7: An example program before (left) and after (right) copy propagation. Note that the parameter x is also just a copy of a, but our version of copy propagation cannot not detect this.

```
let a = 7 in
fun f (x,y) =
  if (x+y > 10) then
    y+1
  else if (y < 4) then
    f(y,x)
  else f (x+1+2, y)
in f (3+a+z,5)
```

```

fun f (x) =
  if (x+5 > 10) then
    6


  else f (x+1+2)
in f (10+z)
```

Figure 2.8: An example program before (left) and after (right) SCCP. Note that SCCP parameter x is also just a copy of a, but our version of copy propagation cannot not detect this. Note simplification of $x + 1 + 2$ is prevented by left-associativity of + and we currently do not re-associate expressions.

2.4 The Back-End

Figure 2.9 shows an overview over the transformation in the LVC back-end. On a high-level, LVC's back-end consists the following transformations, in order:

1. elimination of argument expressions
2. renaming apart to register partition
3. dead code elimination
4. register allocation
   a. spilling

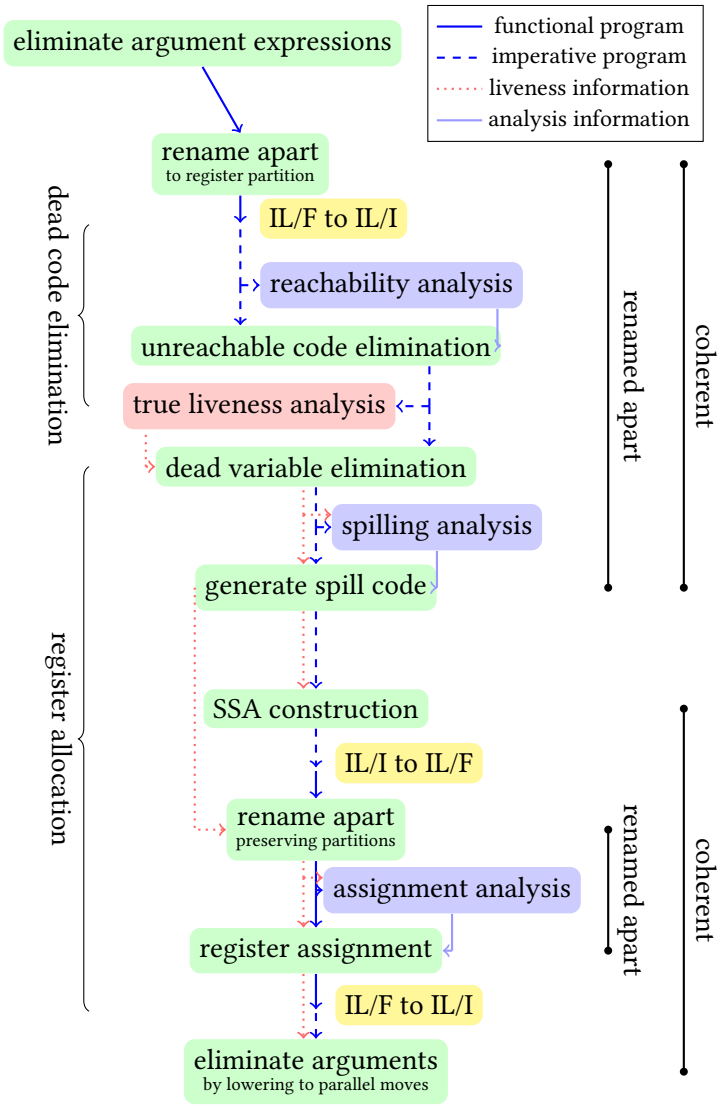

Figure 2.9: Overview over the back-end compilation phases. The spilling analysis supports translation validation with repair.

   b SSA construction
   c renaming apart
   d register assignment

5 eliminating parameters via parallel moves

6 lowering to assembly

Each of these steps consists of several sub-phases, which we now explain high-level with the program in Figure 2.10 as example.

```
1  fun f (x, y, z) =
2   if x > 0 then
3    f (x-1, y, z+z)
4   else
5    if x = 0 then
6     let v = z + y in
7     v + w
8    else
9     f (y-x, -y, z)
10 in f (-4, 2, 1)
```

Figure 2.10: The example program for the back-end: We describe how this program is transformed into machine code. For the sake of the example, we assume that only two registers are available.

2.4.1 Elimination of Argument Expressions

The first transformation before the back-end is the elimination of argument expressions. For this purpose, for each argument that is not already a variable, a let expression is introduced. Note that this transformation is done on IL/F, so finding names for these new variables is easy, as their scope is very limited.

The elimination pass involves a scheduling problem, because the order of the let-expressions determines the evaluation order and the register demand of the resulting program. Note that in Figure 2.11, d and x can be assigned to the same register, and c and y can also share the same register. Had argument elimination in line 10 and 11 in Figure 2.11 produced the following order, this would not be possible:

```
1     let d = -x in
2     let c = y - x in
3     f (c, z, d)
```

```
1  fun f (x, y, z) =                    {w}
2   if x > 0 then                       {x, y, z, w}
3    let a = x - 1 in                   {x, y, z, w}
4    let b = z + z in                   {a, y, z, w}
5    f (a, y, b)                        {a, b, y, w}
6   else
7    if x = 0 then                      {x, y, z, w}
8     let v = z + y in                  {y, z, w}
9     v + w                             {v, w}
10   else
11    let c = y - x in                  {x, y, z, w}
12    let d = -x in                     {c, x, z, w}
13    f (d, z, c)                       {c, d, z, w}
14 in
15   let i = -4 in                      {w}
16   let j = 2 in                       {i, w}
17   let k = 1 in                       {i, j, w}
18   f (i, j, k)                        {i, j, k, w}
```

Figure 2.11: Figure 2.10 after elimination of argument expressions. Each line is annotated with the set of live variables before the respective statement.

For the code above, LVC would need to spill x to ensure its value is available to compute c. Another solution would be to rematerialize x by changing $y - x$ to $y + d$, but LVC does not currently attempt to rematerialize values. We do not consider this optimization problem and introduce lets in the order of the argument vector. In a more realistic compiler, a scheduling phase would have attempted to take care of this problem. However, it is perfectly reasonable to assume that argument expressions are eliminated once the back-end is entered. The main purpose of the pass as we implemented it is to ensure that every argument to a function call is a variable, which is required for our register allocation phase to work.

2.4.2 Renaming Apart to the Register Partition

We partition the variables into two infinite partitions: The registers and the spill slots. The back-end renames the program apart, and uses only names from the register partition. This also ensures that the program is coherent. We use the convention that lower-case variables are registers, and upper-case variables are spill slots. For the sake of the example, we arranged things such that Figure 2.11 is already renamed apart to the register partition, so we can assume the program does not change. Fig-

ure 2.11 is coherent, because it is renamed apart.

2.4.3 Dead Code Elimination in the Back-End

The DCE phase exploits that the program is coherent and uses imperative liveness analysis. This enables LVC to re-use the liveness information computed during DCE in the register allocation phase, which needs imperative liveness information. The re-use of liveness information all the way down to the register assignment phase is indicated by the dotted arrows in Figure 2.9.

The DCE proceeds in the same way as described in §2.2.1 for the front-end. The live-sets are annotated in Figure 2.11. Note that in general, a program is still renamed apart after DCE, and hence still coherent (cf. §2.2.2).

The liveness information after DVE makes every variable live at least at the program point directly following its definition. This property follows from properties of DVE for all variables except those bound to the result of an external function call, for we enforce it in the algorithm that updates liveness information to reflect the changes due to DVE. After DVE, the program is coherent and the register demand is less or equal to the size of the largest live set.

We arranged things such that Figure 2.11 had no dead code, and can hence assume that the program does not change. LVC might α-rename the program, because our procedure for renaming apart is not stable for performance reasons.

2.4.4 Register Allocation

see §13

Register allocation consists of two separate, consecutive phases: spilling and register assignment.

Spilling

see §13.2

Register allocation begins with the spilling phase, the goal of which is to lower the size of the largest live-set, which is called register pressure, to a certain number k. This is done by storing values to variables from the second partition, the spill slots, which we do not count towards register pressure. Spill slots will later be mapped to locations in memory. The spilling phase inserts spills and reloads into the program to lower the register pressure sufficiently, while ensuring that all variables are in registers when used in a computation such as addition, subtraction, etc. The input program for spilling is Figure 2.11, and we can see that the size of the maximal live set is 4.

The spilling phase is verified with respect to the imperative semantics, as this simplifies the handling of spills and loads: A spill slot is simply a global variable which

can be written to and read from anywhere. Furthermore, we can easily map register variables to their corresponding spill slots.

The program after spilling is shown in Figure 2.12. As mentioned before, we follow the convention that lower-case variables denote registers, and upper-case variables denote spill slots. We assume $k = 2$, i.e. that there are only two machine registers, to ensure that spilling is necessary.

```
1  fun f (x, Y, z) =                    {W}
2   if x > 0 then                       {x, Y, z, W}
3    let a = x - 1 in                   {x, Y, z, W}
4    let b = z + z in                   {a, Y, z, W}
5    f (a, Y, b)                        {a, b, Y, W}
6   else
7    if x = 0 then                      {x, Y, z, W}
8     let y = Y in                      {Y, z, W}
9     let v = z + y in                  {y, z, W}
10    let w = W in                      {v, W}
11    v + w                             {v, w}
12   else
13    let Z = z in                      {x, Y, z, W}
14    let y = Y in                      {x, Y, Z, W}
15    let c = y - x in                  {x, y, Z, W}
16    let d = -x in                     {c, x, Z, W}
17    f (c, d, Z)                       {c, d, Z, W}
18 in
19   let W = w in                       {w}
20   let i = -4 in                      {W}
21   let j = 2 in                       {i, W}
22   let J = j in                       {i, j, W}
23   let k = 1 in                       {i, J, W}
24   f (i, J, k)                        {i, J, k, W}
```

Figure 2.12: Figure 2.11 after spilling. Each line is annotated with the set of live variables before the respective statement. Note that the maximal number of simultaneously live registers (denoted by lower-case variables) is 2.

LVC supports translation validation with repair for spilling. In particular, it currently asks an outside program to choose the variable to be spilled at every program point. This external information cannot introduce unsoundness, as LVC repairs it if necessary. Since LVC will repair any input (including none), LVC does not depend

on external information for compilation. We use the possibility to provide spilling information to obtain the optimal spilling displayed in Figure 2.12. In particular, we ensure that w gets spilled early in line 19, at a program point where register pressure does not yet warrant spilling and that the parameter y is always passed in a spill slot instead of a register.

SSA Construction

see §12

After spilling, the program is neither renamed apart nor coherent, that is, only IL/I captures its intended semantics. For example, the program in Figure 2.12 does not have the same semantics in IL/I and IL/F, because W is updated after f is defined. We perform a SSA construction pass to obtain a coherent program again. As discussed in §2.2.2, SSA construction just adds additional parameters, does not change liveness information, and produces a coherent, but not necessarily renamed apart program.

To obtain a coherent program again, SSA construction adds W as a parameter to f. Figure 2.13 shows the program after SSA construction and renaming apart.

Register Assignment

see §13.4

On a renamed apart program, the register assignment can be represented as one global mapping from variable names to registers. A register assignment is computed in a SSA fashion [HGG06], and the program is subsequently renamed accordingly. Together with the fact that the program contains no dead code because of the DCE (§2.4.3), register assignment results in an α-equivalent program that is still coherent, but not necessarily renamed apart anymore. We exploit that α-equivalent programs are semantically equivalent to justify correctness.

The register allocation phase also performs spill slot coalescing, i.e. renaming several slot variables to the same spill slot to save stack space and avoid the need for moving values between spill slots as part of parameter passing. LVC currently supports translation validation with repair for spill slot coalescing, and we provide the mapping $A \to W; J \to Z; Y \to Z;$ to LVC to obtain reasonable sharing and avoid spill slot to spill slot copies.

2.4.5 Eliminating Parameters via Parallel Moves

see §13.5.2

The program on the right-hand side of Figure 2.14 was obtained by lowering the parameter passing to parallel moves and then to a series of assignments. We use the algorithm from CompCert [RSL08] to lower the parallel moves to a sequence of assignments.

```
1  fun f (x, Y, z, W) =                  {W}
2   if x > 0 then                        {x, Y, z, W}
3    let a = x - 1 in                    {x, Y, z, W}
4    let b = z + z in                    {a, Y, z, W}
5    f (a, Y, b, W)                      {a, b, Y, W}
6   else
7    if x = 0 then                       {x, Y, z, W}
8     let u = Y in                       {Y, z, W}
9     let v = z + u in                   {u, z, W}
10    let w = W in                       {v, W}
11    v + w                              {v, w}
12   else
13    let Z = z in                       {x, Y, z, W}
14    let r = Y in                       {x, Y, Z, W}
15    let c = r - x in                   {x, r, Z, W}
16    let d = -x in                      {c, x, Z, W}
17    f (c, d, Z, W)                     {c, d, Z, W}
18 in
19   let A = w in                        {w}
20   let i = -4 in                       {A}
21   let j = 2 in                        {i, A}
22   let J = j in                        {i, j, A}
23   let k = 1 in                        {i, J, A}
24   f (i, J, k, A)                      {i, J, k, A}
```

Figure 2.13: Figure 2.12 after SSA construction and renaming apart.

2.4.6 Translation to PowerPC Assembly

see §16

The translation to PowerPC assembly proceeds in two steps. LVC translates the IL program to CompCert's Linear language, and afterwards we use two transformations from CompCert's pipeline to lay out the stack and translate to assembly code. CompCert does not support printing Linear code, so we can only show the machine code and assembly file code produced by CompCert. Figure 2.15 shows the machine code (including stack layout) produced by CompCert for the program on the right-hand side of Figure 2.14. Figure 2.16 shows the assembly code produced for the program on the right-hand side of Figure 2.14. Note that code generation is almost 1:1, with the notable exception of reordering of the "in"-clause to appear before the function definition in the code. The left-hand side of Figure 2.16 shows the listing Figure 2.14 where this inversion has been depicted by a stylized "where" notations.

```
fun f (r3, Z, r4, W) =
 if r3 > 0 then
  let r3 = r3 - 1 in
  let r4 = r4 + r4 in
  f (r3, Z, r4, W)
 else
  if r3 = 0 then
   let r3 = Z in
   let r3 = r4 + r3 in
   let r4 = W in
   r3 + r4
  else
   let Y = r4 in
   let r4 = Z in
   let r3 = r4 - r3 in
   let r4 = - r4 in


   f (r3, r4, Y, W)
in
  let W = r3 in
  let r3 = -4 in
  let r4 = 2 in
  let Z = r4 in
  let r4 = 1 in
  f (r3, Z, r4, W)
```

```
fun f () =
 if r3 > 0 then
  let r3 = r3 - 1 in
  let r4 = r4 + r4 in
  f ()
 else
  if r3 = 0 then
   let r3 = Z in
   let r3 = r4 + r3 in
   let r4 = W in
   r3 + r4
  else
   let Y = r4 in
   let r4 = Z in
   let r3 = r4 - r3 in
   let r4 = - r4 in
   let Z = r4 in
   let r4 = Y in
   f ()
in
  let W = r3 in
  let r3 = -4 in
  let r4 = 2 in
  let Z = r4 in
  let r4 = 1 in
  f ()
```

Figure 2.14: Left:Figure 2.13 after register assignment to registers `r3` and `r4`. Note that we assumed the only free variable of the program w maps to the register `r3`. Right: The program on the left after parameters have been eliminated.

Note how in Figure 2.16 each IL instruction directly corresponds to the instruction in the assembly listing on the same line.

2.4.7 Reproduciblity

The results from this section can be reproduced using the source code accompanying this thesis, which is distributed under MIT license and available online on GitHub at https://github.com/sigurdschneider/lvc. The commit hash corresponding to the development we describe is be41194f16495d283fe7bbc982c3393ac554dd5b. To re-

```
1
2    let W = r3 in
3    let r3 = -4 in
4    let r4 = 2 in
5    let Z = r4 in
6    let r4 = 1 in
7    f ()
8  where f () =
9   if r3 > 0 then
10   let r3 = r3 - 1 in
11   let r4 = r4 + r4 in
12   f ()
13  else
14   if r3 = 0 then
15    let r3 = Z in
16    let r3 = r4 + r3 in
17    let r4 = W in
18    r3 + r4
19
20   else
21    let Y = r4 in
22    let r4 = Z in
23    let r3 = r4 - r3 in
24    let r4 = - r4 in
25    let Z = r4 in
26    let r4 = Y in
27    f ()
28
```

```
1  LVC() {
2     stack(100, int) = R3
3     R3 = -4
4     R4 = 2
5     stack(112, int) = R4
6     R4 = 1
7     goto 2
8  2:
9     if (R3 <=s 0) goto 3
10    R3 = R3 + -1
11    R4 = R4 + R4
12    goto 2
13 3:
14    if (R3 !=s 0) goto 4
15    R3 = stack(112, int)
16    R3 = R4 + R3
17    R4 = stack(100, int)
18    R3 = R3 + R4
19    return
20 4:
21    stack(108, int) = R4
22    R4 = stack(112, int)
23    R4 = R4 - R3
24    R4 = 0 - R4
25    stack(112, int) = R4
26    R4 = stack(108, int)
27    goto 2
28 }
```

Figure 2.15: Left: Figure 2.14 in stylized `where` notation instead of `let/in` function definition. Right: Corresponding CompCert Mach language code, in which slot variables have been layed out into the stack by CompCert. Listings are aligned to highlight line-by-line correspondence. Note that we are over-provisioning the size of the stack frame at the moment.

```
  let W = r3 in
  let r3 = -4 in
  let r4 = 2 in
  let K = r4 in
  let r4 = 1 in
  f ()
where f () =
 if r3 > 0 then

  let r3 = r3 - 1 in
  let r4 = r4 + r4 in
  f ()
 else
  if r3 = 0 then

   let r3 = Z in
   let r3 = r4 + r3 in
   let r4 = W in
   r3 + r4


  else
   let Y = r4 in
   let r4 = Z in
   let r3 = r4 - r3 in
   let r4 = - r4 in
   let Z = r4 in
   let r4 = Y in
   f ()
```

```
  .text
  .balign 4
  .globl .LVC
.LVC:
  stwu  1, -128(1)
  mflr  0
  stw 0, 116(1)
  stw 3, 100(1)
  addi  3, 0, -4
  addi  4, 0, 2
  stw 4, 112(1)
  addi  4, 0, 1
  b .L102
.L102:
  cmpwi 0, 3, 0
  bf  1, .L103
  addi  3, 3, -1
  add 4, 4, 4
  b .L102
.L103:
  cmpwi 0, 3, 0
  bf  2, .L104
  lwz 3, 112(1)
  add 3, 4, 3
  lwz 3, 100(1)
  add 3, 3, 4
  addi  1, 1, 128
  blr
.L104:
  stw 4, 108(1)
  lwz 4, 112(1)
  subfc 3, 3, 4
  subfic  4, 4, 0
  stw 4, 112(1)
  lwz 4, 108(1)
  b .L102
.L101:
  .type .LVC, @function
  .size .LVC, . - .LVC
  .text
```

Figure 2.16: Left: Figure 2.14 in stylized `where` notation instead of `let/in` function definition. Right: Assembly produced by CompCert for the program on the left (debug information has been ommitted). Listings are aligned to highlight line-by-line correspondence.

produce the register allocation example from this section, first download and build LVC as instructed in the accompanying `README.md` file. Then build the LVC command line client with `make extraction`, change to the compiler directory with `cd compiler`, and run the compiler on the example with by issuing the command `./lvcc.byte examples/overview-alloc.il -v`. Compilation results are stored in files named `overview-alloc.il.`*`phase`* where *`phase`* hints at the compilation phase to which they correspond. The example file contains annotations that specify spill slot coalescing information for the register allocation phase, which are translation validated by LVC. Additionally, we have included a spilling algorithm in LVC that produces an optimal spilling for the example. LVC translation validates the results of this spilling algorithm.

3 Preliminaries

In this section we give definitions of basic structures and notions we use in this thesis. We recommend this section even to experienced readers, as our definitions are non-standard in subtle ways to better accommodate Coq's extraction feature.

3.1 Type Theory

We work in Coq's type theory, which is based on the calculus of inductive constructions [CH88]. We use **P** to denote the type of propositions, and **T** to denote the type of types, leaving universe levels implicit as usual.

Most of our work is constructive, and does not use axioms. Via the Curry-Howard isomorphis, that is, propositions as types, every proof corresponds to a program. Coq's extraction feature [Let08] provides a mechanism which allows to obtain executable OCaml code from proofs and definitions in the type theory. LVC's main components are extracted from Coq definitions.

Occasionally, we may use axioms such as functional extensionality, proof irrelevance, and uniqueness of identity proofs. All uses of axioms are explicitly marked.

3.2 Decidability

We extract decision procedures from many decidability results to OCaml. Our notion of decidability is based on a type-level disjunction $+$, not on the proposition-level disjunction $\vee$, because extraction removes all propositions. Computable

Definition 3.1 ▪ **Deciability**
Let X, Y be types. A predicate $P : X \to \mathbf{P}$ is decidable if $\forall x,\ Px + \neg Px$ is provable. Similarly, a relation $R : X \to Y \to \mathbf{P}$ is decidable if $\forall xy,\ Rxy + \neg Rxy$ is provable.

If a decidability result is constructive, its proof can be turned into an executable OCaml procedure via Coq's extraction feature.

3.3 Option Types and Lists

We use several option types, and a type of lists.

Definition 3.2 If T is a type, then $T_\bot$ is the option type for T, that is a type that has all elements from T and an additional, distinct element $\bot$. We also use the analogously defined option type $T^\top$. Naturally, the two option types can be combined to obtain a type with two additional elements: $T_\bot^\top$.

Definition 3.3 If T is a type, then $\overline{T}$ or $\mathsf{list}\, T$ is the type of lists of elements of type T. We use the notation $\overline{x}$ to denote a list over elements denoted by $x_1, \ldots, x_n$ where n is the length of the list $|\overline{x}|$.

3.4 Equivalence Relation

Definition 3.4 Let X be a type. A relation $\equiv\colon X \to X \to \mathbf{P}$ is an equivalence relation if the following holds:

1. $x \equiv x$ — reflexivity
2. $x \equiv y \to y \equiv x$ — symmetry
3. $x \equiv y \to y \equiv x \to x \equiv y$ — transitivity

3.5 Strict Order

Definition 3.5 Let X be a type. A relation $\sqsubset\colon X \to X \to \mathbf{P}$ is a total strict order if it is transitive and the following holds:

1. $x \not\sqsubset x$ — irreflexivity
2. $x \sqsubset y \to y \not\sqsubset x$ — asymmetry

Note that asymmetry follows from irreflexivity and transitivity.

3.6 Ordered Types

Definition 3.6 An ordered type is a tuple $(X, \sqsubset, \equiv)$ where X is a type and $\sqsubset$ is a decidable strict order and $\equiv$ is a decidable equivalence relation and $\sqsubset$ is total:

1. $x \sqsubset y \vee x \equiv y \vee y \sqsubset x$ — totality

Note that the trichotomy required by totality is computational, because all involved relations are decidable.

3.7 Finite Sets

We use the type class based finite set library developed by Lescuyer [Les11]. The library specifies a interface for implementations of finite sets based on type-classes. Given an ordered type A, an implementation of finite sets over A is a type set A such that the following operations are defined and behave as usual:

- $\in$: $A \to \mathsf{set}\ A \to \mathbf{P}$ is a decidable relation
- $\subseteq, \equiv$: $\mathsf{set}\ A \to \mathsf{set}\ A \to \mathbf{P}$ are a decidable relations
- $\emptyset$: $\mathsf{set}\ A$ is the empty set
- $\{\cdot;\cdot\} : A \to \mathsf{set}\ A \to \mathsf{set}\ A$ is adjunction
- $\{\cdot\} : A \to \mathsf{set}\ A \to \mathsf{set}\ A$ is singleton formation
- $- : \mathsf{set}\ A \to A \to \mathsf{set}\ A$ removes a single element
- $\cup, \cap, \setminus : \mathsf{set}\ A \to \mathsf{set}\ A \to \mathsf{set}\ A$ are union, intersection, and difference
- $\mathsf{fold} : \forall B : \mathbf{T}, (A \to B \to B) \to \mathsf{set}\ A \to B \to B$ is a recursor for sets
- $\forall, \exists : (A \to \mathbb{B}) \to \mathsf{set}\ A \to \mathbb{B}$ are quantifiers for sets
- $\mathsf{filter} : (A \to \mathbb{B}) \to \mathsf{set}\ A \to \mathsf{set}\ A$ computes the subset satisfying a boolean predicate
- $\mathsf{partition} : (A \to \mathbb{B}) \to \mathsf{set}\ A \to \mathsf{set}\ A * \mathsf{set}\ A$ partitions according to a boolean predicate
- $|\cdot| : \mathsf{set}\ A \to \mathbb{N}$ yields the cardinality of a set
- $\mathsf{elements} : \mathsf{set}\ A \to \mathsf{list}\ A$ yields the list of elements without duplicates
- $\mathsf{choose} : \mathsf{set}\ A \to A_\perp$ yields an element of the set if possible
- $\mathsf{min} : \mathsf{set}\ A \to A_\perp$ yields the minimal element if possible
- $\mathsf{max} : \mathsf{set}\ A \to A_\perp$ yields the maximal element if possible
- $(\mathsf{set}\ A, \subset, \equiv)$ is an ordered type

We omit the formal specification for the operations; the interested reader can find them in the Coq source online.

3.8 Agreement of Functions

Definition 3.7 For functions $f, g : A \to B$ We write $f =_X g$ if f and g agree on all variables in X.

3.9 Preorders

In LVC, must use preorders instead of partial orders, because often $\equiv$ is not equality, but a coarser equivalence relation. This situation arises, for example, because we use a set library in Coq that uses an efficient implementation at the cost of not providing extensional sets.

Definition 3.8 ▪ Preorder
A preorder is a tuple $(X, \sqsubseteq, \equiv)$ where X is a type and $\sqsubseteq, \equiv$ are decidable relations of type $X \to X \to \mathbf{P}$ such that the following holds:

1. $\equiv$ is an equivalence relation
2. $x \equiv y \to x \sqsubseteq y$ — reflexivity
3. $x \sqsubseteq y \to y \sqsubseteq x \to x \equiv y$ — antisymmetry

Our definition of a preorder allows the relation $\equiv$ to be explicitly provided, although such a relation could be defined from $\sqsubseteq$. This allows the equality check to provide its own decision procedure, which may make the extracted decision procedure for equivalence more efficient than checking whether $\sqsubseteq$ holds twice.

Lemma 3.9 Let $(X, \sqsubseteq, \equiv)$ be a preorder. Then $x \equiv y \leftrightarrow x \sqsubseteq y \wedge y \sqsubseteq x$.

Definition 3.10 Let $(X, \sqsubseteq)$ be a preorder. We define strict order $\sqsubset$ on elements $x, y : X$ of a preorder as

$$x \sqsubset y \;:\leftrightarrow\; x \sqsubseteq y \wedge y \not\sqsubseteq x$$

Lemma 3.11 Let $(X, \sqsubseteq, \equiv)$ be a preorder. Then $\sqsubset$ is a decidable strict order.

Lemma 3.12 $(X, \sqsubseteq, \equiv)$ is a preorder if $(X, \sqsubset, \equiv)$ is an ordered type.

The reverse direction of Lemma 3.12 does not hold, because $\sqsubseteq$ cannot in general be turned into a strict order.

3.9.1 Canonical preorders

In this section, we will establish **canonical** preorders on several types. Whenever we use a preorder on a type without explicitly specifying the relation $\sqsubseteq$, we mean the canonical preorder on that type. In particular, when we say X is a preordered type and subsequently use $\sqsubseteq$, then we are referring to the canonical preorder on X. We show formation lemmas that provide canonical instances of preorders for structures such as pairs, lists, etc.

Lemma 3.13 Let X, Y be preordered types. Then the canonical preorder on $X \times Y$ can be obtained by component-wise lifting of $\sqsubseteq$.

Lemma 3.14 Let X be a preordered type. The canonical preorder on $\mathsf{list}\, X$ is obtained by component-wise relating lists of the same length.

Lemma 3.15 The canonical preorder on $\mathbb{B}$ is obtained by defining $\sqsubseteq$ to satisfy $b \sqsubseteq b$ for all booleans b and $\mathbf{false} \sqsubseteq \mathbf{true}$.

Lemma 3.16 Let X be a preordered type. Then there are canonical preorders for the option types $X_\bot$ and $X^\top$. The relation $\sqsubseteq$ then additionally satisfies $\bot \sqsubseteq x$, or $x \sqsubseteq \top$, respectively.

Lemma 3.17 Let Y be a preordered type. Then there is a canonical preorders for the every function type $X \to Y$ with finite domain, and the relation $\sqsubseteq$ is defined by point-wise lifting.

Definition 3.18 ▪ Monotonicity
Let X, Y be preordered types and $f : X \to Y$ be a function. We say f is monotone if $x \sqsubseteq x' \to fx \sqsubseteq fx'$.

3.10 Lattices

For many components of LVC, a full lattice is not required, but weaker structures suffice. We define several structures to allow a more fine-grained management of requirements. Our main tool will be a join semi-lattice.

Definition 3.19 A join semi-lattice is a tuple $(X, \sqcup)$ where X is a preordered type and $\sqcup : X \to X \to X$ is a join operation such that

1. $x \sqcup y \equiv y \sqcup x$ — symmetric
2. $x \sqcup (y \sqcup z) \equiv (x \sqcup y) \sqcup z$ — associative
3. $x \sqsubseteq x \sqcup y$ — expansive

4 $x \sqsubseteq y \to x \sqcup y \sqsubseteq y$ idempotent

Condition (3) lower-bounds the join of two values, while condition (4) upper-bounds the join.

Lattices can be defined algebraically or order-theoretically; we mix both ways in Definition 3.19. The reason is that we need efficient extraction for the join operation and the order relation. This requirement precludes that we define one from the other, as usual. However, from the requirements in Definition 3.19 it follows that the usual correspondence between join and order holds:

Lemma 3.20 Let X be a join semi-lattice. Then $x \sqsubseteq y \leftrightarrow x \sqcup y \equiv y$.

Proof. From conditions (3) and (4) with antisymmetry. ■

Lemma 3.20 verifies that our definition behaves as usual.

Definition 3.21 A preorder X is lower bounded if there is an element $\bot : X$ such that $\bot \sqsubseteq x$ for all $x : X$, and upper bounded if there is an element $\top : X$ such that $x \sqsubseteq \top$ for all $x : X$.

3.11 Internally Deterministic Reduction Systems

We define a notion of reduction on a restricted form of LTS which we use to abstract from concrete reduction behavior of the languages we introduce later. We focus on the way the program interacts with its environment via external calls, termination behavior, and possibly by returning a result value. We abstract this behavior with internally deterministic reduction systems (IDRS), that we previously introduced [SSH15].

IDRS a labeled transition systems (LTS) with additional requirements.

Definition 3.22 ■ Event Type
A type $\mathcal{E}$ is an event type, if it contains at least one distinct element τ, which designates the silent event and has decidable equality. We call the elements of $\mathcal{E}$ *events*. By convention, the meta variable ϕ ranges over all events, and ψ ranges over all events except the silent event τ.

Definition 3.23 A labeled transition system (LTS) is a tuple $(\Sigma, \mathcal{E}, \longrightarrow)$ where Σ is a type of states, $\mathcal{E}$ is an event type, and $\longrightarrow$ is a labeled transition relation of type $\Sigma \to \mathcal{E} \to \Sigma \to \mathbf{P}$. We use the notation $\sigma \xrightarrow{\phi} \sigma'$ for transitions.

We omit writing τ-action from the reduction relation and write $\xrightarrow{\tau}$ just as $\longrightarrow$.

$$\textsc{ReflTrans1}\ \frac{}{\sigma \xrightarrow{\mathsf{nil}}{}^{*}\ \sigma} \qquad \textsc{ReflTrans2}\ \frac{\sigma \xrightarrow{\phi} \sigma' \qquad \sigma' \xrightarrow{l}{}^{*}\ \sigma''}{\sigma \xrightarrow{\mathsf{nosilent}\,\phi\,l}{}^{*}\ \sigma''}$$

$$\textsc{Trans}\ \frac{\sigma \xrightarrow{\phi} \sigma' \qquad \sigma' \xrightarrow{l}{}^{*}\ \sigma''}{\sigma \xrightarrow{\mathsf{nosilent}\,\phi\,l}{}^{+}\ \sigma''}$$

$$\mathsf{nosilent}\,\tau\,l = l$$
$$\mathsf{nosilent}\,\psi\,l = \psi, l$$

Figure 3.1: Transitive and reflexive-transitive closure of $\longrightarrow$.

Definition 3.24 ▪ Transitive Closure, Reflexive-Transitive Closure of $\longrightarrow$
We define the transitive closure $\longrightarrow^{+}$ and the reflexive-transitive closure $\longrightarrow^{*}$ of the step relation according to the rules in Figure 3.1.

We omit the empty event list nil when we write $\longrightarrow^{+}$ and $\longrightarrow^{*}$.

Definition 3.25 ▪ Termination
We write $\sigma \not\longrightarrow$ if σ has no $\longrightarrow$-successor.

Definition 3.26 A $(\mathcal{E}, \tau)$-*reduction system* (RS) is a tuple $(\Sigma, \longrightarrow, \mathsf{res})$ such that

1. $(\Sigma, \mathcal{E}, \longrightarrow)$ is a LTS
2. $\mathsf{res} : \Sigma \to \mathbb{V}_{\bot}$
3. $\mathsf{res}\,\sigma = v \Rightarrow \sigma \not\longrightarrow$
4. $\tau \in \mathcal{E}$

An *internally deterministic* reduction system (IDRS) additionally satisfies

5. $\sigma \xrightarrow{\phi} \sigma_1 \wedge \sigma \xrightarrow{\phi} \sigma_2 \Rightarrow \sigma_1 = \sigma_2$ — action-deterministic
6. $\sigma \xrightarrow{\phi} \sigma_1 \wedge \sigma \xrightarrow{\tau} \sigma_2 \Rightarrow \phi = \tau$ — τ-deterministic

Definition 3.27 ▪ Stuck and Final Configurations
We say σ is *stuck* if $\sigma \not\longrightarrow$ and $\mathsf{res}\,\sigma = \bot$. We say σ is *final* if $\sigma \not\longrightarrow$ and $\mathsf{res}\,\sigma \neq \bot$. We write $\sigma \Downarrow w$ if there is σ' such that $\sigma \longrightarrow^{*} \sigma'$ and $\sigma' \not\longrightarrow$ and $\mathsf{res}\,\sigma' = w$ for $w : \mathbb{V}_{\bot}$.

$$\textsc{Div}\ \frac{\sigma \xrightarrow{\tau} \sigma' \qquad \sigma' \Uparrow}{\sigma \Uparrow}$$

Figure 3.2: Rule defining silent divergence of an IDRS state.

Note that we use meta variable in a subtle way: $\sigma \Downarrow w$ just means that σ reduces to a terminal configuration, but $\sigma \Downarrow v$ also means that the terminal configuration is final, because v only ranges over $\mathbb{V}$ and does not include $\bot$.

3.11.1 Silent Divergence

Silent divergence of a state in an IDRS can be formally defined using the coinductive definition in Figure 3.2.

Lemma 3.28 $\sigma \Uparrow$ if there is no σ' such that $\sigma \longrightarrow^* \sigma'$ and either $\sigma' \not\longrightarrow$ or σ' is ready.

Proof. By coinduction. The premises ensure that σ silently reduces, and are themselves stable under silent reduction. ■

Theorem 3.29 For every configuration $\sigma \in \Sigma$ it is propositionally the case that either $\sigma \Uparrow$ or $\sigma \Downarrow w$ with $w \in \mathbb{V}_\bot$ or there is σ' such that $\sigma \longrightarrow^* \sigma'$ and σ' is ready.

Proof. Using excluded middle and Lemma 3.28. ■

Theorem 3.30 For every configuration $\sigma \in \Sigma$ it is informatively the case that either $\sigma \Uparrow$ or $\sigma \Downarrow w$ with $w \in \mathbb{V}_\bot$ or there is σ' such that $\sigma \longrightarrow^* \sigma'$ and σ' is ready.

Proof. Using informed excluded middle, indefinite description, and Lemma 3.28. ■

4 The Intermediate Language IL

In this section, we describe the language IL and its two semantic interpretations IL/F and IL/I.

4.1 Values, Variables, and Expressions

We use $\mathbb{V}$ to denote the type of integer values from CompCert [Ler09b], which we use as type of values for IL. We define a function $\beta : \mathbb{V} \to \mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ to simplify the semantic rule for the conditional. By convention, v ranges over $\mathbb{V}$. We use a countably-infinite alphabet $\mathcal{V}$ for names x, y, z of values, which we call *variables*.

We define a type Exp of expressions with unary and binary expressions.

$$
\begin{array}{rll}
o_1 ::= ? & & \text{conversion to bool} \\
\quad | \; ! & & \text{negation} \\
o_2 ::= & + \mid - \mid \cdot \mid : \mid = \mid \leq & \\
e ::= & c \mid x \mid o_1\, e \mid e\, o_2\, e &
\end{array}
$$

By convention, e ranges over Exp. Expressions are pure, their evaluation is deterministic and may fail. Expression evaluation is a recursively defined function

$$[\![\cdot]\!] : \mathsf{Exp} \to (\mathcal{V} \to \mathbb{V}_\perp) \to \mathbb{V}_\perp$$

that relies on CompCert's definition of the unary and binary operators. Environments are of type $\mathcal{V} \to \mathbb{V}_\perp$ to track uninitialized variables, and are preordered by $\sqsubseteq$, which is the pointwise lifting of the relation defined by the two equations $\perp \sqsubseteq w$ and $w \sqsubseteq w$, where $w \in \mathbb{V}_\perp$.

Lemma 4.1 Expression evaluation is monotone: $V \sqsubseteq V' \to [\![e]\!]\, V \sqsubseteq [\![e]\!]\, V'$.

We define a function $\mathsf{fv} : \mathsf{Exp} \to \mathsf{set}\, \mathcal{V}$ that determines the free variable of an expression as usual.

Lemma 4.2 If environments V, V' agree on $\mathsf{fv}\,(e)$ then $[\![e]\!]\, V = [\![e]\!]\, V'$.

$$
\begin{array}{rll}
\eta & ::= e \mid \alpha(\overline{e}) & \text{extended expression} \\
\mathsf{Term} \ni s, t & ::= \texttt{let}\ x = \eta\ \texttt{in}\ s & \text{variable binding} \\
& \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t & \text{conditional} \\
& \mid e & \text{expression} \\
& \mid \texttt{fun}\ \overline{f\ \overline{x} = s}\ \texttt{in}\ t & \text{function definition} \\
& \mid f\ \overline{e} & \text{application}
\end{array}
$$

Figure 4.1: Syntax of IL. We use both *term* and *program* for the syntactic category Term. We use *statement* to reference to the top-level constructor of a program.

We lift $[\![\cdot]\!]$ pointwise to lists of expressions in a strict fashion: $[\![\overline{e}]\!]$ yields a list of values if none of the expressions in $\overline{e}$ failed to evaluate, and $\bot$ otherwise.

We sometimes omit the side condition $[\![\overline{e}]\!]\ V \neq \bot$ in the presentation if $[\![\overline{e}]\!]\ V$ is used in a place where type $\mathbb{V}$ is required. For example, we write $\beta([\![\overline{e}]\!]\ V) = \mathbf{true}$ instead of $\exists v : \mathbb{V}, [\![\overline{e}]\!]\ V = v \wedge \beta v = \mathbf{true}$.

4.2 Syntax

IL is a first-order language with a tail-call restriction, mutual recursion, and external events. IL syntactically enforces a first-order discipline by using a separate alphabet $\mathcal{F}$ for function names f, g, h. Variables are lexically scoped binders, and a function definition creates a closure that captures variables.

IL uses a third alphabet $\mathcal{A}$ for names α which we call *actions*. The term $\texttt{let}\ x = \alpha(\overline{e})\ \texttt{in}\ \ldots$ behaves like a system call α with argument list $\overline{e}$ that non-deterministically returns a value.

IL allows mutually recursive function definitions. The syntax of IL is given in Figure 4.1.

4.3 Semantics

The semantics of IL/F is given as small-step reduction relation $\longrightarrow_{\mathrm{F}}$ in Figure 4.2. Note that the tail-call restriction ensures that no call stack is required. The reduction relation $\longrightarrow_{\mathrm{F}}$ operates on **configurations** of the form $(L, V, s)_{\mathrm{F}}$ where s is the IL/F term to be evaluated. When clear from context, we may omit the subscript on configurations. We occasionally write a configuration using the notation $L \mid V \mid s$ to have the comma separator available for other notations. The semantics does not

rely on substitution, but uses an environment $V : \mathcal{V} \to \mathbb{V}_\perp$ for variable definitions and a context L of function definitions. Transitions in $\longrightarrow_{\mathrm{F}}$ are labeled with *events* ϕ. By convention, ψ ranges over events different from τ.

$$\mathcal{E} \ni \phi ::= \tau \mid v = \alpha(\overline{v})$$

The silent event is denoted by τ, and by convention, we omit it from the small-step relation and just write $\longrightarrow_{\mathrm{F}}$ for $\xrightarrow{\tau}_{\mathrm{F}}$ by convention.

A **context** is a list of lists of named definitions. For example, the context $K = [f_1 : a_1, f_2 : a_2]; [g_1 : b_1]$ consists of three definitions in two lists. f_1 and f_2 are defined mutually recursively. We define a function *dom* that yields the domain of a context as a list, e.g. $\mathit{dom}\, K = [f_1, f_2]; [g_1]$. A definition in a context may refer to previous definitions and definitions in its list, i.e. in the example context K, the function f_2 may refer to f_1, but g_1 cannot refer to any f_i. Notationally, we use contexts like functions, and abstract from the nests list-of-lists structure: To access the first element with name f, we write L_f and we have $L_f = \perp$ if no such element exists. We write L^{-f} for the context obtained from L by dropping all lists before the first list that contains f. We write ; for context concatenation and $\varnothing$ for the empty context.

A **closure** is a tuple $(V, \overline{x}, s) \in \mathcal{C}$ consisting of an environment V, a parameter list $\overline{x}$, and a function body s. Since a function f in a context can only refer to previously defined functions and functions in its own group, the first-order restriction allows the closures to be non-recursive: function closures do not need to close under functions. An application $f\overline{e}$ causes the function context L to rewind to L^{-f}, i.e. up to the group with the definition of f (rule App). In contrast to higher-order formulations, we do not define closures mutually recursively with the values of the language.

A **system call** $\texttt{let}\ x = \alpha\, \overline{e}\ \texttt{in}\ s$ invokes a function α of the system, which is not assumed to be deterministic. This reflects in the rule Extern, which does not restrict the result value of the system call other than requiring that it is a value. The transition records the system call name α, the argument values $\overline{v}$ and the result value v' in the event $v' = \alpha(\overline{v})$.

4.4 Imperative Interpretation of IL: IL/I

We are interested in a translation to IL from an imperative language that does not require function closures at run-time. To investigate this translation, we introduce a second semantic interpretation for IL which we call IL/I. IL/I is an imperative language, where variables are interpreted as assignables [Har13]. Function application becomes a *goto*, and parameter passing is a parallel assignment of arguments to the parameters, which possibly overwrites variables. Closures are replaced by blocks

IL-Let-Op

$$\frac{[\![e]\!]\, V = v}{L \mid V \mid \texttt{let}\ x = e\ \texttt{in}\ s \longrightarrow L \mid V[x \mapsto v] \mid s}$$

IL-Let-Call

$$\frac{[\![e]\!]\, V = v \qquad \beta(v) = b}{L \mid V \mid \texttt{if}\ e\ \texttt{then}\ s_{\mathbf{true}}\ \texttt{else}\ s_{\mathbf{false}} \longrightarrow L \mid V \mid s_i}$$

IL-Cond

$$\frac{v' \in \mathbb{V} \qquad [\![\overline{e}]\!]\, V = \overline{v}}{L \mid V \mid \texttt{let}\ x = \alpha(\overline{e})\ \texttt{in}\ s \ \longrightarrow_{v' = \alpha(\overline{v})}\ L \mid V[x \mapsto v'] \mid s}$$

ILF-Fun

$$\frac{}{L \mid V \mid \texttt{fun}\ \overline{f\ \overline{x}\ =\ s}\ \texttt{in}\ t \ \longrightarrow_{\mathrm{F}}\ (\!|\overline{f\ \overline{x}\ =\ s}|\!)^{V}_{\mathrm{F}}; L \mid V \mid t}$$

ILF-App

$$\frac{[\![\overline{e}]\!]\, V = \overline{v} \qquad L_f = (V', \overline{x}, s)}{L \mid V \mid f\ \overline{e} \ \longrightarrow_{\mathrm{F}}\ L^{-f} \mid V'[\overline{x} \mapsto \overline{v}] \mid s}$$

$$(\!|\overline{f\ \overline{x}\ =\ s}|\!)^{V}_{\mathrm{F}} = [\overline{f : (V, \overline{x}, s)}]$$

Figure 4.2: Definition of IL/F reduction relation $\longrightarrow_{\mathrm{F}}$. The relation has no subscript in the rules IL-Let-Op and IL-Let-Call and IL-Cond because those rules are also part of IL/I semantics.

$(\overline{x}, s) \in \mathcal{B}$ and blocks do not contain variable environments. Consequently, a called function can see all previous updates to variables. For example, the following two programs each return 5 in IL/I, but evaluate to 7 in IL:

```
let x = 7 in
fun f () = x in
let x = 5 in f ()
```

```
let x = 7 in
fun f () = x in
fun g x = f() in
let y = 5 in g y
```

4.4.1 Small-Step Semantics of IL/I

To obtain the IL/I small-step relation $\longrightarrow_{\mathrm{I}}$, we define

$$(\!|\overline{(\overline{x}, s)}|\!)_{\mathrm{I}} = (\overline{x}_1, s_1, 1), \ldots, (\overline{x}_n, s_n, n)$$

ILI-Fun

$$\frac{}{L \mid V \mid \texttt{fun}\ \overline{f\ \overline{x} = s}\ \texttt{in}\ t \longrightarrow_{\mathrm{I}} (\!|\overline{f\ \overline{x} = s}|\!)_{\mathrm{I}}; L \mid V \mid t}$$

ILI-App

$$\frac{[\![\overline{e}]\!]\, V = \overline{v} \qquad L_f = (\overline{x}, s)}{L \mid V \mid f\ \overline{e} \longrightarrow_{\mathrm{I}} L^{-f} \mid V[\overline{x} \mapsto \overline{v}] \mid s}$$

Figure 4.3: Semantics of IL/I. The relation $\longrightarrow_{\mathrm{I}}$ is defined by the rules IL-Let-Op and IL-Let-Call and IL-Cond from Figure 4.2 and the rules ILI-Fun and ILI-App.

The reduction relation $\longrightarrow_{\mathrm{I}}$ is defined by the rules IL-Let-Op and IL-Let-Call and IL-Cond from Figure 4.2 and the rules ILI-Fun and ILI-App from Figure 4.3.

4.5 α-Equivalence

The semantics of IL/F respects α-equivalence, i.e. consistent renaming of variables does not change the semantics. We say an IL program is renamed apart if every variable name has at most one defining occurrence. Defining occurrences are occurrences on the left-hand side of a let, or occurrences as function parameter. Due to α-equivalence, every IL program can be **renamed apart** without changing it's IL/F semantics. The same does not hold for IL/I programs in general: they cannot be renamed apart without further ado. In §10 we will see how to rename apart an IL/I program.

In the Coq development, we formally define α-equivalence, together with functions that efficiently rename apart an IL program. We specify formally what it means for a program to be renamed apart, and formally show that α-equivalence respects IL/F semantics.

Theorem 4.3 α-equivalence is sound with respect to IL/F semantics.

Example 4.4 The program below shows that it is not easily possible to rename apart a program and preserve IL/I semantics: Since g reads x, both neither definition of x can be renamed without changing IL/I semantics. In particular, renaming apart the program program below changes IL/I semantics, although the resulting program is still α-equivalent.

```
1 fun g() = x in
2 if e then let x = 1 in g()
3      else let x = 2 in g()
```

For space reasons, however, we not report in detail on the α-equivalence part of the development in this thesis.

4.6 IL as IDRS

The semantics of IL/I and IL/F each forms an IDRS: We define res such that $\mathsf{res}\,\sigma = v$ if σ is of the form (L, V, e) and $[\![e]\!]\,V = v$. Otherwise, $\mathsf{res}\,\sigma = \bot$.

4.7 Notational Conventions

We adopt several notational conventions to ease notational burden. One notational convention arises from the fact that information in algorithms needs to be reorganized frequently. A particularily prominent example is reorganizing a value of type $\mathsf{list}\,(A \times B)$ into a value of type $\mathsf{list}\,A \times \mathsf{list}\,B$, and vice versa, if the two lists are of the same length. This reorganization is accomplished by a function zip.

Since reorganizations of this kind arise frequently, and we do not want to clutter our definitions with unending applications of zip, its friends, and their inverses, we do not write applications of zip but leave them implicit.

Similarly, we often lift functions in a point-wise manner to operate on lists: Given a function $A \to B$ it is easy to obtain a function $\mathsf{list}\,A \to \mathsf{list}\,B$ via map, and similar liftings exist for functions of different arity. The Coq proofs have to deal excessively with such transformations, and particularily the associated inversions. On the mathematical level, we do not distinguish between a function and its point-wise lifting, or put another way, we do not explicitly insert applications of map, but omit them.

We are aware that these notational conventions require some form of elaboration on the reader's part. However, all definitions and proofs are backed up by a formal Coq development, which has all the details. The formal development is available online, and we hyperlink each definition to its Coq formalization, and makes it easy to go back and forth between the mathematical definitions in this thesis to the formal Coq definitions in the development. Because of this, we think that our notational conventions improve the presentation by removing unnecessary detail.

4.8 Program Points and Annotations

As one concern of this thesis is the verification of compiler optimizations, the need to represent program analysis information arises frequently. Mathematically, program analysis information for a program is a structure isomorphic to the abstract syntax tree of the program with labels conveying program analysis information. In this metaphor, we use *program point* to refer to a node in the abstract syntax tree. Clearly,

$$
\begin{array}{rll}
 & d \in D & \\
\mathsf{Ann}\, D \ni a, b ::= & d & \text{for applications and expressions} \\
 & \mid d \cdot a & \text{for lets} \\
 & \mid d \cdot a, b & \text{for conditionals} \\
 & \mid d \cdot \overline{a}, b & \text{for function definitions}
\end{array}
$$

Figure 4.4: Syntax of IL annotation trees.

the set of program points of an IL program is isomorphic to its set of subterms. Program analysis information associates information with each program point. We now discuss how we realize this association in the Coq development, and how we handle the mathematical presentation of this association in this thesis.

4.8.1 Annotations

In the implementation in the Coq development, we associate program analysis information of some type D with an IL program s by using an *annotation tree* that is isomorphic to the AST of s and is labeled with values of type D at each node in the tree. Annotation trees have type $\mathsf{Ann}\, D$ for a type of analysis information D, and are inductively defined according to the grammar given in Figure 4.4.

The most important operation on IL annotation trees $a : \mathsf{Ann}\, D$ is the projection of the top-level annotation: $[a]$. Sometimes, we need to set the top-level annotation of an annotation tree: $\mathsf{setTopAnn}\, a\, d$. There is also a function that produces an annotation tree isomorphic to a given program s, and uses a value $d : D$ as annotation at every node: $\mathsf{initAnn}\, s\, d$.

4.8.2 Notational Conventions for Annotations

Notationally, however, making the annotations trees explicit introduces clutter into the definitions of transformations. Suppose we wanted to define a transformation t that depends on the program and annotation of type $\mathbb{B}$. The defining equation for let-bindings might look as follows:

$$\mathsf{t}\,(\texttt{let}\ x = e\ \texttt{in}\ s)\,(d \cdot a) = \mathsf{if}\ d \wedge [a]\ \mathsf{then}\ \texttt{let}\ x = e\ \texttt{in}\ (\mathsf{t}\, s\, a)\ \mathsf{else}\ \mathsf{t}\, s\, a$$

Note that the sans-serif font if represents a meta-level conditional, while the font `if` represents an element of IL syntax. The fictive translation in the equation above depends on the annotation of the let-binding d and annotation $[s]$, which belongs

to s. It is easy to see that matching on the annotations and passing the right sub-annotations to recursive calls quickly clutters the definition, while providing little insight. Fortunately, many transformations operate locally, that is, they only depend on the analysis information of the term they operate on. In such a setting, we notationally pretend that analysis information is organized as additional labels in the AST and display it inline. For example, we write the equation from above as:

$$\mathsf{t}\,(\texttt{let}\ x = e\ \texttt{in}\ (s : d_1) : d_2) = \mathsf{if}\ d_1 \wedge d_2\ \mathsf{then}\ \texttt{let}\ x = e\ \texttt{in}\ (\mathsf{t}\ s)\ \mathsf{else}\ \mathsf{t}\ s$$

In this equation we use a notation involving : to indicate that s is annotated with the boolean value d_1, and the whole argument term is annotated with d_2.

There is a technical foundation for this notational trick, namely the isomorphism between two trees of the same shape of types tree A and tree B, and one tree of type tree$(A \times B)$. We sometimes pretend that the analysis information is organized in one tree, in particular, that the analysis information is available as an additional label at each node of the AST. We might choose notations different from : to associate annotations with subterms, for example, we write $s\,\{a\}$ to indicate that the information associated with s is a. In other places we will have to match on annotation trees explicitly, or we have to produce them when we transform program annotations. Since both representations are isomorphic, take the freedom to freely switch between the two representations.

In the Coq development, analysis information is always kept in a second tree. In our opinion, this has the following advantages over an approach where annotations are embedded in the syntax of the program:

- We do not have to show that program semantics is oblivious to annotations.
- We can work with several annotations of types $D_1, D_2, \ldots$ without having to encode them in tupels of type $D_1 \times D_2 \times \ldots$, but instead just use several annotation trees.

5 Trace-based Semantic Equivalence

At the heart of every verification project, there is a formal definition of what correctness means. There are two important aspects of the definition of correctness. First, it must match the high-level, intuitive meaning of correctness. Second, it must be easy to work with in correctness proofs. This often means that different characterizations of correctness are useful. In this chapter we focus on a trace-based notion of equivalence. Trace-based characterizations are not only intuitive, but also explicate that properties defined with respect to the set of traces a system produces are invariant under a semantic relation.

In compiler verification, correctness must specify under which conditions one program is a valid implementation of another. For this purpose, a semantic relation called *implementation relation* is specified. The question of semantic equivalence is of lower priority, but often simpler to specify. Usually, the implementation relation is asymmetric and contains the semantic equivalence relation. We define the following trace-based semantic relations between states of (possibly different) IDRS (see §3.11):

$$\overset{\text{TR}}{=} \subseteq \overset{\text{TR}}{\searrow} \subseteq \overset{\text{TR}}{\searrow_1}$$

The relation $\overset{\text{TR}}{=}$ is simply trace equivalence. The relations $\overset{\text{TR}}{\searrow}$ and $\overset{\text{TR}}{\searrow_1}$ are implementation relations. $\overset{\text{TR}}{\searrow}$ allows no change in the implementation of the system calls, while $\overset{\text{TR}}{\searrow_1}$ allows non-determinism in the system calls to be resolved, but requires that at least one valid behavior for each system call remains. We define the relations in this section, but postpone the proof of the inclusion chain to Theorem 6.24.

We do not define the relations on the semantics of IL, but use the IDRS abstraction we introduced in §3.11, of which IL's semantics are instances (§4.6).

5.1 Trace Equivalence

A reasonable notion of equivalence for two configurations in two possibly different $(\mathcal{E}, \tau)$-IDRS is to abstact from internal behavior and require that the two configurations can interact in the same way with their environment. For IDRS, all relevant

Figure 5.1: Two processes with the same traces that are not bisimilar: The process on the right must decide before doing the τ-step whether to do α or β, while the process on the left can delay the decision until after the τ-step.

interactions are recorded in event traces, hence trace equivalence presents itself als a natural notion of equivalence. Such a trace equivalence is the established and accepted notion of equivalence in verified compilers [Ler09b] for deterministic languages. Trace equivalence famously breaks down if the two processes depicted in Figure 5.1 must be distunguished. The right process is, however, ruled out by the IDRS requirements for determinism.

In this section, we define two characterizations of trace equivalence and show them equivalent. The first is based on partial traces, which are finite objects. The second is based on maximal, possibly infinite traces.

5.1.1 Partial Trace Equivalence

A partial trace π adheres to the following grammar:

$$\Pi \ni \pi ::= \epsilon \mid v \mid \bot \mid \psi\pi$$

We inductively define the relation $\triangleright \subseteq \Sigma \times \Pi$ such that $\sigma \triangleright \pi$ whenever σ produces the partial trace π. By definition, $\triangleright$ erases τ transitions from the partial trace. This reflects the intention that we do not want to distinguish configurations on the basis of internal computation steps.

The partial traces a configuration produces are given by $\mathcal{P}\sigma = \{\pi \mid \sigma \triangleright \pi\}$.

Definition 5.1 ▪ Partial Trace Equivalence
Let $(\Sigma, \longrightarrow, \mathsf{res})$ and $(\Sigma', \longrightarrow', \mathsf{res}')$ be $(\mathcal{E}, \tau)$-IDRS and let $\sigma \in \Sigma$ and $\sigma' \in \Sigma'$. Then $\sigma \overset{\text{Tr}}{=} \sigma' :\leftrightarrow \mathcal{P}\sigma = \mathcal{P}\sigma'$

Note that we set up IDRS such that the relation $\overset{\text{Tr}}{=}$ easily relates configurations from different IDRS because they agree on events and the meaning of τ.

Theorem 5.2 $\overset{\text{Tr}}{=}$ is an equivalence relation.

We prove the following important stability properties.

$$\textsc{Pr-Tau}\ \frac{\sigma \xrightarrow{\tau} \sigma' \qquad \sigma' \triangleright \pi}{\sigma \triangleright \pi} \qquad \textsc{Pr-End}\ \frac{}{\sigma \triangleright \epsilon} \qquad \textsc{Pr-Trm}\ \frac{\sigma \longrightarrow\text{-terminal}}{\sigma \triangleright \mathsf{res}\ \sigma}$$

$$\textsc{Pr-Evt}\ \frac{\sigma \xrightarrow{\psi} \sigma' \qquad \sigma' \triangleright \pi \qquad \phi \neq \tau}{\sigma \triangleright \psi, \pi}$$

Figure 5.2: Rules assigning partial traces to configurations.

Lemma 5.3 If $\sigma \longrightarrow^* \sigma'$ then $\sigma \overset{\text{Tr}}{=} \sigma'$.

Proof. From the fact that IDRS are τ-deterministic. ∎

Lemma 5.4 If $\sigma_1 \overset{\text{Tr}}{=} \sigma_2$ and $\sigma_1 \xrightarrow{\phi} \sigma_1'$ and $\sigma_2 \xrightarrow{\phi} \sigma_2'$ then $\sigma_1' \overset{\text{Tr}}{=} \sigma_2'$.

Proof. From the fact that IDRS are τ- and action-deterministic (Definition 3.26). Intuitively, σ_1' is the *only* state that σ_1 can reach with ϕ, and similarly for σ_2. ∎

Lemma 5.5 If $\mathcal{P}\sigma \subseteq \mathcal{P}\sigma'$ and $\sigma \Downarrow w$ then $\sigma' \Downarrow w$. If $\sigma \Downarrow w$ and $\sigma' \Downarrow w$ then $\sigma \overset{\text{Tr}}{=} \sigma'$.

Proof. From the fact that IDRS are τ- and action-deterministic and partial traces contain the result value. ∎

Partial Traces and Silent Divergence

The partial traces are expressive enough to characterize silent divergence.

Theorem 5.6 $\sigma \Uparrow$ if and only if $\mathcal{P}\sigma = \{\epsilon\}$.

Proof. The forward direction is by induction on $\triangleright$ and uses Lemma 5.3. The backwards direction is by coinduction and case distinction on whether and which reduction is possible. $\mathcal{P}\sigma = \{\epsilon\}$ ensures that only silent computation steps are non-contradictory. ∎

Corollary 5.7 If $\mathcal{P}\sigma' \subseteq \mathcal{P}\sigma$ and $\sigma \Uparrow$ then $\sigma' \Uparrow$. If $\sigma \Uparrow$ and $\sigma' \Uparrow$ then $\sigma \overset{\text{Tr}}{=} \sigma'$.

5.1.2 Infinite Trace Equivalence

Partial traces characterize equivalence based on finite objects. The relation $\blacktriangleright$ assignes a possibly infinite trace to a configuration. This involves a coinductive definition of traces, which is analogous to the definition of the partial traces $\prod$. We hence leave the definition implicit and use the symbols from $\prod$ for infinitary traces also. The rules co-inductively defining $\blacktriangleright$ are given in Figure 5.3.

$$\textsc{Tr-Div}\ \frac{\sigma \Uparrow}{\sigma \blacktriangleright \epsilon} \qquad \textsc{Tr-Trm}\ \frac{\sigma \Downarrow w}{\sigma \blacktriangleright w} \qquad \textsc{Tr-Evt}\ \frac{\sigma \longrightarrow^* \sigma' \qquad \sigma \xrightarrow{\psi} \sigma' \qquad \sigma' \blacktriangleright \pi}{\sigma \blacktriangleright \psi, \pi}$$

Figure 5.3: Rules assigning possibly infinite traces to configurations.

Rule Tr-Div is defined in terms of the coinductively defined notion of silent divergence $\Uparrow$. We do this, because a rule similar to Pr-Tau would introduce more traces than we want. To see this, consider a configuration δ such that $\delta \Uparrow$, and hence satisfies $\delta \xrightarrow{\tau} \delta$. With a rule similar to Pr-Tau in the definition of $\blacktriangleright$, one can show that $\delta \blacktriangleright \pi$ holds for any trace π.

The traces a configuration can produce are given as $\mathcal{T}\sigma = \{\pi \mid \sigma \blacktriangleright \pi\}$.

We prove the following important properties.

Lemma 5.8 $\blacktriangleright$ is left-total, i.e. for every σ there is a trace π such that $\sigma \blacktriangleright \pi$.

Proof. This proof requires informed excluded middle and indefinite description. Using these axioms, we coinductively define a function $\mathsf{tr} : \Sigma \to \Pi$ that constructs a trace given a configuration. We then show by coinduction that $\sigma \blacktriangleright \mathsf{tr}\,\sigma$. ■

Lemma 5.9 If $\sigma \longrightarrow^* \sigma'$ then $\mathcal{T}\sigma = \mathcal{T}\sigma'$.

Proof. From the fact that IDRS are τ-deterministic. ■

Lemma 5.10 If $\mathcal{T}\sigma_1 \subseteq \mathcal{T}\sigma_2$ and $\sigma_1 \xrightarrow{\phi} \sigma_1'$ and $\sigma_2 \xrightarrow{\phi} \sigma_2'$ then $\mathcal{T}\sigma_1' \subseteq \mathcal{T}\sigma_2'$.

Proof. From the fact that IDRS are τ- and action-deterministic (Definition 3.26): Intuitively, σ_1' is the *only* state that σ_1 can reach with ϕ, and similarly for σ_2. ■

Lemma 5.11 If $\mathcal{T}\sigma \subseteq \mathcal{T}\sigma'$ and $\sigma \Downarrow w$ then $\sigma' \Downarrow w$. If $\sigma \Downarrow w$ and $\sigma' \Downarrow w$ then $\mathcal{T}\sigma = \mathcal{T}\sigma'$.

Proof. From the fact that IDRS are τ- and action-deterministic and traces contain the result value. ■

Lemma 5.12 If $\sigma \overset{\text{Tr}}{=} \sigma'$ then $\mathcal{T}\sigma \subseteq \mathcal{T}\sigma'$.

Proof. By coinduction on the derivation of the trace $\pi \in \mathcal{T}\sigma$ with the stability properties of $\overset{\text{Tr}}{=}$ proven in Lemma 5.3 and Lemma 5.4 and Corollary 5.7. ■

Lemma 5.13 If $\mathcal{T}\sigma_1 \subseteq \mathcal{T}\sigma_2$ then $\mathcal{P}\sigma_1 \subseteq \mathcal{P}\sigma_2$.

Proof. By induction on the derivation of the partial trace $\pi \in \mathcal{P}\sigma_1$.

- The case for Pr-End is trivial.
- The case for Pr-Tau follows by induction with Lemma 5.9.
- The case for Pr-Trm follows from Lemma 5.11.
- In the case for Pr-Evt, we have that $\sigma_1 \xrightarrow{\psi} \sigma_1'$ and $\sigma_1' \triangleright \pi$, and must show $\sigma_2 \triangleright \psi, \pi$. We show that there is a trace π' such that $\sigma_1 \blacktriangleright \psi, \pi'$ using Tr-Evt and Lemma 5.8. From the first premise we know that also $\sigma_2 \blacktriangleright \psi, \pi'$, and by inversion that there is σ_2' such that $\sigma_2 \xrightarrow{\psi} \sigma_2'$ and $\sigma_2' \blacktriangleright \pi'$. The claim $\sigma_2 \triangleright \psi, \pi$ now follows from Tr-Evt and the inductive hypothesis, the premise of the latter follows from Lemma 5.10. ■

Theorem 5.14 $\sigma \overset{\text{Tr}}{=} \sigma'$ if and only if $\mathcal{T}\sigma = \mathcal{T}\sigma'$.

Proof. The forward direction is Lemma 5.12 and the backwards direction follows from Lemma 5.13. ■

Theorem 5.14 establishes that the finite partial traces suffice to characterize the infinite traces.

5.2 Implementation Relations

A relaxation of trace equivalence that is even more important in compiler verification is the *implementation relation*. The *implementation relation* specifies under which conditions a state is a correct implementation of another. The implementation relation is more permissive than trace equivalence, as it allows the target to refine the behavior of the source by determining underspecification.

Underspecification is present in many programming languages. One prominent reason is performance, for example by avoiding run-time checks. An example is division by zero in C [Ler09b]. The behavior of a division by zero is undefined to avoid the run-time cost of checking the divisor for zero and handle the error. A compiler can

thus safely assume that division by zero does not occur. Since division by zero is undefined, any havoc that occurs is well within the bounds of the specification. This is in stark contrast to languages like ML, where division by zero must throw an exception. Consequently, an ML compiler must insert code to check for a zero divisor at every division, except if the compiler cang prove that a zero divisor cannot occur.

We define two flavors of implementation relation: $\overset{\text{Tr}}{\searrow}$ which rules out changes to the external behavior, and $\overset{\text{Tr}}{\searrow_1}$, which allows non-determinism of external behavior to be resolved down to preserving at least one behavior. Both implementation relations allow to resolve underspecification arising from stuck states. The implementation relation $\overset{\text{Tr}}{\searrow_1}$ also allows resolving underspecification in the behavior of external calls. Our external calls are non-deterministic to allow an IDRS to specify an upper bound on the behavior of external calls. The implementation relation $\overset{\text{Tr}}{\searrow_1}$ allows a transformation to further restrict the behavior of an external call. This is useful, if late in the compiler an external call is actually instantiated with a concrete implementation. Such an implementation might be deterministic, and hence $\overset{\text{Tr}}{\searrow}$ would not allow the instantiation. At the same time, $\overset{\text{Tr}}{\searrow_1}$ requires that each system call retains at least one behavior, i.e. implementing a system call with a crash is ruled out.

5.2.1 Implementation Relation Preserving Non-Determinism

In this section we define the implementation relation $\overset{\text{Tr}}{\searrow}$, which requires that the target realizes exactly the as much non-determinism as the source. This relation hence preserves external behavior. Most transformations we consider in this thesis satisfy this relation, because they do not alter the behavior of external calls. The relation is useful to show that a transformation does not alter the specification of external behavior.

We start by defining lower bounds on the behavior of the implementation of a configuration σ, that is, a set of traces any valid implementation of the state σ must produce. Similarly, we define an upper bound on the behavior of the implementation of a configuration σ, that is, a set of traces any trace produced by any implementation of σ must stay within.

Definition 5.15 We define the relations $\trianglerighteq$ and $\overline{\triangleright}$ inductively using different subsets of the rules given in Figure 5.4. The relation $\trianglerighteq$ is defined as least fixed point of the rules Req-Tau, Req-Evt and Req-Final. The relation $\overline{\triangleright}$ is defined as the least fixed point of the rules Req-Tau, Req-Evt, Req-Final and Sp-Stuck. The lower bound on traces of the implementation of a state σ is defined as

$$\mathcal{R}\sigma = \{\pi \mid \sigma \trianglerighteq \pi\}$$

and the upper bound on traces of the implementation of a state σ is defined as

$$\mathcal{S}\sigma = \{\pi \mid \sigma \not\rhd \pi\}$$

$$\textsc{Req-Tau}\ \frac{\sigma \longrightarrow \sigma' \qquad \sigma' \rhd \pi}{\sigma \rhd \pi} \qquad \textsc{Req-Evt}\ \frac{\sigma \xrightarrow{\psi} \sigma' \qquad \sigma' \rhd \pi}{\sigma \rhd \psi, \pi} \qquad \textsc{Req-Final}\ \frac{\sigma \Downarrow v}{\sigma \rhd v} \qquad \textsc{Req-End}\ \frac{}{\sigma \rhd \epsilon}$$

$$\textsc{Sp-Stuck}\ \frac{\sigma \Downarrow \bot}{\sigma \not\rhd \pi}$$

Figure 5.4: Rules assigning request and spec prefixes to configurations.

Theorem 5.16 $\mathcal{R}\sigma \subseteq \mathcal{P}\sigma \subseteq \mathcal{S}\sigma$.

The lower bound $\mathcal{R}\sigma$ of traces an implementation of σ may produce is included in the traces $\mathcal{P}\sigma$ that σ itself produces; the only difference being that $\mathcal{R}\sigma$ does not contain the traces from $\mathcal{P}\sigma$ that end in $\bot$, but only the strict prefixes of those. The upper bound $\mathcal{S}\sigma$ on the traces an implementation of σ includes the traces $\mathcal{P}\sigma$ the configuration itself produces. Additionally, $\mathcal{S}\sigma$ contains any continuation of a trace ending in error, i.e. any trace $\pi\pi'$ where $\pi\bot \in \mathcal{P}\sigma$ and π' is any trace.

The sets $\mathcal{R}\sigma$ and $\mathcal{S}\sigma$ formally deal with underspecification encoded in the language semantics as "getting stuck". Whenever the semantics of the source gets stuck, we want to impose no restrictions on the implementation. At the same time, however, we require an implementation to include the prefixes of each trace that leads up to a stuck state in the source semantics.

Definition 5.17 $\sigma' \overset{\textsc{Tr}}{\searrow} \sigma$ if $\mathcal{R}\sigma \subseteq \mathcal{P}\sigma' \subseteq \mathcal{S}\sigma$.

Our definition of implementation relation is stronger than the implementation relation of CompCert and VeLLVM. Their implementation relations exclude traces which end in an error, and just require that all non-error traces of the original program are included in the non-error traces of the implementation. Note that if a program has no errors, then Sp-Stk, which distinguishes $\mathcal{R}$ from $\mathcal{S}$, is never applicable, and for such programs, $\overset{\textsc{Tr}}{\searrow}$ is equivalent to trace equivalence $\overset{\textsc{Tr}}{=}$.

$$\textsc{TrU-Div}\ \frac{\sigma \Uparrow}{\sigma \overline{\blacktriangleright} \epsilon} \qquad \textsc{TrU-Evt}\ \frac{\sigma \longrightarrow^* \sigma' \quad \sigma \xrightarrow{\psi} \sigma' \quad \sigma' \overline{\blacktriangleright} \pi}{\sigma \overline{\blacktriangleright} \psi, \pi} \qquad \textsc{TrU-Trm}\ \frac{\sigma \Downarrow v}{\sigma \overline{\blacktriangleright} v} \qquad \textsc{TrU-Trm}\ \frac{\sigma \Downarrow \bot}{\sigma \overline{\blacktriangleright} \pi}$$

Figure 5.5: Rules assigning possibly infinite upper traces to configurations.

5.2.2 Implementation Relation Allowing Determination

In this section, we define the implementation relation $\overset{\textsc{Tr}}{\searrow_1}$ that allows non-determinism present in the source to be determined: The relation requires preservation of only one external behavior at each external call, but accepts all non-deterministic behaviors allowed by the source. The relation is useful if a transformation tightens the specification of external calls and thereby removes behavior from system calls.

We define a relation $\overline{\blacktriangleright}$ such that $\mathcal{U}\,\sigma = \{\pi \mid \sigma \overline{\blacktriangleright} \pi\}$ is an upper bound on the traces any implementation of σ may produce. The relation $\overline{\blacktriangleright}$ is coinductively defined according to the rules in Figure 5.5.

We define the external implementation relation $\overset{\textsc{Tr}}{\searrow_1}$ as follows:

Definition 5.18 $\sigma' \overset{\textsc{Tr}}{\searrow_1} \sigma$ if $\mathcal{T}\sigma' \subseteq \mathcal{U}\sigma$.

$\overset{\textsc{Tr}}{\searrow_1}$ simply requires that all (possibly infinite) traces of the target σ' are admitted by the source σ. We cannot require that the source produces all traces of the target, because the target program might have resolved underspecification, i.e. replaced a stuck state with a state that has behavior. Lemma 5.8 ensures that at least one trace is preserved because it guarantees that $\mathcal{T}\sigma'$ cannot be empty.

The main reason why for including $\overset{\textsc{Tr}}{\searrow_1}$ in this development will become clear in §6.

6 Simulation-based Semantic Equivalence

In this section, we give bisimulation-based characterizations of the semantic equivalence relation from the previous section that are more amenable in proofs. We discuss properties of the bisimulation definition, weaknesses of the resulting proof methods, and challenges arising from their formalization in Coq. We discuss a bisimulation with a measure index in a style similar to CompCert [Ler09b] and show it equivalent to the non-indexed version.

In §6.3, we discuss the approach of Hur et al. [Hur+13] to formalizing co-inductive proofs in Coq, and use his method to characterize our bisimulation relation. This definition is the basis for most coinductive proofs in LVC. In §6.4, we discuss the approach of Schäfer and Smolka [SS17] to formalizing coinductive proofs in Coq.

In §6.5, we discuss the challenges with obtaining transitivity lemmas. We show that a famous counter-example from the literature can be adapted to show that one general form of transitivity is even unsound. We then use Schäfer and Smolka's approach to obtain a weaker transitivity result involving a lock-step simulation that is useful to modularize proofs.

6.1 Simulation-based Semantic Equivalence

Trace-based equivalences do not provide a straight-forward proof strategy. Both equivalence and refinement involve two trace inclusions, which must be shown separately. This is inconvenient for two reasons. First, the backwards direction is usually tedious, because it involves inverting the result of a transformation to gain information about its source program. Second, and more importantly, the proof of the backwards direction repeats most of the arguments of the forward direction and only differs in small details. Leroy [Ler09a] and Sevcík et al. [Sev+13] avoid backwards proofs altogether by exploiting that forward and backward simulation coincide in their setting. In this section, we develop similarity and bisimilarity as a proof tool for correctness proofs that does not rely on the coincidence of forward and backward simulation, as on IDRS, forward and backward direction do not coincide as shown in Example 18.1. Nevertheless, our definition of similarity allows to show forward and backward direction with one proof, because our proof rules for bisimilarity al-

$$\textsc{Bisim-Silent}\ \frac{\sigma_1 \longrightarrow^+ \sigma_1' \qquad \sigma_2 \longrightarrow^+ \sigma_2' \qquad \sigma_1' \sim \sigma_2'}{\sigma_1 \sim \sigma_2} \qquad \textsc{Bisim-Term}\ \frac{\sigma_1 \Downarrow w \qquad \sigma_2 \Downarrow w}{\sigma_1 \sim \sigma_2}$$

$$\textsc{Bisim-Extern}\ \frac{\sigma_1 \longrightarrow^* \sigma_1' \qquad \sigma_2 \longrightarrow^* \sigma_2' \qquad \sigma_1', \sigma_2' \text{ ready} \qquad \sigma_1' \overset{\sim}{\rightsquigarrow} \sigma_2' \qquad \sigma_2' \overset{\sim}{\rightsquigarrow} \sigma_1'}{\sigma_1 \sim \sigma_2}$$

$$\textsc{Sim-Extern}\ \frac{\sigma_1 \longrightarrow^* \sigma_1' \qquad \sigma_2 \longrightarrow^* \sigma_2' \qquad \sigma_1', \sigma_2' \text{ ready} \qquad \sigma_1' \overset{\sim}{\rightsquigarrow} \sigma_2'}{\sigma_1 \sim \sigma_2}$$

$$\textsc{Sim-Error}\ \frac{\sigma_1 \longrightarrow^* \sigma_1' \qquad \sigma_1' \not\longrightarrow \qquad \mathsf{res}\ \sigma_1' = \bot}{\sigma_1 \gtrsim \sigma_2}$$

Figure 6.1: Defining rules of similarity and bisimilarity.

low to argue the backwards direction, or more precisely, its contra-position, in-place together with the forward direction.

6.1.1 Bisimilarity

Bisimilarity is obtained as the greatest fixed-point of the proof rules given in Figure 6.1. Several definitions are required. We call a configuration *ready* if its next reduction produces an external event, that is, an event different from τ. We write $\sigma_1 \overset{R}{\rightsquigarrow} \sigma_2$ for a one-step forward-simulation property of a relation R: Every transition σ_1 takes can also be taken by σ_2, and the two successor configurations are related by R.

$$\begin{aligned} \sigma \text{ ready} &:= \exists \sigma', \sigma \xrightarrow{\phi} \sigma' \wedge \phi \neq \tau \\ \sigma_1 \overset{R}{\rightsquigarrow} \sigma_2 &:= \forall \sigma_1',\ \sigma_1 \xrightarrow{\phi} \sigma_1' \to \exists \sigma_2',\ \sigma_2 \xrightarrow{\phi} \sigma_2' \wedge R\, \sigma_1' \sigma_2' \end{aligned}$$

Definition 6.1 ■ Bisimilarity
Let $(S, \longrightarrow, \mathsf{res})$ and $(S', \longrightarrow', \mathsf{res}')$ be $(\mathcal{E}, \tau)$-IDRS. We define bisimilarity as relation of type $S \to S' \to \mathbf{P}$, where $\mathbf{P}$ is the universe of propositions. Bisimilarity $\sim$ is coinductively defined as the greatest relation closed under the rules Bisim-Silent, Bisim-Extern, Bisim-Term from Figure 6.1.

Description of the Rules Bisim-Silent allows to match finitely many steps on both sides, as long as all transitions are silent. This is sound because of the determinism requirements in IDRS, but would not yield a meaningful definition in a setting with internal non-determinism. Bisim-Extern ensures that every external transition of σ_1' is matched by the same external transition of σ_2', and vice versa. This ensures that if two states are in relation, they react to every possible result value of the external call in a bisimilar way. σ_1', σ_2' are required to be ready to simplify case distinctions by ensuring that the next event cannot be τ.

6.1.2 Reduction, Expansion and Divergence

It is easy to prove the following properties, which we built into the definition.

Lemma 6.2 ▪ Closedness under Expansion
If $\sigma_1' \sim \sigma_2'$ and $\sigma_1 \longrightarrow^* \sigma_1'$ and also $\sigma_2 \longrightarrow^* \sigma_2'$ then $\sigma_1 \sim \sigma_2$.

Proof. By inversion with τ- and action-determinism of IDRS. ■

Lemma 6.3 ▪ Closedness under Reduction
If $\sigma_1 \sim \sigma_2$ and $\sigma_1 \longrightarrow^* \sigma_1'$ and also $\sigma_2 \longrightarrow^* \sigma_2'$ then $\sigma_1' \sim \sigma_2'$.

Proof. By induction on the length of the reduction sequences with Lemma 6.2. ■

Closedness under silent expansion and reduction are important in proofs: With these properties we can show programs equivalent that differ in the number of silent steps they take. This is frequently required, for example, if a statement is removed, or implemented with more than one statement.

Lemma 6.4 If $\sigma \Uparrow$ and $\sigma' \Uparrow$ then $\sigma \sim \sigma'$.

6.1.3 Relation to Trace Equivalence

We show that bisimilarity characterizes trace equivalence, and similarity characterizes the implementation relation.

Lemma 6.5 If $\sigma_1 \sim \sigma_2$ then $\sigma_1 \overset{\text{Tr}}{=} \sigma_2$.

Proof. We show trace inclusion by induction on the derivation of $\sigma_1 \triangleright \pi$. The proof uses Lemma 6.3, inversion properties of $\sim$, and determinism of IDRS. ■

Lemma 6.6 If $\sigma_1 \overset{\text{Tr}}{=} \sigma_2$ then $\sigma_1 \sim \sigma_2$.

Proof. By coinduction followed by a case distinction on the behavior of σ_1 according to Theorem 3.29. Both termination and divergence transport along $\overset{\text{Tr}}{=}$ via Lemma 5.11 and Corollary 5.7, and bisimilarity can be proven using Bisim-Term and Lemma 6.4, respectively. Otherwise, there are σ_1', σ_1'' such that $\sigma_1 \longrightarrow^* \sigma_1'$ and $\sigma_1' \xrightarrow{\psi} \sigma_1''$ ready. We derive that $\sigma_1 \triangleright \psi, \epsilon$ and use the premise to obtain $\sigma_2 \triangleright \psi, \epsilon$. Inversion on the latter yields that there are σ_2', σ_2'' such that $\sigma_2 \longrightarrow^* \sigma_2'$ and $\sigma_2' \xrightarrow{\psi} \sigma_2''$. We apply Bisim-Extern, apply the cohypothesis and use Lemma 5.4 to discharge its premise. ■

Theorem 6.7 ▪ Soundness and Completeness
$\sigma_1 \overset{\text{Tr}}{=} \sigma_2 \leftrightarrow \sigma_1 \sim \sigma_2$.

Proof. The forward direction is Lemma 6.6. The backwards direction is Lemma 6.5. ■

6.2 Two Notions of Similarity

Definition 6.8 Let $(S, \longrightarrow, \mathsf{res})$ and $(S', \longrightarrow', \mathsf{res}')$ be $(\mathcal{E}, \tau)$-IDRS. Similarity $\gtrsim$ is defined as the greatest relation of type $S \to S' \to \mathbf{P}$ closed under the rules Bisim-Silent, Bisim-Extern, Bisim-Term and Sim-Error in Figure 6.1.

Sim-Error can be used to justify similarity for any configuration on the right side, if the left side can be shown to reduce to a stuck configuration.

Theorem 6.9 ▪ Soundness and Completeness
$\sigma \gtrsim \sigma' \leftrightarrow \sigma' \overset{\text{Tr}}{\searrow} \sigma$.

Proof. Similar to the proof of Theorem 6.7; we do not state the necessary lemmas here. ■

Definition 6.10 Let $(S, \longrightarrow, \mathsf{res})$ and $(S', \longrightarrow', \mathsf{res}')$ be $(\mathcal{E}, \tau)$-IDRS. Similarity $\gtrsim_1$ is defined as the greatest relation of type $S \to S' \to \mathbf{P}$ closed under the rules Bisim-Silent, Sim-Extern, Bisim-Term and Sim-Error in Figure 6.1.

Again, Sim-Error can be used to justify similarity for any configuration on the right side, if the left side can be shown to reduce to a stuck configuration. Additionally, for external calls only the backwards direction has to be shown via Sim-Extern.

Theorem 6.11 ▪ Soundness and Completeness
$\sigma \gtrsim_1 \sigma' \leftrightarrow \sigma' \overset{\text{Tr}}{\searrow_1} \sigma$.

Proof. Similar to the proof of Theorem 6.7; we do not state the necessary lemmas here. ∎

6.2.1 Bisimilarity as Symmetrization of Similarity

Our definition of similarity and bisimilarity is not standard. In particular, bisimilarity is not obtained as symmetrization of similarity. Definition by symmetrization is useful if the properties one wants to show are symmetric properties: A proof of bisimilarity can then be obtained from a proof of similarity and symmetry. To show a property that is not symmetric, the proofs for the forward and the backward direction are, in general, different and do not follow by symmetry.

In our setting, we make bisimilarity the basic definition, and obtain simulation by adding the "escape" rule Sim-Error that justifies similarity if the left configuration is stuck. Sim-Error realizes the intuition that if the source is stuck, we do not impose any restriction on behavior of the target.

In our proofs, we show forward and backward direction in one proof. A similarity proof usually works by assuming that the simulee is assumed to take a step, and the proof obligation is to show that the simulant can do a similar step such that the two resulting states are in relation again. In constrast, to show that two states are in bisimulation, the idea is to start with a case distinction on whether the simulee can take a step, and if it cannot, to show that the simulant is stuck as well. If the proof is done in this way, it is not required that forward and backward simulation coincide, and the approach scales naturally to a non-deterministic setting.

6.2.2 Divergence as Underspecification

Note the rule Bisim-Silent in Figure 6.1 requires at least one silent step on both sides. This is necessary for (bi) similarity to preserve silent divergence.

Clearly, if Bisim-Silent was formulated in terms of the reflexive-transitive closure instead of the transitive closure, it would always apply—with catastrophic consequences: simulation would derail to the full relation.

If, however, instead of Bisim-Silent we had the following rule, which admits the right-hand side configuration to not take a step the situation becomes more interesting.

$$\frac{\sigma_1 \longrightarrow^+ \sigma_1' \qquad \sigma_2 \longrightarrow^* \sigma_2' \qquad \sigma_1' \sim \sigma_2'}{\sigma_1 \sim \sigma_2}\ \textsc{Sim-Silent'}$$

In this setting, a silently diverging configuration δ could be shown equivalent to any other configuration σ by a simple coinductive argument: $\delta \sim \sigma$ can be shown for any configuration σ by coinduction and application of Sim-Silent', where δ does

a silent step and σ doesn't do a step. Clearly, the relation would be asymmetric then, as this reasoning only works if δ is on the left hand side. Moreover, it seems wrong to consider diverging states equivalent to any other state. For implementation relations, however, a definition that relates divering states on the left side to any state on the right side is interesting. Such a definition would essentially mean silent divergence becomes an instance of underspecification.

The question whether to treat silent divergence as underspecification is a normative one, and there are arguments for and against it. Two reasons why silent divergence should be treated as underspecification are: First, and most importantly, writing a silently divering program is most probably nonsensical. Second, a compiler could soundly assume that every loop that causes no external effects terminates, without proving termination of that loop. Interestingly, the C11 standard [ISO11] does precisely that, when it states in section §6.8.5p6 that a compiler may consider a loop terminating if its condition is not constant and the loop does not have external effects. See the following C working group at ISO/IEC note [Boe] for further discussion of this issue.

There are also three reasons why silent divergence should be preserved: First, it is standard in compiler verification to preserve silent divergence. CompCert [Ler09b] preserves silent divergence since the beginning, and other verified compilers such as VeLLVM [Zha+12] follow the approach. Second, preserving termination seems to better match current programmer intuition. And third, many optimizations, and in particular all optimizations in this thesis, simply preserve silent divergence. For this reason, we chose to setup our implementation relation in such a way that silent divergence is preserved. This yields slightly stronger results, as we also show that silent divergence is preserved. In general, we think that considering silent divergence underspecification is the more promising approach to follow, but this requires changing the intuitive expectation of correctness, which is a social issue that needs to be resolved first.

6.2.3 Constructing Bisimulations in Coq

The mechanical construction of bisimulation relations in plain Coq is inconvenient. To understand why this is the case, one has to understand the approach that was taken to implement coinductive types in Coq's type theory [Coq93]. The main intuition behind the implementation is that the elements of inductive and coinductive types are trees formed form applications of the constructors of the respective types. Each element of an inductive type is, in this sense, a tree of finite depth, while an element of a coinductive type can be a tree of infinite depth. Both inductive and coinductive types contain infinitely branching trees. For example, the proof that two silently diverging states are bisimilar is an infinite (linear) tree that relates the states and its successors.

Such a possibly infinitary tree is represented in plain Coq by a corecursive function that describes the tree's construction. The corecursive function serves as coinductive proof by describing an infinitary tree in a corecursive fashion, which is very similar to recursion: The function describes some finite portion of the infinitary tree, and then delegates to corecursion.

As corecursive functions must describe trees of possibly infinite depth, corecursive functions cannot be required to terminate. However, some validity requirement is in order, because a corecursive definition must not delegate directly to itself, as that would introduce absurdity in the same way as a recursively defined function with a defining equation à la $fx = fx$ does.

Emphasizing the computational perspective, a corecursive function can be understood as a construction plan for an infinitary tree that cannot be constructed in finite time. Productivity is the requirement that the corecursive function admits partial construction of the infinitary tree up to any finite depth in finite time. From this definition it follows that every terminating function is also productive [Coq93].

The inflexibility of coinductive proofs in Coq arises not from this general approach, but from the criterion that Coq uses to ensure productivity of corecursive definitions. This criterion requires corecursion to only occur directly as argument to at least one constructor of the coinductive type. This is sufficient for productivity, but precludes many uses of lemmas, as the constructor check does not work well across function application (i.e. lemmas), especially if the applied functions are themselves defined recursively or corecursively. This severly limits the proof power of direct coinductive proofs in Coq. We will circumvent this restriction in §6.3, where we encode the greatest fixed-point in a way that encodes the productivity requirement differently and thus side-steps the syntactic productivity criterion.

6.2.4 Bisimulation with Measure Index

The simulation used in CompCert [Ler09b] is indexed by a well-founded measure to account for stutter steps. Stutter steps occur in proofs if, for instance, a let binding was removed by an optimization. In this case, one wants to argue that the left-hand side configuration takes one step (executing the let binding) while the right-hand side does not reduce. As discussed in §6.2.2, we cannot add a stutter rule to the bisimulation without dire consequences, but also cannot apply the cohypothesis (i.e. corecursion) directly, because of the productivity requirement. This means that additional provisions are required in a proof where a stutter step is involved.

We now discuss how CompCert solves the problem of stutter steps with the introduction of a measure index to the simulation. The rules in Figure 6.2 show a modified definition of our bisimilarity that includes a well-founded measure index in the style of CompCert. A valid simulation proof now also requires that m is an element of some type M, and the less-than relation $<$ on M is well-founded. A simulation

$$\textsc{BisimIndex-Silent}\quad \frac{\sigma_1 \longrightarrow_{\mathrm{F}}^{+} \sigma_1' \qquad \sigma_2 \longrightarrow_{\mathrm{F}}^{+} \sigma_2' \qquad \sigma_1' \sim_m \sigma_2'}{\sigma_1 \sim_m \sigma_2}$$

$$\textsc{BisimIndex-Term}\quad \frac{\sigma_1 \Downarrow w \qquad \sigma_2 \Downarrow w}{\sigma_1 \sim_m \sigma_2}$$

$$\textsc{BisimIndex-Extern}\quad \frac{\sigma_1 \longrightarrow_{\mathrm{F}}^{*} \sigma_1' \qquad \sigma_2 \longrightarrow_{\mathrm{F}}^{*} \sigma_2' \qquad \sigma_1', \sigma_2' \text{ ready} \qquad \sigma_1' \overset{\sim}{\rightsquigarrow} \sigma_2' \qquad \sigma_2' \overset{\sim}{\rightsquigarrow} \sigma_1'}{\sigma_1 \sim_m \sigma_2}$$

$$\textsc{BisimIndex-Stutter}\quad \frac{\sigma_1 \longrightarrow_{\mathrm{F}}^{*} \sigma_1' \qquad \sigma_2 \longrightarrow_{\mathrm{F}}^{*} \sigma_2' \qquad \sigma_1' \sim_{m'} \sigma_2' \qquad m' < m}{\sigma_1 \sim_m \sigma_2}$$

Figure 6.2: Rules of a bisimilarity relation indexed with a well-founded measure. The index m is must be an element of some type M and $<$ is a well-founded relation of type $M \to M \to \mathbf{P}$.

proof can now use the rule Bisim-Stutter to accomodate stutter steps by decreasing the measure m. The following theorem shows that introducing the measure may increase proof power, but does not change the relation.

Theorem 6.12 Let M be a type and $m : M$ and let $<$ be a well-founded relation on M. Then $\sigma_1 \sim_m \sigma_2$ if and only if $\sigma_1 \sim \sigma_2$.

Proof. The backwards direction is by coinduction, as every rule of $\sim$ is also a rule of $\sim_m$. For the forward direction, we first show by well-founded induction on m that the top-level rule in any derivation of $\sim_m$ can be assumed to be BisimIndex-Silent, BisimIndex-Term, and BisimIndex-Extern, as the silent steps occuring in any preceeding applications of BisimIndex-Stutter can be merged into the silent steps of one of the former rules. With this inversion lemma, the forward direction can be shown by coinduction. ■

6.2.5 Invariants in Coinductive Proofs

Coinductive proofs always show that a certain relation (i.e. the simulation) is contained in the coinductively defined relation (i.e. bisimilarity). The simulation relation encodes the proof invariant, and is often formulated in terms of syntactic properties of the two programs, such as, for example, that the right-hand side program is obtained by applying a specific transformation to the left-hand side program.

One important property of such proofs is the interaction of the invariant encoded in the simulation with function application. Suppose we want to show that two function applications are in simulation. We must reduce both sides one step to be able

to apply the cohypothesis. Applying the coinductive hypothesis, however, we must show that the function bodies are in the syntactically defined simulation relation. In a coinductive proof, the invariant encoded in the simulation relation must be shown to also hold for function bodies. This means that a plain coinductive proof provides no function abstraction in the sense that applying related functions with related arguments is enough to yield related behavior. We show how to regain function abstraction for simulation proofs in §8.

6.3 Parametrized Coinduction

In the formal Coq development, we define simulation and bisimulation primarily via parametrized coinduction [Hur+13] to side-step the too restrictive guardedness check for co-fixed points in Coq and gain more proof power. Parametrized coinduction allows to account for productivity in a semantic way and supports the use of lemmas. In this section, we recapitulate the basic setup of parametrized coinduction following the work of Hur et al. [Hur+13]. In §6.3.2 we outline how parametrized coinduction works.

Definition 6.13 ▪ Complete Prelattice
A complete prelattice $(X, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ is a complete lattice that is defined with respect to $x \equiv y := x \sqsubseteq y \wedge y \sqsubseteq x$ instead of equality, i.e. a lattice which does not require anti-symmetry of $\sqsubseteq$.

The setup relies on the notion of a complete prelattice. Hur et al. do not require anti-symmetry, but base the paper presentation on a complete lattice nonetheless. We apply parametrized coinduction to functions into **P**, the universe of proprositions. Function types into **P** only form a complete lattice if the axioms of propositional extensionality and functional extensionality are assumed. The function types into **P** each form a complete prelattice without axioms, though.

Definition 6.14 ▪ Greatest Fixed Point
Let X be a complete prelattice. We define a function

$$\begin{aligned} \text{cofix} \;&:\; (X \to X) \to X \\ \text{cofix}\, f \;&:=\; \bigsqcup \{y \in X \mid y \sqsubseteq fy\} \end{aligned}$$

We use the notations $\nu x.s := \text{cofix}(\lambda x.s)$ and $\nu f := \text{cofix} f$.

Fact 6.15 Let X be a complete prelattice and f be a monotone function. Then $\text{cofix} f \sqsubseteq f(\text{cofix} f)$.

Definition 6.16 ■ Hur's Parametrized Greatest Fixed Point
Let X be a complete prelattice and $f : X \to X$ be a monotone function. We define a function

$$\begin{aligned}\mathbf{G} &: (X \xrightarrow{mon} X) \xrightarrow{mon} X \xrightarrow{mon} X\\ \mathbf{G}\ f\ x &:= \nu y.f(x \sqcup y)\end{aligned}$$

It is easy to check that $\mathbf{G}$ and $\mathbf{G}f$ are monotone.

Lemma 6.17 ■ Hur's Initialize
$\nu f \equiv \mathbf{G}f\bot$.

Lemma 6.18 ■ Hur's Unfold
$\mathbf{G}fx \equiv f(x \sqcup \mathbf{G}fx)$.

Lemma 6.19 ■ Hur's Accumulate
$y \sqsubseteq \mathbf{G}fx \leftrightarrow y \sqsubseteq \mathbf{G}f(x \sqcup y)$.

Proof. See [Hur+13]. ■

Corollary 6.20 ■ Hur's Coinduction
If $\forall z, x \sqsubseteq z \to y \sqsubseteq z \to y \sqsubseteq \mathbf{G}fz$ then $y \sqsubseteq \mathbf{G}fx$.

The usage of Corollary 6.20 as coinductive proof principle is outlined below in §6.3.2. The definition of $\mathbf{G}$ and together with its lemmas are provided by the Paco library [Hur+13]. The Paco library realizes $\mathbf{G}$ for each arity directly as a coinductively defined predicate, instead of using the cofixed point operator we defined for this presentation in Definition 6.14.

6.3.1 Bisimilarity as Parametrized Greatest Fixed Point

We obtain definitions equivalent to similarity and bisimilarity with the fixed point operator $\mathbf{G}$ from a single function. The use of a single function allows us to show many properties which hold for both, similarity and bisimilarity, with one lemma. This saves a lot of repetition particularly in the proof of transitivity.

Definition 6.21 We define the function sim that generates similarity and bisimilarity in Figure 6.3.

$$
\begin{aligned}
&\mathsf{STy} \ni s ::= \mathsf{bisim} \mid \mathsf{sim} \\
&sim : (\mathsf{STy} \to \Sigma \to \Sigma \to \mathbf{P}) \to (\mathsf{STy} \to \Sigma \to \Sigma \to \mathbf{P}) \\
&sim\, r\, p\, \sigma_1\, \sigma_2 := (\exists w.\ \sigma_1 \Downarrow w \wedge \sigma_2 \Downarrow w) && \textsc{Bisim-Term} \\
&\quad \vee (\exists \sigma_1' \sigma_2'.\ \sigma_1 \longrightarrow_{\mathrm{F}}{}^{+}\ \sigma_1' \wedge \sigma_2 \longrightarrow_{\mathrm{F}}{}^{+}\ \sigma_2' \wedge r\, p\, \sigma_1'\, \sigma_2') && \textsc{Bisim-Step} \\
&\quad \vee (\exists \sigma_1' \sigma_2'.\ \sigma_1 \longrightarrow_{\mathrm{F}}{}^{+}\ \sigma_1' \wedge \sigma_2 \longrightarrow_{\mathrm{F}}{}^{+}\ \sigma_2' \\
&\qquad \wedge \sigma_1', \sigma_2' \text{ ready} \wedge \sigma_1' \overset{r\,p}{\rightsquigarrow} \sigma_2' \wedge \sigma_2' \overset{r\,p}{\rightsquigarrow} \sigma_1') && \textsc{Bisim-Extern} \\
&\quad \vee (p = \mathsf{sim} \\
&\qquad \wedge \exists \sigma_1'.\ \sigma_1 \longrightarrow_{\mathrm{F}}^{*}\ \sigma_1' \wedge \sigma_1' \text{ terminal} \wedge \mathsf{res}\ \sigma_1' = \bot) && \textsc{Sim-Error}
\end{aligned}
$$

Figure 6.3: Generating function for simulation and bisimulation. Each disjunct corresponds to a rule from Figure 6.1.

6.3.2 Outline of Parametric Co-Induction

A parametrized coinduction using sim always has the form

$$R_r \subseteq \mathbf{G}\, sim\, r\, p$$

for relation R (the simulation) and r. Applying Corollary 6.20 sets up the coinduction: We have to show

$$R \subseteq \mathbf{G}\, sim\, r'\, p$$

but can assume $r \subseteq r'$ and $R \subseteq r'$. The assumption $R \subseteq r'$ is the coinductive hypothesis. The proof typically proceeds by unfolding $\mathbf{G}$ according to Lemma 6.18:

$$R \subseteq sim(r' \cup \mathbf{G}\, sim\, r')\, p$$

Unfolding exposes the generating function sim, each disjunct of which corresponds to a constructor (cf. Figure 6.1). In places where the constructor uses corecursion, the function sim applies its parameter, which is $r' \cup \mathbf{G}\, sim\, r'$ in our proof. This ensures that the co-hypothesis $R \subseteq r'$ is only applied after one of the constructors has been "used". In this way, the parameter in the definition of $\mathbf{G}$ encodes the productivity requirement semantically.

The Paco library comes with a set of elaborate tactics that automatically pack arbitrary sets of premises into an relation R, and help unpacking R after setting up the coinductive proof for the user's convenience. Packing and unpacking relies on the axiom of unicity of equivalence proofs.

6.3.3 Equivalence to the Non-Parametric Definition

Definition 6.22 Let $r : \mathsf{STy} \to \Sigma \to \Sigma \to \mathbf{P}$. We define:

$$\approx_r^p := \mathbf{G}\ sim\ r\ p$$
$$\gtrsim_r := \approx_r^{\mathsf{sim}}$$
$$\sim_r := \approx_r^{\mathsf{bisim}}$$

The following theorem establishes trust in the non-standard setup. The definitions obtained from parametrized coinduction and the function sim are equivalent to the more basic definitions from §6.1.

Lemma 6.23 $\gtrsim \equiv \gtrsim_\bot$ and $\sim \equiv \sim_\bot$ and $\gtrsim_1 \equiv \gtrsim_{1\bot}$.

We can now prove that the three relations are increasingly coarser:

Theorem 6.24 $\sim \subseteq \gtrsim \subseteq \gtrsim_1$.

6.3.4 Expansion and Contraction

Similarity has many important properties that we will need in correctness proofs. The lemmas are formulated with respect to $\approx_r^p$. This means that one proof establishes the property for both simulation and bisimulation.

A key properties of similarity is stability under silent reduction.

Lemma 6.25 The following lemmas stated in rule form are sound.

Sim-Expansion-Closed
$$\frac{\sigma_1 \longrightarrow_{\mathrm{F}}{}^* \sigma_1' \qquad \sigma_2 \longrightarrow_{\mathrm{F}}{}^* \sigma_2' \qquad \sigma_1' \approx_r^p \sigma_2'}{\sigma_1 \approx_r^p \sigma_2}$$

Sim-Contraction
$$\frac{\sigma_1 \longrightarrow_{\mathrm{F}} \sigma_1'' \qquad \sigma_1' \longrightarrow_{\mathrm{F}} \sigma_1'' \qquad \sigma_2 \longrightarrow_{\mathrm{F}} \sigma_2'' \qquad \sigma_2' \longrightarrow_{\mathrm{F}} \sigma_2'' \qquad \sigma_1 \approx_r^p \sigma_2}{\sigma_1' \approx_r^p \sigma_2'}$$

Sim-Expansion-Closed in particular means that whenever $\sigma \longrightarrow_{\mathrm{F}}{}^* \sigma'$ then for every r the two configurations are in simulation $\sigma \approx_r^p \sigma'$. Note that Lemma 6.25 holds for arbitrary r.

6.4 Greatest Fixed Points via Tower Induction

We now give yet another characterization of the greatest fixed-point introduced by Schäfer and Smolka [SS17] and based on the work by Pous [Pou16]. We will ultimately use this characterization to prove a useful transitivity result in §6.5.2. We

quickly recapitulate the setup of Schäfer and Smolka [SS17] here. The construction is based on an inductive characterization of the companion [Pou16].

Definition 6.26 ▪ Schäfer's f-tower
Be X a complete prelattice and $f : X \to X$ a monotone function. The f-tower is the predicate $T_f : X \to \mathbf{P}$ defined inductively by the following rules:

$$\frac{x \in T_f}{f\,x \in T_f} \qquad \frac{M \subseteq T_f}{\bigsqcap M \in T_f}$$

Note that we use the convention to use set notation for predicates and write $M \subseteq T_f$ for $\forall x, M\,x \to T_f\,x$. Using the f-tower, the companion of f can be defined.

Definition 6.27 ▪ Schäfer's companion
Let X be a complete prelattice and let the function $f : X \to X$ be monotone. The companion of f is defined as $t_f(x) := \bigsqcap\{y \in T_f \mid x \sqsubseteq y\}$.

Theorem 6.28 ▪ Schäfer's tower induction
Let X be a complete prelattice and $P : X \to \mathbf{P}$ be a predicate such that forall M, whenever $\bigsqcap M \in P$ then $M \subseteq P$. If $P(t_f\,x)$ implies $P(f(t_f\,x))$ then $P(t_f\,x)$ holds for all x.

Lemma 6.29 ▪ Schäfer's characterization
Let X be a complete prelattice and $f : X \to X$ be a monotone function. Then $\nu f \equiv t_f \bot$.

For the proofs of Theorem 6.28 and Lemma 6.29, see the work of Schäfer and Smolka [SS17].

6.5 Transitivity

Simulation and bisimulation are transitive relations. The main work is showing Lemma 6.30, from which the property follows.

Lemma 6.30 Let p : STy and $\sigma_1, \sigma_2, \sigma_3 : \Sigma$. If $\mathbf{G}\ sim \bot\, p\, \sigma_1\, \sigma_2$ and $\mathbf{G}\ sim \bot\, p\, \sigma_2'\, \sigma_3$ and $\sigma_2 \longrightarrow_{\mathrm{F}}{}^{*} \sigma_2'$ or $\sigma_2' \longrightarrow_{\mathrm{F}}{}^{*} \sigma_2$ then $\mathbf{G}\ sim \bot\, p\, \sigma_1\, \sigma_3$.

Proof. The proof is by case analysis on $\mathbf{G}\ sim \bot\, p\, \sigma_1\, \sigma_2$ and $\mathbf{G}\ sim \bot\, p\, \sigma_2'\, \sigma_3$. The cases are not difficult, but tedious. ∎

Lemma 6.31 $\gtrsim_\bot$ is a preorder.

Proof. Reflexivity is trivial; transitivity is an instance of Lemma 6.30. ■

Lemma 6.32 $\sim_\perp$ is an equivalence relation.

Proof. Reflexivity and symmetry are trivial; transitivity is an instance of Lemma 6.30. ■

Note that Lemma 6.31 and Lemma 6.32 follow from obvious properties of the corresponding trace-based relations they characterize as shown in Theorem 6.7 and Theorem 6.9. Exploring direct ways to show transitivity of bisimilarity, however, is important in its own right. The reason is that we need transitivity properties not only for $r = \perp$, but for arbitrary relations for them to be useful in proofs by coinduction. Transitivity of $\approx^p_r$ for arbitrary r, however, cannot hold, as it is well-known that weak bisimilarity is not an up-to technique for weak bisimilarity [SM92].

Example 6.33 We adapt a counter-example of Sangiorgi and Milner [SM92] to show that $\approx^p_r$ is not transitive for arbitrary r. Suppose σ and σ' are configurations such that $\sigma \longrightarrow_{\mathrm{F}\tau} \sigma'$ and σ' is not bisimilar to the silently diverging configuration δ. We assume that $\approx^p_r$ is transitive for p and all r, and use it to derive a contradiction, in particular, we show that $\sigma \approx^p_r \delta$ by parametric coinduction. We apply Bisim-Silent and reduce both sides one step and have to show $\sigma' \approx^p_{r'} \delta$ where $r \subseteq r'$. From Lemma 6.30 we know that $\sigma \approx^p_{r'} \sigma'$. Using this together with the hypothetical transitivity lemma, it remains to show that $\sigma(r' \cup \approx^p_{r'})\delta$, which follows by the coinductive hypothesis: a contradiction.

6.5.1 Lock-step Bisimilarity

It is, however, possible to obtain useful transitivity lemmas, but they require a different setup. We settle for a version of an approach by Pous [Pou06], which amounts to require that one of the bisimilarity premises of the transitivity lemma is not weak, but counts silent steps. We show a form of Pous result for the special case of lock-step bisimilarity. Technically, Pous approach works also for relations that are coarser than lock-step bisimilarity, but having the transitivity result for lock-step bisimilarity will suffice for our application.

Definition 6.34 We define the function $locksim$ that generates lock-step bisimilarity in Figure 6.4, which we write with the symbol $\overset{1}{\sim}_r$.

A lock-step simulation does not abstract from internal computation steps, but requires the two computations to proceed in lock-step fashion even on silent steps.

$$
\begin{aligned}
&\mathit{locksim} : (\Sigma \to \Sigma \to \mathbf{P}) \to (\Sigma \to \Sigma \to \mathbf{P}) \\
&\mathit{locksim}\, r\, p\, \sigma_1\, \sigma_2 := (\sigma_1 \not\longrightarrow \wedge \sigma_2 \not\longrightarrow \wedge \mathsf{res}\ \sigma_1 = \mathsf{res}\ \sigma_2) && \textsc{Bisim-Term} \\
&\quad \vee (\exists \sigma_1' \sigma_2'.\ \sigma_1 \longrightarrow_{\mathrm{F}} \sigma_1' \wedge \sigma_2 \longrightarrow_{\mathrm{F}} \sigma_2' \wedge r\, \sigma_1'\, \sigma_2') && \textsc{Bisim-Step} \\
&\quad \vee (\sigma_1, \sigma_2\ \mathsf{ready} \wedge \sigma_1 \overset{r}{\rightsquigarrow} \sigma_2 \wedge \sigma_2 \overset{r}{\rightsquigarrow} \sigma_1) && \textsc{Bisim-Extern}
\end{aligned}
$$

Figure 6.4: Generating function for lock-step bisimulation.

6.5.2 Transitivity of Tower Bisimilarity

In joint work with Steven Schäfer, we proved the following version of transitivity. This lemma shows that lock-step bisimilarity is an up-to relation for bisimilarity.

Lemma 6.35 ▪ Transitivity
If $t_{sim}\, r\, \sigma_1\, \sigma_2$ and $t_{locksim}\, \bot\, \sigma_2\, \sigma_3$ then $t_{sim}\, r\, \sigma_1\, \sigma_3$.

Proof. By tower induction (Theorem 6.28) and case analysis on the two premises. ■

The crucial point of Lemma 6.35 is that Schäfer's companion allows us to show that the relation r in the first premise is the same relation as in the conclusion. We will use Lemma 6.35 exemplary in Theorem 10.8 to show how this lemma is useful to modularize a proof.

6.5.3 Transitivity of Parametric Bisimulation

A result similar to Lemma 6.35 with Hur's characterization of the greatest fixed point seems possible using techniques based on up-to functions as detailed in the work of Hur et al. [Hur+13]. In our setting, however, this would have involved considerably more work, as we would have to prove the main theorems about up-to techniques, which are not contained in the Paco library.

7 While

As a first application of our basic program equivalence setup, we translate an imperative while language to IL/I. The imperative while language we translate is essentially Winskel's while [Win93], but with a change in the treatment of sequentialization. Our version of while simplifies continuation management by ensuring that sequentializations is left-normal, that is to say, all occurring sequentializations have a simple command on their left-hand side. This change allows for simpler semantic rules (one for each command) than the usual rules required for a sequentialization if the left-hand side can be a program. We ensure left-normality in the semantic rules by using an append procedure to compute the left-normal sequentialization of two programs. Intuitively, this approach works, because sequentialization in Winskel's while is associative.

7.1 Syntax

The difference to Winskel's while and other formulations found in the literature [Win93; Pie+17] is the formulation of sequentialization. Our version of the sequentialization is left-normal, that is to say, the left-hand side of every sequentialization is an an atomic statement, and does not allow further sequentialization there. To achieve this, we take programs to be lists of atomic statements and take precautions in the formulation of the semantics to maintain this invariant by appending lists if necessary. The syntax of linear while is given in Figure 7.1. We adopt the

$p, q ::= s; p \mid ;$		program
$s ::=$		statement
	$\mid x := e;$	assignment
	$\mid \texttt{if}\ (e)\ p\ \texttt{else}\ q$	conditional
	$\mid \texttt{while}\ (e)\ s$	iteration

Figure 7.1: Syntax of While.

$$\textsc{LWhile-Def}\quad \frac{[\![e]\!]\sigma = v}{\langle x := e; p \mid \sigma\rangle \longrightarrow \langle p \mid \sigma[x \mapsto v]\rangle}$$

$$\textsc{LWhile-Cond}\quad \frac{[\![e]\!]\sigma = i}{\langle \texttt{if}\ (e)\ p_{\mathbf{true}}\ \texttt{else}\ p_{\mathbf{false}},\ q \mid \sigma\rangle \longrightarrow \langle p_i +\!\!+\ q \mid \sigma\rangle}$$

$$\textsc{LWhile-Loop}\quad \frac{}{\langle \texttt{while}\ (e)\ p,\ q \mid \sigma\rangle \longrightarrow \langle (\texttt{if}\ (e)\ (p +\!\!+ \texttt{while}\ (e)\ p)),\ q \mid \sigma\rangle}$$

Figure 7.2: Semantics of While.

notational convention that when writing programs, we omit writing the semicolon separator after statements that themselves end in a semicolon. For example we write $x := e; y := e'$; instead of $x := e; ; y := e'$;.

7.2 Semantics

The semantics of linear while is defined on configurations $\langle p \mid E\rangle$ where p is a linear while program and E is a variable environment. The small-step semantics of linear while is given in Figure 7.2. The semantics uses list concatenation $+\!\!+$ to ensure linearity of programs. We also use the notation $\texttt{if}\ (e)\ p$ for programs of the form $\texttt{if}\ (e)\ p\ \texttt{else}\ ;$. This notation fits nicely with the semantic rule for conditional Cond, because for $[\![e]\!]\sigma = \mathbf{false}$ we have

$$\langle \texttt{if}\ (e)\ p\ \texttt{else}\ ;,\ q) \mid \sigma\rangle \longrightarrow \langle q \mid \sigma\rangle$$

Note how append $+\!\!+$ takes care of the empty list in the alternative of the condition and allows to reduce the conditional to q in one step.

7.3 Translation to IL/I

In this section we translate linear while programs to IL/I programs. The translation is given in Figure 7.3. Note that termination of LWtoIL is not obvious to Coq and requires using a measure function with the program package.

$$
\begin{aligned}
&\mathsf{LWtoIL} : \mathsf{program} \to \mathcal{F} \to \mathsf{Exp} \\
&\mathsf{LWtoIL}\ (x := e; p)\ f = \\
&\quad \texttt{let}\ x = e\ \texttt{in}\ \mathsf{LWtoIL}\ p\ f \\
&\mathsf{LWtoIL}\ (\texttt{if}\ (e)\ p_1\ \texttt{else}\ p_0; q)\ f = \\
&\quad \texttt{fun}\ g\ () = \mathsf{LWtoIL}\ q\ f\ \texttt{in} \\
&\quad \texttt{if}\ e\ \texttt{then}\ \mathsf{LWtoIL}\ p_1\ g \\
&\quad \texttt{else}\ \mathsf{LWtoIL}\ p_0\ g && g \text{ fresh} \\
&\mathsf{LWtoIL}\ (\texttt{while}\ (e)\ p; q)\ f = \\
&\quad \texttt{fun}\ g\ () = \\
&\qquad \texttt{if}\ e\ \texttt{then}\ \mathsf{LWtoIL}\ p\ g \\
&\qquad \texttt{else}\ \mathsf{LWtoIL}\ q\ f \\
&\quad \texttt{in}\ g\ () && g \text{ fresh}
\end{aligned}
$$

Figure 7.3: Translation from While to IL/I.

7.3.1 Correctness

The main issue in the proof is factoring out the part that requires coinduction, which we do in the following lemma.

Lemma 7.1 If

$$
\begin{aligned}
&\forall q\, L\, f\, E\, r, (\exists v, [\![e]\!]\, E = v \wedge \beta v = \mathbf{true}) \\
&\quad \to (\forall E', \langle q \mid E' \rangle \approx_r^p (L, E', f\,())_{\mathrm{I}}) \\
&\quad \to \langle p +\!\!+ q \mid E \rangle \approx_r^p (L, E, \mathsf{LWtoIL}\ p\ f)_{\mathrm{I}}
\end{aligned}
$$

and

$$
\forall E, \langle p \mid E \rangle \approx_r^p (f : (\epsilon, \texttt{if}\ e\ \texttt{then}\ \mathsf{LWtoIL}\ p\ f\ \texttt{else}\ t); L, E, t)_{\mathrm{I}}
$$

and there is v such that $[\![e]\!]\, E = v$ and $\beta v = \mathbf{true}$ then

$$
\begin{aligned}
&\langle p +\!\!+ \texttt{while}\ (e)\ p; q \mid E \rangle \\
&\approx_r^p (f : (\epsilon, \texttt{if}\ e\ \texttt{then}\ \mathsf{LWtoIL}\ p\ f\ \texttt{else}\ t); L, E, \mathsf{LWtoIL}\ p\ f)_{\mathrm{I}}.
\end{aligned}
$$

Proof. By coinduction. We immediately apply the first premise which reduces the proof obligation to showing for some E'

$$\begin{aligned} &\langle \texttt{while}\ (e)\ p; q \mid E' \rangle \\ \approx^p_r\ &(f : (\epsilon, \texttt{if}\ e\ \texttt{then}\ \mathsf{LWtoIL}\ p\ f\ \texttt{else}\ t); L, E', f\ ())_{\mathrm{I}} \end{aligned}$$

Case distinction on the evaluation behavior of the condition e.

- If there is v such that $[\![e]\!]\, E = v$ we apply a constructor of the simulation to reduce the left-hand side one step, and the right-hand side two steps (the concrete reduction depends on whether $\beta v = \mathbf{true}$). If $\beta v = \mathbf{true}$ we arrive at states related by the cohypothesis. If $\beta v = \mathbf{false}$ we arrive at states related by the second premise.
- If there is no v such that $[\![e]\!]\, E = v$ both sides are stuck and we are done. ■

The correctness proof is now routine.

Theorem 7.2 If for all environments E we have $\langle q \mid E \rangle \approx^p_r (L, E, f\ ())_{\mathrm{I}}$, then $\langle p +\!\!+ q \mid E \rangle \approx^p_r (L, E, \mathsf{LWtoIL}\ p\ f)_{\mathrm{I}}$.

Proof. By size-induction on p with Lemma 7.1. ■

8 Compatibility Rules for Inductive Simulation Proofs

In §5 we discussed different approaches to the formalization of coinductive proofs. Per-se, all approaches presented so far suffer from different weaknesses.

- With the notable exception of the measure-indexed simulation, stutter steps cannot be dealt with natively as described in §6.2.4.
- Function abstraction as described in §6.2.5 requires additional setup.
- Proof modularization with transitivity is difficult, because it is subject to productivity constraints which limits its application to lock-step simulations.

In this section, we develop an proof method for showing similarity that works by induction on the program structure. This proof method is the basis for all correctness proofs in LVC, and a naturally good match for correctness proofs of transformations that are defined by recursion on the program structure. Our inductive proof technique addresses all three weaknesses of plain simulation proofs: It naturally accommodates stutter steps without relying on a measure-indexed simulation. It provides function abstraction that realizes the slogan known from logical relations:

> applying related functions to related arguments produces related results

Under certain circumstances, the inductive method also allows free application of transitivity lemmas, without constraints to lock-step simulations.

The inductive method works by induction in the program structure. This allows the inductive hypothesis to be applied to structurally smaller arguments without further restrictions. To make this work, lemmas showing the constructors of the IL syntax are compatible with the simulation are required. In the case of function definitions, such a compatibility lemma will be difficult to get. There two problems that need to be dealt with. The first is that function definitions in IL introduce possibly diverging behaviors, which a simple structural induction does not account for. To account for behavior of fixed-points in IL, we show by coinduction that function definitions are compatible with the simulation. For this purpose we use parametrized coinduction

(§6.3) to stratify the proof, i.e. to make a notion of "productivity" tenable by introducing the parameter relation r (see §6.3). The stratification introduced the parameter r to the coinductive relation, which necessarily yields statements in the spirit of:

> for all r, applying r-related functions to related arguments produces r-related results

As a top-level correctness statement, this would be satisfactory, as we get the desired result for the simulation as an instance of the statement when we set r to the empty relation (§6.3.3). However, there is a problem with the function compatibility statement. Because it also requires stratification, the function compatibility statement roughly reads as follows:

> If (for all r, related function bodies produce r-related results when applied to related arguments in any r-related function context), then (for all r, the corresponding IL recursive functions are r-related).

Note the use of quantifies. There is a universal quantification of r in the premise. This forces us to generalize the whole proof in this way, even if we use the inductive method. The generalization, however, restricts the applicability of transitivity, which for good reasons does not hold for arbitrary r (see §6.5). Without the universal quantification, we would have to satisfy the premise of the compatibility lemma:

> for all r, related function bodies produce r-related results when applied to related arguments in any r-related function context

from only knowing that

> for $r = \bot$, related function bodies produce r-related results when applied to related arguments in any r-related function context

This property, which we call r-generalization, does not easily follow; however, we will show this property for a special case when we show that bisimilarity is contextual in §8.4. The proof of this property is very interesting.

This chapter proceeds as follows. We start by giving simple but useful compatibility lemmas for let-binding and conditionals. We then try to prove contextuality of bisimilarity, which is the simplest proof that incurs all the problems we discussed so far. We then formulate a compatibility lemma for function definitions in a very general form, and prove it correct. This lemma is one of the foundations of our inductive method, and we use it to finish the proof of contextuality of bisimilarity. We will discover that the result we obtain is not general enough, because we are missing r-generalization. We then show that under certain assumptions, r-generalization holds and use it to prove the usual results about contextual equivalence.

Sim-Let-Op
$$\frac{[\![e]\!]\,V = [\![e']\!]\,V' \qquad \forall v, (L, V[x \mapsto v], s)\ (\approx_r^p \cup\, r)\ (L', V'[x' \mapsto v], s')}{(L, V, \texttt{let}\ x = e\ \texttt{in}\ s) \approx_r^p (L', V', \texttt{let}\ x' = e'\ \texttt{in}\ s')}$$

Sim-Let-Call
$$\frac{[\![\overline{e}]\!]\,V = [\![\overline{e'}]\!]\,V' \qquad \forall v, (L, V[x \mapsto v], s)\ (\approx_r^p \cup\, r)\ (L', V'[x' \mapsto v], s')}{(L, V, \texttt{let}\ x = f\,\overline{e}\ \texttt{in}\ s) \approx_r^p (L', V', \texttt{let}\ x' = f\,\overline{e'}\ \texttt{in}\ s')}$$

Sim-Cond
$$\frac{[\![e]\!]\,V = [\![e']\!]\,V' \qquad \beta([\![e]\!]\,V) = \mathbf{true} \to (L, V, s)\ (\approx_r^p \cup\, r)\ (L', V', s') \qquad \beta([\![e]\!]\,V) = \mathbf{false} \to (L, V, t)\ (\approx_r^p \cup\, r)\ (L', V', t')}{(L, V, \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t) \approx_r^p (L', V', \texttt{if}\ e'\ \texttt{then}\ s'\ \texttt{else}\ t')}$$

Figure 8.1: Sound structural rules for let-binding and conditionals. Recall that β converts a value to a boolean.

8.1 Compatibility of Variable Binding and Conditional

We show several structural compatibility properties for let-binding and conditionals. The lemmas are stated in rule form and hold for both IL and IL/I, as they only depend on semantic rules that both languages share. In the formal development, we use a type-class based abstraction to only show the rules once. Here, we simply state them once and use them for either language. The rules are also valid for both simulation and bisimulation and are hence formulated with respect to $\approx_r^p$.

Lemma 8.1 The rules in Figure 8.1 are admissible.

Proof. We only show Sim-Let-Op. After rewriting with Lemma 6.18, we have to show that $(L, V, \texttt{let}\ x = e\ \texttt{in}\ s)$ and $(L', V', \texttt{let}\ x' = e'\ \texttt{in}\ s')$ are related by $\mathit{sim}\,(r \cup \approx_r^p)\,p$. Case analysis on $[\![e]\!]\,V$.

- Case $[\![e]\!]\,V = v$. We unfold sim and show the case Bisim-Silent. The two required successor states exist:
  1. $(L, V, \texttt{let}\ x = e\ \texttt{in}\ s) \longrightarrow_{\mathrm{F}}^{+} (L, V[x \mapsto v], s)$
  2. $(L', V', \texttt{let}\ x' = e'\ \texttt{in}\ s') \longrightarrow_{\mathrm{F}}^{+} (L', V'[x' \mapsto v], s')$

  $(L, V[x \mapsto v], s)\ (\approx_r^p \cup r)\ (L', V'[x' \mapsto v], s')$ holds by assumption, which finishes the case.
- Case $[\![e]\!]\,V = \bot$. We unfold sim and show the case Bisim-Term. Both states are terminal, and the way we defined the result function ensures that $(L, V, \texttt{let}\ x = e\ \texttt{in}\ s) \Downarrow \bot$ and $(L', V', \texttt{let}\ x' = e'\ \texttt{in}\ s') \Downarrow \bot$. ∎

We prove in general that conditionals can be eliminated if the value of the condition is statically known. Recall that β converts a value to a boolean.

Lemma 8.2 If

1. $\beta([\![e]\!]\varnothing) = \bot \to [\![e]\!]\,V = [\![e]\!]\,V'$
2. $\forall v, [\![e]\!]\,V = v \to \beta v = \mathbf{true} \to \beta([\![e]\!]\varnothing) \neq \mathbf{false} \to (L, V, s_1) \approx_r^p (L', V', s_1')$
3. $\forall v, [\![e]\!]\,V = v \to \beta v = \mathbf{false} \to \beta([\![e]\!]\varnothing) \neq \mathbf{true} \to (L, V, s_2) \approx_r^p (L', V', s_2')$

then

$$(L, V, \texttt{if}\,e\,\texttt{then}\,s_1\,\texttt{else}\,s_2) \approx_r^p (L', V', \quad \begin{array}{l}\text{if } [\![e]\!]\emptyset = \mathbf{true} \text{ then } s_1' \\ \text{else if } [\![e]\!]\emptyset = \mathbf{false} \text{ then } s_2' \\ \text{else } \texttt{if}\,e\,\texttt{then}\,s_1'\,\texttt{else}\,s_2').\end{array}$$

Proof. Case analysis on $\beta([\![e]\!]\varnothing)$.

- Case $\beta([\![e]\!]\varnothing) = \mathbf{true}$. By monotonicity of expression evaluation, there is a value v such that $[\![e]\!]\,V = v$ and $\beta v = \mathbf{true}$. We apply Sim-Expansion-Closed, reducing only the right side one step and finish with the second assumption.
- Case $\beta([\![e]\!]\varnothing) = \mathbf{false}$. Analogous to the previous case.
- Case $\beta([\![e]\!]\varnothing) = \bot$. Case analysis on $[\![e]\!]V$. If $[\![e]\!]\,V = \bot$, both sides are stuck by the first assumption. We unfold via Lemma 6.18 and use the case Sim-Term of sim to show simulation. If $[\![e]\!]\,V = v$, then $[\![e]\!]\,V' = v$ by the first assumption. Case analysis on βv. If $\beta v = \mathbf{true}$ ($\beta v = \mathbf{false}$) we apply Sim-Expansion-Closed to reduce both sides one step and finish with the second (third) assumption. ∎

8.2 Bisimilarity and Similarity are Contextual

To motivate the problem with the compatibility rule for function definitions, we embark on a result about bisimilarity. We want to show that a program s cannot distinguish bisimilar function contexts. This result is an important stepping stone on the way to show that bisimilarity and similarity are contextual, that is, sound for contextual equivalence and contextual approximation.

We start with an informal definition of program equivalence that we will realize formally later via Definition 8.23 when we have a framework for a general compatibility lemma for fixed-points in IL available. The definition of program equivalence regards two programs equivalent, if they behave equivalently in bisimilar function contexts.

Definition 8.3 ▪ Closure Equivalence
Two closures are equivalent under function contexts L and L', written as $L \mid L' \vdash (V, \overline{x}, s) \simeq_r^p (V', \overline{x}', s')$ if for all argument values $\overline{v}$ the following states are equivalent: $(L, V[\overline{x} \mapsto \overline{v}], s) \approx_r^p (L', V[\overline{x}' \mapsto \overline{v}], s')$.

Definition 8.4 ▪ Function Context Equivalence
Two function contexts L, L' are equivalent $L \simeq_r^p L'$ if they define the same functions, that is, $dom\, L = dom\, L'$, and for all $f \in dom\, L$ the closures are equivalent: $L^{-f} \mid L'^{-f} \vdash L_f \simeq_r^p L'_f$.

Definition 8.5 ▪ Program Equivalence
Two terms s and s' are equivalent $s \simeq_r^p s'$ if for all equivalent contexts $L \simeq_r^p L'$ we have $(L, V, s) \approx_r^p (L', V, s')$.

Allowing equivalent contexts, and not requiring syntactically equal contexts is important to know that IL functions behave extensionally. We now try to show that program equivalence is reflexive, which ultimately implies that in IL functions are extensional. The proof here will fail at first, but we will discover two important issues: First, we will discover which lemma we are missing and motivate the so called *extension lemma*, the proof of which is the main result of this section. Second, we reiterate the general difficulty of a coinductive proof for such a reflexivity property.

Lemma 8.6 ▪ Reflexivity
$s \simeq_r^p s$.

Proof. (Attempt) By induction on s.

- The case for let follows from the inductive hypothesis, and lemmas Sim-Let-Op and Sim-Let-Call.
- The conditional case follows by Sim-Cond and the inductive hypotheses.
- In the case of application $f\,\overline{e}$, since $dom\, L = dom\, L'$, either both L_f and L'_f exist or both don't. If they exists we are done by $L \simeq_r^p L'$. Otherwise, both sides are stuck and we finish with Sim-Term.
- The case for operation is by Sim-Term.
- In the function definition case, we have to show:

$$(L, V, \texttt{fun}\ F\ \texttt{in}\ t) \approx_r^p (L', V, \texttt{fun}\ F\ \texttt{in}\ t)$$

We reduce both sides one step and have to show

$$((\!|F|\!)_V; L, V, t) \approx_r^p ((\!|F|\!)_V; L', V, t)$$

We apply the inductive hypothesis and have to show its premise:

$$(\!|F|\!)_V; L \simeq_r^p (\!|F|\!)_V; L'$$

If f is from $dom\, L$, we are done by assumption. If f is a newly defined function $F_f = (\overline{x}, s)$, we have to show:

$$((\!|F|\!)_V; L, V[\overline{x} \mapsto \overline{v}], s) \approx_r^p ((\!|F|\!)_V; L', V[\overline{x} \mapsto \overline{v}], s)$$

```
fun f (x,y) =
 if (x > 9) then 1
 else f (x+1, y)
in f (3,2)
```

```
fun f (x) =
 if (x > 9) then 1
 else f (x+1)
in f (3)
```

Figure 8.2: An example program before (left) and after (right) dead variable elimination.

The inductive hypothesis applies, but leaves us with its premise:

$$(\!|F|\!)_V; L \simeq_r^p (\!|F|\!)_V; L'$$

The proof has gone in circle, because we failed to account for the semantic fixed-point definition of the recursive functions. In particular, we have not yet justified that if two function bodies are equivalent, then the two corresponding functions are also equivalent.

Note that if we had done this prove by co-induction, we would be stuck at the same point: We could not discharge the premise of the coinductive hypothesis. ■

8.3 A General Lemma for Compatibility of Fixed Points in IL

We now develop a general form of a compatibility lemma for fixed-points in IL, which we call extension lemma. This compatibility lemma enables an inductive proof method for (bi)similarity, i.e. for statements with the conclusion $(L, V, s) \approx_r^p (L', V', s')$. In general, such proofs require relating function contexts L, L' in a more elaborate way than Definition 8.4. The reason is we want to support transformations that *change the signature of functions*. For example, consider the two equivalent programs in Figure 8.2, where the parameter y was eliminated. To accommodate this and similar transformations, we related contexts with the relation $L\ r\ L' :^{\mathcal{P}} \Lambda$ which we define below. This relation is parameterized by a structure $\mathcal{P}$, which we call *proof relation*. $\mathcal{P}$ describes which functions in L, L' are related, and how their arguments and parameters must be related for the corresponding function applicatons to be equivalent.

Definition 8.7 ■ Proof Relation
A proof relation $(A, Param, Arg, Idx)$ is a tuple such that

1. A : Type
2. $Param : A \to \overline{\mathcal{V}} \to \overline{\mathcal{V}} \to \mathbf{P}$

3 $Arg : (\mathcal{V} \to \mathbb{V}_\bot) \to (\mathcal{V} \to \mathbb{V}_\bot) \to A \to \overline{\mathbb{V}} \to \overline{\mathbb{V}} \to \mathbf{P}$

4 $Idx : A \to \mathcal{F} \to \mathcal{F} \to \mathbf{P}$

The proof relation is indexed by a type A, which is used to associate additional information with a function. This will often be program analysis information. For the example in Figure 8.2, A could be instantiated with set $\mathcal{V}$ and we could associated with each function the set of variables a it depends on. A proof relation defines defines relative to this additional information of type A the conditions on formal parameters ($Param$), arguments at function calls (Arg), and function names (Idx) that define if two functions are related.

For the verification of the transformation from Figure 8.2, for example, the proof relation specifies via $Param$ that the parameters the function in the right-hand side program (that is the functions in L') are obtained from the parameters of the corresponding left-hand side function in L by erasing parameters not in the set of type $A = \mathsf{set}\,\mathcal{V}$ associated with the function. The proof relation can specify a similar erasure for arguments at applications by defining Arg accordingly.

8.3.1 Relating Function Contexts

We define the relation $L\ r\ L' :^{\mathcal{P}} \Lambda$ that relates functions from L, L' according to a proof relation Pr.

Definition 8.8 Given a proof relation $\mathcal{P}$ and analysis information context Λ, and function context L, L' we say Λ and L, L' are in parameter relation with respect to $\mathcal{P}$, written $Param\,\Lambda\,L\,L'$, if whenever $Idx\,\Lambda_f\,f\,f'$ and $L_f = (V, \overline{x}, s)$ and $L'_{f'} = (V', \overline{x}', s')$ then $Param\,\Lambda_f\,\overline{x}\,\overline{x}'$.

Definition 8.9 Given a proof relation $\mathcal{P}$, and function context L, L', and analysis information context Λ, we define a relation $\mathsf{App}^{\mathcal{P}}\,\Lambda\,L\,L'$ on configurations such that

$$\begin{aligned}
&\mathsf{App}^{\mathcal{P}}\,\Lambda\,L\,L'\,(L, V, f\,\overline{e})\,(L', V', f'\,\overline{e'}) \\
:\leftrightarrow\ &\exists W\,W'\overline{x}\,\overline{x}'s\,s', L_f = (W, \overline{x}, s) \wedge L'_{f'} = (W', \overline{x}', s') \\
&\wedge\ Idx\,\Lambda_f\,f\,f' \wedge Arg\,W\,W'\,\Lambda_f\,([\![\overline{e}]\!]\,V)\,([\![\overline{e'}]\!]\,V') \\
&\wedge\ |\overline{x}| = |\overline{e}| \wedge |\overline{x}'| = |\overline{e}'|
\end{aligned}$$

The relation $\mathsf{App}^{\mathcal{P}}\,\Lambda\,L\,L'$ relates application configurations that satisfy the requirements imposed by the proof relation.

Definition 8.10 Two function contexts L, L' are in r-relation with respect to Λ and $\mathcal{P}$, written $L\ r\ L' :^{\mathcal{P}} \Lambda$, if

1 $dom\,\Lambda = dom\,L$

2 $Param\,\Lambda\,L\,L'$

3 $Idx\,\Lambda_f\,f\,f' \to (f \in dom\,L \leftrightarrow f' \in dom\,L')$

4 $\mathsf{App}^{\mathcal{P}}\,\Lambda\,L\,L' \subseteq r$

Lemma 8.11 If $r \subseteq r'$ and $L\ r\ L' :^{\mathcal{P}} \Lambda$ then $L\ r'\ L' :^{\mathcal{P}} \Lambda$.

8.3.2 Extending Related Function Contexts

We prove the central lemma that enables the inductive proof method, which we call the *extension lemma*. The extension lemma solves the problem in the proof of Lemma 8.6 in a general way: When descending under function definitions, related L and L' are extended with new closures. The inductive hypothesis provides that the bodies of these functions are related, but this does not readily mean that the corresponding semantic fixed-points are related. The extension lemma (Lemma 8.18 below) accounts for the semantics of the fixed-point operator.

Definition 8.12 Given a proof relation $\mathcal{P}$, function context L, L' and K, K', and analysis information Λ, we define a relation $\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\,\Lambda\,K\,K'$ on configurations such that

$$\begin{aligned}&\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\,\Lambda\,K\,K'\,(K; L, V[\overline{x} \mapsto v], s)\,(K; L', V'[\overline{x}' \mapsto v'], s')\\ &:\leftrightarrow\ \exists f\,f'\,W\,W', K_f = (W, \overline{x}, s) \wedge K'_{f'} = (W', \overline{x}', s')\\ &\qquad \wedge Idx\,\Lambda_f\,f\,f' \wedge Arg\,W\,W'\,\Lambda_f\,\overline{v}\,\overline{v'}\end{aligned}$$

The relation $\mathsf{Bdy}^{\mathcal{P}}_{K,K'}\,\overline{a}\,F\,F'$ relates configurations that are obtained by one reduction from application configurations that satisfy the requirements imposed by the proof relation. We set up our inductive proofs such that the inductive hypothesis provides that these configurations are equivalent.

Definition 8.13 A proof relation $\mathcal{P}$ separates two function contexts K, K' under $\Lambda; \Lambda'$, written $K \parallel \Lambda; \Lambda' \parallel_{\mathcal{P}} K'$ if:

1 $dom\,\Lambda = dom\,K$

2 $Idx\,(\Lambda; \Lambda')_f\,f\,f' \to (f \in dom\,K \leftrightarrow f' \in dom\,K')$

Lemma 8.14 ▪ Extending Parameter Relations
If $K \parallel \Lambda; \Lambda' \parallel_{\mathcal{P}} K'$ and we also have $Param\,\Lambda\,K\,K'$ and $Param\,\Lambda'\,L\,L'$ then $Param\,(\Lambda; \Lambda')\,(K; L)\,(K'; L')$.

Separation requires that functions in K are only related to functions in K', and vice versa.

Lemma 8.15 Let $\mathcal{P}$ be a proof relation. If

1. $K \parallel \Lambda; \Lambda' \parallel_{\mathcal{P}} K'$,
2. $\mathit{Param}\, \Lambda\, K\, K'$,
3. $\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\, (\Lambda; \Lambda')\, K\, K \subseteq (\approx^p_r \cup\, r)$, and
4. $L \approx^p_r L' :^{\mathcal{P}} \Lambda'$

then $K; L \approx^p_r K; L' :^{\mathcal{P}} \Lambda; \Lambda'$.

Proof. The proof distinguishes whether the function pair is from K and K' or L and L'. This is possible because $\mathcal{P}$ separates K, K' under Λ. In the first case, the result follows from (3) after a lock-step simulation step that reduces function applications on both sides. In the second case, the result follows from (4) and Sim-Retract. ■

Definition 8.16 Given a proof relation $\mathcal{P}$, function context K, K' are in r-relation under L and L' with respect to $\mathcal{P}$ and $\Lambda; \Lambda'$, which we write as $L \mid L' \vdash K\ r\ K' :^{\mathcal{P}} \Lambda; \Lambda'$, if

1. $K \parallel \Lambda; \Lambda' \parallel_{\mathcal{P}} K'$
2. $\mathit{Param}\, \Lambda\, K\, K'$
3. $\forall r, (K;L)\ r\ (K';L') :^{\mathcal{P}} \Lambda; \Lambda' \to \mathsf{Bdy}^{\mathcal{P}}_{L,L'}\, K\, K'\, (\Lambda; \Lambda') \subseteq r$

Lemma 8.17 ▪ **Fix Compatibility**
Let $\mathcal{P}$ be a proof relation. Then if it holds that $L \mid L' \vdash K \approx^p_r K' :^{\mathcal{P}} \Lambda; \Lambda'$ and $L \approx^p_r L' :^{\mathcal{P}} \Lambda'$ then $\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\, K\, K'\, (\Lambda; \Lambda') \subseteq \approx^p_r$.

Proof. By coinduction via Corollary 6.20. We have to show

$$\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\, F\, F'\, (\Lambda; \Lambda') \subseteq \gtrsim_{r'}$$

from $r \subseteq r'$ and the coinductive hypothesis $\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\, F\, F'\, (\Lambda; \Lambda') \subseteq r'$. Applying clause (3) of the first premise reduces the proof obligation to

$$K; L \approx^p_{r'} K; L' :^{\mathcal{P}} \Lambda; \Lambda'$$

We apply Lemma 8.15. The third premise of Lemma 8.15 follows from the coinductive hypothesis and $r' \subseteq \gtrsim_{r'} \cup\, r'$, the fourth premise follows from monotonicity (Definition 6.16 and Lemma 8.11). ■

Lemma 8.17 shows that equivalence of function bodies is sufficient to show that the corresponding recursive functions are equivalent.

Lemma 8.18 ▪ Extension
Let $\mathcal{P}$ be a proof relation. If we have $L \mid L' \vdash K \approx_r^p K' :^{\mathcal{P}} \Lambda; \Lambda'$ and $L \approx_r^p L' :^{\mathcal{P}} \Lambda'$ then $K; L \approx_r^p K'; L' :^{\mathcal{P}} \Lambda; \Lambda'$.

Proof. We apply Lemma 8.15. The only non-trivial premise is to show

$$\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\, K\, K'\, (\Lambda; \Lambda') \subseteq \approx_r^p \cup\, r$$

We make use of the fact $\approx_r^p \subseteq \approx_r^p \cup\, r$ and finish the proof with Lemma 8.17. ■

Lemma 8.19 ▪ Fun Compatibility
Let $\mathcal{P}$ be a proof relation. If $L \mid L' \vdash (\!|F|\!)_V \approx_r^p (\!|F'|\!)_{V'} :^{\mathcal{P}} \Lambda; \Lambda'$ and $L \approx_r^p L' :^{\mathcal{P}} \Lambda'$ and

$$\begin{aligned} \forall r,\, (\!|F|\!)_V; L \approx_r^p (\!|F'|\!)_{V'}; L' :^{\mathcal{P}} \Lambda; \Lambda' \to \\ (\!|F|\!)_V; L, V, t) \approx_r^p ((\!|F'|\!)_{V'}; L', V', t') \end{aligned}$$

then $(L, V, \texttt{fun}\ F\ \texttt{in}\ t) \approx_r^p (L', V', \texttt{fun}\ F'\ \texttt{in}\ t')$.

Proof. We reduce both sides one step. We apply the last premise and have to show $(\!|F|\!)_V; L \approx_r^p (\!|F'|\!)_{V'}; L' :^{\mathcal{P}} \Lambda; \Lambda'$. Lemma 8.18 finishes the proof. ■

When using Lemma 8.18 or Lemma 8.19 it suffices to show that the bodies of new function definitions are related according to Definition 8.16. Item (3) already provides that the contexts containing the new functions are related. §9 contains several proofs that show-case the inductive method in detail.

8.3.3 Using Related Function Contexts to Prove the Application Case

Definition 8.20 Argument evaluation of $L, V, \overline{e}$ and $L', V', \overline{e}'$ agrees with respect to p and $\mathcal{P}$ if whenever $L_f = (W, \overline{x}, s)$ and $L_{f'} = (W, \overline{x}', s')$ and $Param\, a\, \overline{x}\, \overline{x}'$ then

1. if $[\![\overline{e}]\!]\, V = \overline{v}$ and $|\overline{x}| = |\overline{v}|$ then there exists $\overline{v}'$ such that $[\![\overline{e}']\!]\, V = \overline{v}'$ and $|\overline{x}'| = |\overline{v}'|$ and $Arg\, W\, W'\, a\, \overline{v}\, \overline{v}'$
2. if $p =$ bisim and $[\![\overline{e}]\!]\, V = \bot$ then $[\![\overline{e}']\!]\, V' = \bot$
3. if $p =$ bisim and $|\overline{x}| \neq |\overline{e}|$ then $|\overline{x}'| \neq |\overline{e}'|$

Lemma 8.21 Let $\mathcal{P}$ be a proof relation. If

1. $L \approx_r^p L' :^{\mathcal{P}} \Lambda$
2. $Idx\, \Lambda_f\, f\, f'$

3 argument evaluation of $L, V, \overline{e}$ and $L', V', \overline{e}'$ agrees with respect to p and $\mathcal{P}$

then $(L, V, f\,\overline{e}) \approx_r^p (L', V', f'\,\overline{e}')$.

Proof. From (2) we have that Λ_f is defined, and by definition of (1) $dom\,\Lambda = dom\,L$, which means $f \in dom\,L$, hence again by definition of (2) $f' \in dom\,L'$. We assume that $L_f = (V, \overline{x}, s)$ and $L_{f'} = (V, \overline{x}', s')$. By definition of (1) we know $Param\,\overline{a}\,L\,L'$, hence $Param\,\Lambda_f\,\overline{x}\,\overline{x'}$. Case analysis.

- Case $[\![\overline{e}]\!]\,V = \overline{v}$.
  - If $|\overline{x}| = |\overline{e}|$ we exploit clause (1) of premise (3) and obtain the fact

    $$\mathsf{App}^{\mathcal{P}}\,\Lambda\,L\,L'\,(L, V, f\,\overline{e})\,(L', V', f'\,\overline{e}')$$

    We know $\mathsf{App}^{\mathcal{P}}\,\Lambda\,L\,L' \subseteq r$ from premise (1) and are done.
  - If $|\overline{x}| \neq |\overline{e}|$. If $p = \mathsf{sim}$, we are done using Sim-Error. If $p = \mathsf{bisim}$, we exploit clause (3) of assumption (3) and obtain that $|\overline{x}'| \neq |\overline{e}'|$. Both sides are stuck (Sim-Term).
- Case $[\![\overline{e}]\!]\,V = \bot$. If $p = \mathsf{sim}$, we are done by Sim-Error. If $p = \mathsf{bisim}$, we exploit clause (2) of assumption (3) and obtain that $[\![\overline{e}']\!]\,V' = \bot$. Both sides are stuck. ∎

8.4 Bisimulation and Contexts

We now finish the proof of Lemma 8.6 with the inductive method. We first define a simple proof relation.

Definition 8.22 We define the proof relation $\mathcal{P}_{\mathsf{ctx}}$ where

$$\begin{aligned} A &:= \mathsf{list}\,\mathcal{V} \\ Param\,\overline{x}\,\overline{y}\,\overline{y'} &:= |\overline{x}| = |\overline{y}| \wedge |\overline{y}| = |\overline{y'}| \\ Arg\,V\,V'\,\overline{x}\,\overline{v}\,\overline{v'} &:= \overline{v'} = \overline{v} \wedge |\overline{x}| = |\overline{v}| \\ Idx\,\_\,f\,f' &:= f = f' \end{aligned}$$

We define a projection from a closure to the list of parameter lists: $\mathsf{pa}\,(\overline{x}, s) = \overline{x}$ and lift it to function contexts in a point-wise fashion.

Definition 8.23 ▪ Program Equivalence with Proof Relation
Two terms s and s' are bisimilar, written $s \dot{\simeq}_r^p s'$, if for all bisimilar contexts $L \approx_r^p L' :^{\mathcal{P}_{\mathsf{ctx}}} \mathsf{pa}\,L'$ we have $(L, V, s) \approx_r^p (L', V, s')$:

$$s \dot{\simeq}_r^p s' := \forall L\,L',\ L \approx_r^p L' :^{\mathcal{P}_{\mathsf{ctx}}} \mathsf{pa}\,L' \to (L, V, s) \approx_r^p (L', V, s')$$

We now informally argue that Definition 8.23 realizes the program equivalence from Definition 8.5.

Lemma 8.24 $\simeq_r^p = \dot{\simeq}_r^p$.

Proof. The proof is informal and amounts to showing that $L \approx_r^p L' :^{\mathcal{P}_{\text{ctx}}} \mathsf{pa}\, L'$ is equivalent to function context equivalence (Definition 8.4) of L and L', i.e $L \simeq_r^p L'$. This is the case because function application with different parameter and argument length get stuck. ∎

Lemma 8.25 ▪ **Reflexivity**
$s \dot{\simeq}_r^p s$.

Lemma 8.26 ▪ **Reflexivity**
$L \approx_r^p L :^{\mathcal{P}} \mathsf{pa}\, L$.

Proof. By induction just as the proof of Lemma 8.25; we use Lemma 8.18 where we previously got stuck. ∎

Lemma 8.27 ▪ **Transitivity**
$s \dot{\simeq}_\bot^p s' \to s' \dot{\simeq}_\bot^p s'' \to s \dot{\simeq}_\bot^p s''$.

Theorem 8.28 $s \dot{\simeq}_\bot^p s' \to s \dot{\simeq}_r^p s'$.

Proof. Unfolding definitions, we have to show that

$$\forall L\, L',\ L \approx_\bot^p L' :^{\mathcal{P}_{\text{ctx}}} \mathsf{pa}\, L' \to (L, V, s) \approx_\bot^p (L', V, s')$$

implies

$$\forall\, r L\, L',\ L \approx_r^p L' :^{\mathcal{P}_{\text{ctx}}} \mathsf{pa}\, L' \to (L, V, s) \approx_r^p (L', V, s').$$

The proof is involved. We first use external events not occurring in s, s' to construct L and L' in such that $L \approx_\bot^p L' :^{\mathcal{P}_{\text{ctx}}} \mathsf{pa}\, L'$ and such that a function in L is only bisimilar to the function of the same name in L' and only if both functions are called with the same arguments. Using this context, we instantiate the premise and obtain a proof of $(L, V, s) \approx_\bot^p (L', V, s')$. We can then invert this proof to construct a proof of $(L, V, s) \approx_r^p (L', V, s')$, because by the properties of L and L', calls to the context must correspond. ∎

The proof of Theorem 8.28 is interesting in its own right. The proof relies on the fact that contexts L, L' can be constructed. For a different proof relation, we would have to construct contexts L, L' such that only functions related by Idx with arguments values related by Arg are related. This is essentially a matter of whether the parameter relation Arg is IL decidable. We think that Theorem 8.28 can be generalized to proof relations that are IL decidable in this sense, but have no proof.

With Theorem 8.28 we can show a congruence rule for function definitions from Lemma 8.19, which we proved earlier.

Lemma 8.29 If we have $\forall i, s_i \mathrel{\dot{\simeq}^p_\bot} s'_i$ and $t \mathrel{\dot{\simeq}^p_\bot} t'$ then

$$\texttt{fun}\ \overline{f\ \overline{x} = s}\ \texttt{in}\ t \mathrel{\dot{\simeq}^p_\bot} \texttt{fun}\ \overline{f\ \overline{x} = s'}\ \texttt{in}\ t'.$$

Proof. Directly from Lemma 8.19 with Theorem 8.28 to discharge the premises, which require all-quantified parameter relations r. ∎

8.4.1 Contextual Equivalence

We define contextual equivalence for IL terms. For IL it is not sufficient to observe termination and non-termination, as an IL context cannot observe the result value of the term that is substituted into the hole.

Definition 8.30 ▪ Contextual Equivalence/Approximation
Two terms s and s' are contextually equivalent/contextually approximations, written $s \mathrel{\simeq^p_{ctx}} s'$, if for all IL contexts C, i.e. IL terms with a hole, we have we have $(\varnothing, \emptyset, C[s]) \approx^p_\bot (\varnothing, \emptyset, C[s'])$, i.e. in states with empty function contexts and empty variable environments, the two terms are bisimilar in every context.

Definition Definition 8.30 is parametric in p, and yields contextual equivalence for $p = \text{bisim}$ and contextual approximation for $p = \text{sim}$.

Theorem 8.31 $s \mathrel{\dot{\simeq}^p_\bot} s' \to s \mathrel{\simeq^p_{ctx}} s'$.

Proof. Induction on C and in the function definition case Lemma 8.18, one premise of which is discharged with Theorem 8.28. ∎

Theorem 8.32 $s \mathrel{\simeq^p_{ctx}} s' \to s \mathrel{\dot{\simeq}^p_\bot} s'$.

Theorem 8.32 ensures that a syntactic context can encode all information of valid semantic contexts. It requires a helper lemma that shows that for every function context L, we can constructor a context C that produces a function context bisimilar to L.

9 Liveness and Dead Variable Elimination

This section introduces two basic notions from compiler construction: liveness and true liveness. Liveness is essential for the definition of coherence, and an important piece of analysis information for the register allocation approach we discuss in §13. True liveness is the basis for dead variable elimination (DVE), which is very useful in practice and which we prove correct in this section.

The first notion of liveness we introduce in §9.1 is simply called liveness: a variable is live at a program point if it may be read *later on*. This means that the value of a variable that is not live cannot influence the behavior of the program. A variable, however, may be live even if its value cannot influence the computation of the program. This could be the case if, for instance, the computation the variable is used in is never used. Liveness is interesting for resource estimation. In particular, liveness information can be used to determine the minimal number of distinct locations required to store all variables. This number is usually much lower than the number of distinct variable names occurring in the program. Both register assignment (§13.4) and spilling (§13.2) use liveness information to deal with the problem of assigning variables to storage locations under different constraints.

The second notion of liveness we introduce in §9.2 is a variant of liveness called *true liveness*. True liveness over-approximates the undecidable semantic notion that a variable influences the computation of the program; this means that, depending on the program, less variables are truely live than live. In particular, if a variable is only used in a computation that itself is never used, then it is not truly live. The complement of true liveness is interesting for optimization: removing all variables that are not truly live preserves program behavior. In section §9.3, we verify a dead variable elimination based on true liveness information.

While we support translation validation for both liveness and true liveness information, LVC uses the verified program analysis described in §15 to compute precise true liveness information. Liveness information is obtained indirectly by the fact that after DVE has been run, the true liveness information of the resulting program (almost) passes as liveness information. Recall that liveness information requires, in general, more variables to be live than true liveness information. After variables that are not truly live have been removed from a program, the true liveness information

Line	IL program	Liveness Annotation		
		IL/F	IL/I	
1	`let b = 3 in`	$\{d\}$	$\{\}$	
2	`let a = 5 in`	$\{b, d\}$	$\{b\}$	
3	`let c = 2*b in`	$\{a, b, d\}$	$\{a, b\}$	
4	`fun f (x,y) =`	$\{b, c, d\}$	$\{b, c\}$	Globals of f: $\{b\}$
5	`  let z := 2*x in`	$\{b, x, y\}$	$\{b, x, y\}$	Live-ins of f: $\{b, x, y\}$
6	`  x+b in`	$\{b, x, z\}$	$\{b, x, z\}$	
7	`fun g () = d in`	$\{c, d\}$	$\{b, c\}$	Globals and live-ins
8	`f(c,7)`	$\{c\}$	$\{b, c\}$	of g: $\{d\}$

Figure 9.1: A program together with sets of live variables. In each line, the sets correspond to the live variables *before* the respective statement.

(almost) satisfies the additional requirements for liveness information. We formally describe this issue in §9.3.2.

9.1 Liveness for IL/I and IL/F

A variable x is *significant* to a program s and a context L, if there is an environment V and a value v such that $(L, V, s) \not\sim (L, V[x \mapsto v], s)$. Significance is not decidable, as it is a non-trivial semantic property. As a first approximation, we can think of liveness as an approximation of significance that allows to decide whether a variable is live at a certain program point.

We use Figure 9.1 to explain the notion of liveness informally. The variable a in line 2 of Figure 9.1 is considered live in the statement following its definition in line 3, even though a is not used in the program. Similarly, z is live in line 7. Our version of liveness always requires a variable to be live in the statement following its definition. We do this to ensure that the number of distinct locations required in a program is always upper-bounded by the size of the largest set of live variables, a property we will leverage during register allocation in §13.4. In Figure 9.1 a and b must be stored in different locations, as b would get overwritten in line 2 otherwise. The presence of the live set $\{a, b\}$ in line 3 reflects that two distinct locations are required to hold the live values at that program point. For a similar reason, the parameter y is considered live in the function body of f in line 5. When f is applied, any variable sharing the location with y would get over-written, hence y must have a distinct location even though it is never read in the function body. We introduce two important notions related to the live variables for functions:

Definition 9.1 ▪ Live-ins and Globals
The **live-ins** of a function are the variables live in the first line of the function body. The **globals** of a function are the live-ins without the parameters.

For example, the live-ins of f in Figure 9.1 are $\{b, x, y\}$ and the globals of f are $\{b\}$.

9.1.1 Different Notions of Liveness for IL/I and IL/F

Function definitions are treated differently depending on whether we are interested in liveness with respect to IL/I or IL/F. In IL/F, the globals are read at the function definition, and stored in the closure, and restored after function application. This means that the globals are not live between in the function definition and the function application, as shown in Figure 9.1, where $\{b\}$ is not live in line 8 with respect to IL/F. In IL/I on the other hand, functions do not have closures, and hence the globals must be considered live between the function definition and the function application. In Figure 9.1, b is live in line 8 with respect to IL/I. This means that also the live-ins (and hence the globals) may differ between the two notions of liveness. On the other hand, the global d of the function g, which is never called, is live with respect to IL/F in line 7, but never live with respect to IL/I. This also shows that IL/I liveness is sensitive to reachability information, which we discuss in §11.

9.1.2 Inductive Definition of the Liveness Judgment

We specify soundness of liveness information with the judgment **live**, which is inductively defined by the rules shown in Figure 9.2 and has the following form:

$$\mathbb{\Lambda} \vdash \mathbf{live}_p\ s : \mathbf{X} \quad \text{where} \quad \begin{array}{ll} \mathbb{\Lambda} : \mathit{context}\,(\mathsf{set}\ \mathcal{V}) & \text{function globals} \\ \mathbf{X} : \mathsf{Ann}\,(\mathsf{set}\ \mathcal{V}) & \text{liveness annotation} \\ \mathrm{s} : \mathsf{Exp} & \text{program} \\ \mathrm{p} \in \{\mathrm{I}, \mathrm{F}\} & \text{interpretation} \end{array}$$

The context $\mathbb{\Lambda}$ records the globals of at least the functions names that occur free in s. The predicate $\mathbb{\Lambda} \vdash \mathbf{live}_p\ s : \mathbf{X}$ can be read as follows:

> $\mathbf{X}$ *is a liveness annotation that annotates each program point in* s *with at least the variables live at that program point with respect to* $\mathbb{IL}$ *interpretation* p *and in any context where functions have the globals* $\mathbb{\Lambda}$.

Definition 9.2 ▪ Function Globals Context
Throughout this thesis, $\mathbb{\Lambda}$ always denotes function globals context, i.e. a context that records the globals of the functions in the context.

Definition 9.3 Throughout this thesis, $\mathbf{X}$ always denotes a liveness annotation, that is, an annotation tree of type Ann set $\mathcal{V}$ as defined in §4.8.

$$\textsc{Live-Op}\quad \frac{\mathsf{fv}\,\eta \subseteq X \qquad X' \setminus \{x\} \subseteq X \qquad x \in X' \qquad \mathbb{A} \vdash \mathbf{live}_p\, s : X'}{\mathbb{A} \vdash \mathbf{live}_p\, \mathtt{let}\, x = \eta\, \mathtt{in}\, s : X}$$

$$\textsc{Live-Exp}\quad \frac{\mathsf{fv}\, e \subseteq X}{\mathbb{A} \vdash \mathbf{live}_p\, e : X} \qquad \textsc{Live-App}\quad \frac{\mathsf{fv}\, \overline{e} \subseteq X \qquad \mathrm{I} \in p \to \mathbb{A}_f \subseteq X}{\mathbb{A} \vdash \mathbf{live}_p\, f\, \overline{e} : X}$$

$$\textsc{Live-Cond}\quad \frac{\mathsf{fv}\, e \subseteq X \qquad X_1 \cup X_2 \subseteq X \qquad \mathbb{A} \vdash \mathbf{live}_p\, s_1 : X_1 \qquad \mathbb{A} \vdash \mathbf{live}_p\, s_2 : X_2}{\mathbb{A} \vdash \mathbf{live}_p\, \mathtt{if}\, e\, \mathtt{then}\, s_1\, \mathtt{else}\, s_2 : X}$$

$$\textsc{Live-Fun}\quad \frac{\forall g,\ \overline{f : X \setminus \overline{x}}; \mathbb{A} \vdash \mathbf{live}_p\, s_g : X_g \qquad \forall g,\ \overline{x}_g \subseteq X_g \wedge \overline{x}_g \text{ duplicate-free} \qquad \overline{f : X \setminus \overline{x}}; \mathbb{A} \vdash \mathbf{live}_p\, t : X_1 \qquad X_1 \subseteq X_2 \qquad \mathrm{F} \in p \to \forall g,\ X_g \subseteq X_2}{\mathbb{A} \vdash \mathbf{live}_p\, \mathtt{fun}\, \overline{f\, \overline{x} = s : \overline{X}}\, \mathtt{in}\, t : X_2}$$

Figure 9.2: Liveness: An approximation of the significant variables for IL/I.

Recall from the discussion in §4.8 that while analysis information is organized in annotations trees like $\mathbf{X}$, we want to keep the notational overhead of destructuring these tree low. For this reason, we use the notation $\mathbb{A} \vdash \mathbf{live}_p\, s : X$ where X is of type $\mathsf{set}\, \mathcal{V}$ and the annotation tree is left implicit. In the definition of the rules of the liveness judgment in Figure 9.2, for example, we annotate live sets X at each program point, but leave the annotation tree $\mathbf{X}$ from which these originate implicit. More information on annotation trees can be found in §4.8.

Description of the Rules

Live-Op defines the derivation rule for $\mathbb{A} \vdash \mathbf{live}_p\, \mathtt{let}\, x = \eta\, \mathtt{in}\, s : \mathbf{X}$. Note that for the sake of simpler notation, we do not mention $\mathbf{X}$ in the definition of Live-Op, but $\mathtt{let}\, x = \eta\, \mathtt{in}\, s : X$ to indicate that $X = [\mathbf{X}]$, i.e. that X is the top-level annotation in $\mathbf{X}$. Similarily, we write $s : X'$ to indicate that X' is the top-level annotation of the annotation tree belonging to s; this tree is the only direct sub-tree of $\mathbf{X}$. The rule ensures that all variables free in η are live. Every live variable of the continuation s except x must be live at the assignment. We require x to be live in the continuation, because x is written to and we are ultimately interested in resource estimation.

Live-Cond ensures that the live variables of a conditional contain at least the free variables of the condition, and the variables live in the consequence and alternative.

Live-Exp ensures that for programs consisting of a single expression e at least the free variables of e are live.

Live-App ensures that the free variables of every argument are live. For the IL/I liveness judgment (i.e. if $\mathrm{I} \in p$) we also require that the globals Λ_f of f are live at the call site.

Live-Fun handles function definitions. First note that the annotations of the function bodies $\overline{X}$ are the live-ins, and those are required to contain the function's arguments (top-left premise). We, however, extend the context Λ with the functions' globals $\overline{f : X \setminus \overline{x}}$, which we obtain by removing the arguments $\overline{x}$ from the live-ins X of each function. For the recursion on the function bodies s_g we take the live-ins X_g of the function g as live set. The live variables X_1 of the continuation t must be live at the function definition: $X_1 \subseteq X_2$. For the IL/F liveness judgment (i.e. if $\mathrm{F} \in p$) we also require $X_g \subseteq X_2$, i.e. that the globals X_g of f are live at the function definition. This reflects that in IL/F, at least conceptually, the globals are read at the function definition to construct IL/F closures. For both interpretations, we for every newly defined function g that all parameters are live $\overline{x}_g \subseteq X_g$ and that the parameter list is duplicate free.

Duplication-Free Parameters

The requirement that parameters are duplication free is important for the lowering of parallel moves, which we discuss in §13.5. The reason we add this requirement in the liveness judgment is that during LVC's compilation process, liveness information is obtained from true liveness information after dead variable elimination (see §9.3.2). Dead variable elimination ensures that all parameters are duplicate free, and afterwards all phases maintain the property. Having the requirements in the liveness rules makes it convenient to show that parameters remain duplication free until parallel moves are lowered, where the property is finally required.

9.1.3 Decidability and Translation Validation

Theorem 9.4 ▪ Liveness is Decidable
For all global contexts Λ, and liveness interpretations p, it is efficiently decidable for a given program s and an liveness annotation $\mathbf{X}$ whether $\Lambda \vdash \mathbf{live}_p\ s : \mathbf{X}$ holds.

The proof of Theorem 9.4 is constructive and yields an efficient, extractable decision procedure that can be used to translation-validate the results of a liveness analysis. The decision procedure recursively descends on the program structure, checking the conditions of the appropriate rule in every step. Currently, LVC does not employ translation validation for liveness information, and does not compute liveness information. Instead, LVC uses a verified program analysis to compute true liveness

information (which we describe in §9.2), and uses the results from §9.3.2 to obtain liveness information.

9.1.4 Free Variables, Liveness and Significance on IL/F

The liveness annotation ensures that the free variables of an IL program are always live with respect to IL/F.

Lemma 9.5 If $\mathbb{A} \vdash \mathbf{live_F}\ s : X$ then $\mathsf{fv}\ s \subseteq X$.

Example 9.6 A property similar to Lemma 9.5 does not hold for IL/I. Consider the program in Figure 9.1, where d is a free variable, but not contained in the live set.

Only the free variables are significant to an IL/F program. Recall that we write $V =_X V'$ if V and V' agree on X, that is if $\forall x \in X, Vx = V'x$. We define a relation

$$L \overset{\mathsf{fv}}{=} L'$$

on IL/I contexts L, L' to hold if for all f we have that either both L_f and L'_f are undefined, or $L_f = (V, \overline{x}, s)$ and $L'_f = (V', \overline{x}, s)$ and $V =_{\mathsf{fv}\, s} V'$.

Lemma 9.7 If $L \overset{\mathsf{fv}}{=} L'$ and $V =_{\mathsf{fv}\, s} V'$ then $(L, V, s)_{\mathrm{F}} \overset{1}{\sim}_r (L', V', s)_{\mathrm{F}}$.

Proof. By coinduction. ■

Corollary 9.8 ■ **Liveness Approximates Significance for IL/F**
If $\mathbb{A} \vdash \mathbf{live_F}\ s : X$ and $V =_X V'$ then $(L, V, s)_{\mathrm{F}} \overset{1}{\sim}_r (L, V', s)_{\mathrm{F}}$.

Note that we prove that the two states are in lock-step simulation $\overset{1}{\sim}_r$.

9.1.5 Liveness and Significance for IL/I

We show that the live variables approximate the significant variables. We write $L \models \mathbb{A}$ if a context L satisfies the assumptions $\mathbb{A}$, that is, if for each function f the globals recorded in $\mathbb{A}_f$ do not contain the parameters the function recorded in L_f, and the globals are valid function body in L_f. We formally defined this relation inductively on the context. Recall from §4.3 that a context is a list of groups of named definitions, where each group corresponds to set of mutually recursively defined functions. The following definition is defined inductively on the group structure of the contexts, i.e. the recursion in LiveCtx1 occurs on contexts where a whole group of functions has been removed.

$$\textsc{LiveCtx1}\ \frac{L \models \mathbb{A} \qquad \forall g, \overline{x}_g \,||\, X_g \qquad \forall g, \overline{f : X}; \mathbb{A} \vdash \mathbf{live_I}\ s_g : X_g \cup \overline{x}_g}{\overline{f : (\overline{x}, s)}; L \models \overline{f : X}; \mathbb{A}} \qquad \textsc{LiveCtx2}\ \frac{}{\varnothing \models \varnothing}$$

LiveCtx1 ensures for each function f in $\overline{f}$ that X_g does not contain parameters and that $X_g \cup \overline{x}_g$ is a large enough live set for the function body s under the context $\overline{f : X}; \Lambda$.

We can now formally state the soundness of the live predicate for IL/I. We prove that if $\Lambda \vdash \mathbf{live}_p\, s : X$, then X contains at least the significant variables of s in every context L that satisfies the assumptions Λ.

Theorem 9.9 For every program s, if $\Lambda \vdash \mathbf{live}_\mathrm{I}\, s : X$ and $L \models \Lambda$ and $V =_X V'$, then for all r we have $(L, V, s)_\mathrm{I} \overset{1}{\sim}_r (L, V', s)_\mathrm{I}$.

Note that we prove that the two states are in lock-step simulation $\overset{1}{\sim}_r$.

9.1.6 Minimal Live Sets and Live Set Annotations

When given the liveness annotations at function definitions for a program s, minimal live sets for all other program points can be uniquely determined by a bottom-up traversal. This means that to provide liveness information, it generally suffices to give the liveness annotations at function definitions. We will use this fact in §13.2, when we have to recompute liveness information: We will only describe how to obtain the new liveness information at function definitions, as liveness information on other program points is then uniquely determined.

9.2 True Liveness for IL/I and IL/F

A variable x is *truly significant* to a program s and a context L, if there is an environment V and values v, v' such that $(L, V[x \mapsto v], s) \not\sim (L, V[x \mapsto v'], s)$. True significance is not decidable, as it is a non-trivial semantic property. We will define the notion of truly live variables that over-approximate the set of truly significant variables. We revisit the example from Figure 9.1, and give the corresponding true live sets in Figure 9.3. There are two differences to the live-sets from Figure 9.1: First, the variables a and z are never live in the program, because they are never used. Second, the parameter y is never live in the program, because it is never used.

9.2.1 Inductive Predicate

We specify soundness of true liveness information with the judgment **tlive**, which is inductively defined by the rules shown in Figure 9.2 and has the following form:

$$\Lambda \mid \zeta \vdash \mathbf{tlive}_p\, s : \mathbf{X} \quad \text{where}$$

Λ : *context* (set $\mathcal{V}$)		function liveness
ζ : *context* (list $\mathcal{V}$)		function parameters
$\mathbf{X}$: set $\mathcal{V}$		live variables
s : Exp		program
p $\in$ {I, F}		interpretation

Line	IL program	True Liveness Annot.		
		IL/F	IL/I	
1	`let b = 3 in`	$\{d\}$	$\{\}$	
2	`let a = 5 in`	$\{b, d\}$	$\{b\}$	
3	`let c = 2*b in`	$\{b, d\}$	$\{b\}$	
4	`fun f (x,y) =`	$\{b, c, d\}$	$\{b, c\}$	Globals of f: $\{b\}$
5	`  let z := 2*x in`	$\{b, x\}$	$\{b, x\}$	Live-ins of f: $\{b, x\}$
6	`  x+b in`	$\{b, x\}$	$\{b, x\}$	
7	`fun g () = d in`	$\{c, d\}$	$\{b, c\}$	
8	`f(c,7)`	$\{c\}$	$\{b, c\}$	of g: $\{d\}$

Figure 9.3: A program together with sets of truly live variables. In each line, the sets corresponds to the truly live variables *before* the respective statement.

The predicate $\Lambda \mid \zeta \vdash \mathbf{tlive}_p\ s : \mathbf{X}$ can be read as follows:

> $\mathbf{X}$ *is an annotation tree annotating every program point in* s *with at least the truely live variables at that program point with respect to IL interpretation* p *and for any function context satisfying the liveness assumptions* Λ *and the parameters* ζ.

Definition 9.10 ▪ Function Live-Ins Context
Throughout this thesis, Λ always denotes a function live-in context, that is, a context that maps function names to the corresponding live-ins.

Definition 9.11 ▪ Function Parameter Context
Throughout this thesis, Λ always denotes a function parameter context, that is, a context that maps function names to the corresponding function's parameters.

Given Λ and ζ and Definition 9.1, a function globals context can be reconstructed according to the following equation (note the point-wise lifting):

$$\Lambda = \Lambda \setminus \zeta$$

Recall from the liveness section that the meta-variable X always denotes a set of variables.

Description of the Rules

TLive-Op ensures that the set of variables X' that are live at s is contained in the set of variable live before the let-statement, except the variable x the let defines. The free variables of e are only included in the live set X, if x itself is live.

$$\textsc{TLive-Op}\quad \frac{X' \setminus \{x\} \subseteq X \qquad x \in X' \to \mathsf{fv}\,(e) \subseteq X \qquad \zeta \mid \Lambda \vdash \mathbf{tlive}_p\, s : X'}{\zeta \mid \Lambda \vdash \mathbf{tlive}_p\, \mathtt{let}\, x = e\, \mathtt{in}\, s : X}$$

$$\textsc{TLive-Exp}\quad \frac{\mathsf{fv}\,(e) \subseteq X}{\zeta \mid \Lambda \vdash \mathbf{tlive}_p\, e : X} \qquad \textsc{TLive-Call}\quad \frac{X' \setminus \{x\} \subseteq X \qquad \mathsf{fv}\,(\overline{e}) \subseteq X \qquad \zeta \mid \Lambda \vdash \mathbf{tlive}_p\, s : X'}{\zeta \mid \Lambda \vdash \mathbf{tlive}_p\, \mathtt{let}\, x = \alpha\, \overline{e}\, \mathtt{in}\, s : X}$$

$$\textsc{TLive-App}\quad \frac{|\overline{x}| = |\overline{e}| \qquad \forall i, x_i \in \Lambda_f \to \mathsf{fv}\,(e_i) \subseteq X \qquad p = \mathrm{I} \to \Lambda_f \setminus \zeta_f \subseteq X}{\zeta \mid \Lambda \vdash \mathbf{tlive}_p\, f\, \overline{e} : X}$$

$$\textsc{TLive-Cond}\quad \frac{\begin{array}{c}\mathsf{fv}\,(e) \subseteq X \\ X_1 \cup X_2 \subseteq X \qquad \zeta \mid \Lambda \vdash \mathbf{tlive}_p\, s_1 : X_1 \qquad \zeta \mid \Lambda \vdash \mathbf{tlive}_p\, s_2 : X_2\end{array}}{\zeta \mid \Lambda \vdash \mathbf{tlive}_p\, \mathtt{if}\, e\, \mathtt{then}\, s_1\, \mathtt{else}\, s_2 : X}$$

$$\textsc{TLive-Fun}\quad \frac{\begin{array}{c}\overline{f : \overline{x}}; \zeta \mid \overline{f : Y}; \Lambda \vdash \mathbf{tlive}_p\, t : X_1 \qquad \forall g,\, \overline{f : \overline{x}}; \zeta \mid \overline{f : Y}; \Lambda \vdash \mathbf{tlive}_p\, s_g : X_g \\ X_1 \subseteq X_2 \qquad p = \mathrm{F} \to \forall g, X_g \setminus \overline{x}_g \subseteq X_2\end{array}}{\zeta \mid \Lambda \vdash \mathbf{tlive}_p\, \mathtt{fun}\, \overline{f\, \overline{x} = s : X}\, \mathtt{in}\, t : X_2}$$

Figure 9.4: Definition of the predicate $\zeta \mid \Lambda \vdash \mathbf{tlive}_p\, s : X$ for true liveness. The context ζ : *context* (list $\mathcal{V}$) contains parameters of functions, Λ : *context* (set $\mathcal{V}$) contains the true live-ins of the function body, s is a program annotated with live-in annotations, and X is the set of true live variables.

TLive-Call is similar to TLive-Op, but always requires the free variables of $\overline{e}$ to be live, as external events cannot be removed, even if their result is unused.

TLive-Exp requires the free variables of $\overline{e}$ to be live.

TLive-Cond ensures that the live variables at a conditional contain the live variables of both the consequence and the alternative.

TLive-App requires that whenever a parameter x_i is in the live set of the function body X_f, then the free variables of the corresponding argument expression e_i are live at the application. We require that the length of the argument vector $|\overline{e}|$ agrees with the length of the parameter vector $\overline{x}$. For the IL/I liveness judgment (i.e. if $p = \mathrm{I}$) we also require that the globals $\Lambda_f \setminus \zeta_f$ of f are live at the call site.

TLive-Fun records for each f the live-in annotation X_f in the context Λ. The live

variables X_1 of the continuation t must be live at the function definition: $X_1 \subseteq X_2$. For the IL/F liveness judgment (i.e. if $p = \mathrm{F}$) we also require that the globals $X_f \setminus \overline{x}_f$ of f are live at the function definition. This reflects that that the function definition globals are read to construct IL/F closures.

9.2.2 Decidability and Translation Validation

Theorem 9.12 ▪ True Liveness is Decidable
For all global contexts $\mathbb{A}$, and liveness interpretations p, it is efficiently decidable for a given program s and an liveness annotation $\mathbf{X}$ whether $\mathbb{A} \vdash \mathbf{live}_p\ s : \mathbf{X}$ holds.

The proof of Theorem 9.12 is constructive and yields an efficient, extractable decision procedure. The decision procedure recursively descends on the program structure, checking the conditions of the appropriate rule in every step. Currently, LVC does not employ translation validation for true liveness information, but uses a verified program analysis we describe in §15.

9.3 Dead Variable Elimination

The idea for a dead variable elimination (DVE) according to true liveness information is simple: The transformation traverses the program, and removes variable bindings if the variable is not in the true live set of the continuation. Similarily, parameters, and all corresponding arguments of applications, that are not in the true live-ins of the respective function body are removed.

We realize dead variable elimination (DVE) with the recursive function dve defined in Figure 9.5. The recursive procedure descends through the program, removes unused let-bindings, and filters parameter and argument lists according to the liveness information for function bodies.

Note that dve takes four formal parameters. The last parameter is the true liveness annotation. In the definition of dve in Figure 9.5, we do not explicitly take the 4th parameter to avoid destructuring of the tree, but pretend that the annotations are inline in the program syntax (cf. §4.8.2).

9.3.1 Correctness for IL/I and IL/F

We show the correctness for dead variable elimination with the inductive correctness method we developed in §8. The correctness proof we show is for IL/I, but a very similar result also holds for IL/F. The formal development contains both proofs.

$$
\begin{aligned}
\mathsf{dve} \;:\; \mathsf{list}\,\mathcal{V} \to \mathsf{list}\,(\mathsf{set}\,\mathcal{V}) \to \mathsf{Exp} \to \mathsf{Ann}\,(\mathsf{set}\,\mathcal{V}) \to \mathsf{Exp}&\\
\mathsf{dve}\,\zeta\,\Lambda\,(\texttt{let}\;x = e\;\texttt{in}\;(s : X))\;x &= \mathsf{let}\;s' = \mathsf{dve}\,\zeta\,\Lambda\,s\;\mathsf{in}\\
&\quad\; \mathsf{if}\;x \in X\;\mathsf{then}\;\texttt{let}\;x = e\;\texttt{in}\;s'\\
&\quad\; \mathsf{else}\;s'\\
\mathsf{dve}\,\zeta\,\Lambda\,(\texttt{let}\;x = \alpha\,\overline{e}\;\texttt{in}\;s) &= \texttt{let}\;x = \alpha\,\overline{e}\;\texttt{in}\;(\mathsf{dve}\,\zeta\,\Lambda\,s)\\
\mathsf{dve}\,\zeta\,\Lambda\,(\texttt{if}\,e\,\texttt{then}\,s_1\,\texttt{else}\,s_2) &= \texttt{if}\,e\,\texttt{then}\,(\mathsf{dve}\,\zeta\,\Lambda\,s_1)\\
&\quad\; \texttt{else}\,(\mathsf{dve}\,\zeta\,\Lambda\,s_2)\\
\mathsf{dve}\,\zeta\,\Lambda\,(f\,\overline{e}) &= f\,(\mathsf{filterby}\,(\lambda x.x \in \Lambda_f)\,\zeta_f\,\overline{e})\\
\mathsf{dve}\,\zeta\,\Lambda\,e &= e\\
\mathsf{dve}\,\zeta\,\Lambda\,(\texttt{fun}\;\overline{f\,\overline{x}\;=\;s : X}\;\texttt{in}\;t) &= \mathsf{let}\;\Lambda' := \overline{X}; \Lambda\;\mathsf{in}\\
&\quad\; \mathsf{let}\;\zeta' := \overline{\overline{x}}; \zeta\;\mathsf{in}\\
&\quad\; \texttt{fun}\;\overline{f\,(\mathsf{filter}\,(\lambda x.x \in X)\,\overline{x})\;=\;\mathsf{dve}\,\zeta'\,\Lambda'\,s}\\
&\quad\; \texttt{in}\;\mathsf{dve}\,\zeta'\,\Lambda'\,t
\end{aligned}
$$

Figure 9.5: Definition of Dead Variable Elimination. Note that while we give the 4th parameter in the type of dve, in the defining equations we pretend that the liveness annotations are available inline in the program syntax (cf. §4.8.2). In this way, the destructuring of the annotation tree for the recursive call is implicitly handled by the destructuring of the program syntax. We only give the live sets if they occur on the right-hand side of the defining equation. The let-case relies on the live set X of the continuation s. The case for function definition relies on the live sets $\overline{X}$ of the function bodies $\overline{x}$.

Definition 9.13 We define the proof relation $\mathcal{P}_{\mathsf{dve}}$ where

$$
\begin{aligned}
A &:= \mathsf{list}\,\mathcal{V} \times \mathsf{set}\,\mathcal{V}\\
\mathit{Param}\,(\overline{x}, X)\,\overline{y}\,\overline{y'} &:= \overline{x} = \overline{y} \wedge \overline{y'} = \mathsf{filter}\,(\lambda x.x \in X)\,\overline{y}\\
\mathit{Arg}\,V\,V'\,(\overline{x}, X)\,\overline{v}\,\overline{v'} &:= \overline{v'} = \mathsf{filterby}\,(\lambda(x).x \in X)\,\overline{x}\,\overline{v}\\
&\quad \wedge |\overline{x}| = |\overline{v}|\\
\mathit{Idx}\,\_\,f\,f' &:= f = f'
\end{aligned}
$$

Lemma 9.14 If $\mathit{dom}\,F = \mathit{dom}\,\Lambda = \mathit{dom}\,F'$ then we have $F \parallel \Lambda; \Lambda' \parallel_{\mathcal{P}_{\mathsf{dve}}} F'$.

Lemma 9.15 Let $F = \overline{f\,\overline{x}\;=\;s : X}$ and let F' be such that $|F| = |F'|$ and for all i

$$F'_i = (\mathsf{filter}\,(\lambda x.x \in X_i)\,\overline{x}_i, \mathsf{dve}\,(\overline{\overline{x}}; \zeta)\,(\overline{X}; \Lambda)\,s_i)$$

Then $\mathit{Param}\,(\overline{f : (\overline{x}, X)})\,F\,F'$.

We are now ready to show the correctness theorem. Recall that we write $V =_X V'$ if V and V' agree on the values of the variables in the set X.

Theorem 9.16 Let $\zeta \mid \Lambda \vdash \mathbf{tlive_I}\ s : \mathbf{X}$ and $L \gtrsim_r L' :^{\mathcal{P}_{\mathsf{dve}}} \mathsf{zip}\,\zeta\,\Lambda$ and $V =_X V'$. Then:

$$(L, V, s)_{\mathrm{I}} \gtrsim_r (L', V', \mathsf{dve}\,\zeta\,\Lambda\,s)_{\mathrm{I}}.$$

Proof. Induction on s and in each case inversion of **tlive**.

- The case for let-call follows from Sim-Let-Call and the inductive hypothesis.
- In the case for let op, we do a case analysis in $x \in X$.
  - If $x \in X$, the case follows from Sim-Let-Op with the fact that $[\![e]\!]\,V = [\![e]\!]\,V'$ because V and V' agree on $\mathsf{fv}\,(e)$.
  - If $x \notin X$, case analysis on $[\![e]\!]\,V$.
    - If $[\![e]\!]\,V = \bot$, the left side is stuck (Sim-Error).
    - If $[\![e]\!]\,V = v$, we use Sim-Expansion-Closed to reduce the left side one step and are done by the inductive hypothesis with the observation that $V[x \mapsto v]$ still agrees with V' on the X because $x \notin X$.
- The case for the conditional follows by Sim-Cond from Lemma 8.1 and the inductive hypotheses.
- The case for application follows from $L \sim_r L' :^{\mathcal{P}_{\mathsf{dve}}} \mathsf{zip}\,\zeta\,\Lambda$ with Lemma 8.21, after discharging premises. The relation $\mathit{Idx}\,(\overline{x}, X)\,f\,f$ holds by definition. Argument evaluation agrees because

$$\begin{aligned}&[\![\mathsf{filterby}\,(\lambda x.x \in X_f)\,\overline{x}\,\overline{e}]\!]\,V'\\ &= \mathsf{filterby}\,(\lambda x.x \in X_f)\,\overline{x}\,([\![\overline{e}]\!]\,V)\end{aligned}$$

  since we already know that $[\![\overline{e}]\!]\,V \neq \bot$ and V and V' agree on the live variables.
- The case for a return expression is trivial, since the return expressions are identical and environments agree on the live variables.
- In the function definition case, we have F' such that $|F'| = |F|$ and for all i

$$F'_i = (\mathsf{filter}\,(\lambda x.x \in X_i)\,\overline{x}_i, \mathsf{dve}\,(\overline{\overline{x}}; \zeta)\,(\overline{X}; \Lambda)\,s_i)$$

  We apply Lemma 8.19 and have to discharge premises. The second premise holds by assumption, the third is the inductive hypothesis. The first two requirements of the first premise are Lemma 9.14 and Lemma 9.15. It remains to show from

$$(\!|F|\!)_{\mathrm{I}}; L \sim_r (\!|F'|\!)_{\mathrm{I}}; L' :^{\mathcal{P}} (\overline{(\overline{x}, X)}; \Lambda) \qquad (*)$$

that

$$\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\ F\ F'\ (\overline{(\overline{x}, X)}; \Lambda) \subseteq\ \sim_r$$

After unfolding Bdy (Definition 8.12), we get that $Idx\,(\overline{x}, X)\ f\ f'$ and $F_f = (\overline{x}, s)$ and $F'_{f'} = (\overline{x}', s')$ and $Arg\,V\,V'\,(\overline{x}, X)\,\overline{v}\,\overline{v'}$. And after further unfolding we get $\overline{x}' = \mathsf{filter}\,(\lambda x.x \in X)\,\overline{x}$ and $\overline{v}' = \mathsf{filterby}\,(\lambda(x).x \in X)\,\overline{x}\,\overline{v}$. We have to show that

$$\begin{aligned}&(\!|F|\!)_{\mathrm{I}}; L, V[\overline{x} \mapsto v], s)\\ \sim_r\ &(\!|F'|\!)_{\mathrm{I}}; L', V'[\overline{x}' \mapsto \overline{v'}], \mathsf{dve}\,(\overline{\overline{x}}; \zeta)\,(\overline{X}; \Lambda)\,s)\end{aligned}$$

Inductive hypothesis provides the latter. Its premises are discharged by $(*)$ and the observation that the updated environments still agree on the live variables. ■

9.3.2 Liveness after Dead Variable Elimination

DVE operates with respect to true liveness information. This true liveness information can be adapted with minor changes to the program after DVE. Figure 9.6 shows how the liveness information from the original program is transformed by the function dve_live. The function dve_live takes as first argument a set of variables, which will always be added to the top-level live set it is generating. For better readability, we depict the second argument (which really is the program and its liveness annotation tree) in type-assignment style. The function returns an annotation tree with constructors from §4.8.1, which matches in shape the result of the DVE transformation.

The main property of dve_live is that it preserves the live-sets from the original liveness information.

Lemma 9.17 If $\zeta \mid \Lambda \vdash \mathbf{tlive}_p\ s : \mathbf{X}$ then $G \cup [\mathbf{X}] = [\mathsf{dve\_live}\,G\,s\,\mathbf{X}]$.

The following theorem states that the result of dve_live is sound liveness information for the transformed program with respect to the liveness judgment (not the true liveness judgment).

Theorem 9.18 If $\zeta \mid \Lambda \vdash \mathbf{tlive}_p\ s : \mathbf{X}$ then

$$\Lambda \setminus \zeta \vdash \mathbf{live}_p\ (\mathsf{dve}\,\zeta\,\Lambda\,s\,\mathbf{X}) : (\mathsf{dve\_live}\,G\,s\,\mathbf{X})$$

The definition of dve_live together with Lemma 9.17 and Theorem 9.18 allow some observations. First, the live-ins after DVE are exactly the live-ins before DVE, as G is the empty set in the corresponding recursive call in the defining equation of dve_live for function definitions. Since liveness requires all parameters to be live, this means that DVE is *effective* in the sense that all dead parameters are removed.

$$
\begin{aligned}
\mathsf{dve\_live} \;:\; \mathsf{set}\,\mathcal{V} \to \mathsf{Exp} \to \mathsf{Ann}\,(\mathsf{set}\,\mathcal{V}) \to \mathsf{Ann}\,(\mathsf{set}\,\mathcal{V}) \\
\mathsf{dve\_live}\,G\,(\texttt{let}\;x = e\;\texttt{in}\,(s : X)) &= \mathsf{if}\;x \in X\;\mathsf{then}\;G \cup X \cdot \mathsf{dve\_live}\,\emptyset\,s \\
&\quad\;\mathsf{else}\;\mathsf{dve\_live}\,(G \cup X)\,s \\
\mathsf{dve\_live}\,G\,(\texttt{let}\;x = \alpha\,\overline{e}\;\texttt{in}\;s : X) &= G \cup X \cdot \mathsf{dve\_live}\,\{x\}\,sa \\
\mathsf{dve\_live}\,G\,(\texttt{if}\;e\;\texttt{then}\;s_1\;\texttt{else}\;s_2 : X) &= G \cup X \cdot (\mathsf{dve\_live}\,\emptyset\,s_1), (\mathsf{dve\_live}\,\emptyset\,s_2) \\
\mathsf{dve\_live}\,G\,(f\,\overline{e} : X) &= G \cup X \\
\mathsf{dve\_live}\,G\,(e : X) &= G \cup X \\
\mathsf{dve\_live}\,G\,(\texttt{fun}\;\overline{f\,\overline{x} = s}\;\texttt{in}\;t : X) &= G \cup X \cdot \overline{\mathsf{dve\_live}\,\emptyset\,s}, \mathsf{dve\_live}\,\emptyset\,t
\end{aligned}
$$

Figure 9.6: Definition of liveness transformation accompanying DVE. We display liveness annotation (4th parameter) inline in the program as described in §4.8.2.

Second, G is also empty in the recursive call for variable definitions. Together with the soundness requirements for liveness, we can conclude that DVE is also effective in the sense that all dead variables are removed.

The recursive call for dead variable definitions extends the G set with the live variables X at the dead definition. This might seem curious, because if x is dead, the live set X must also be the live set at the consecutive statement s. However, we defined true liveness to allow all post-fixed points, i.e. we do not know that true liveness information is minimal. For this reason we must make sure to include X in the next live set, as it might contain variables that are claimed live because the true liveness information was not minimal. For minimal true liveness information, however, adding X does not add variables to the next live set.

10 Coherence

This section introduces the main tool behind LVC's approach to SSA: coherence. Put simply, coherence allows to rename apart an IL/I program. In Example 4.4 we saw that in general renaming apart does not preserve IL/I semantics. Coherence will ultimately allows us to rename apart IL/I programs, which is one way to establishes the static single assignment property for IL/I programs. We will see, however, that coherence provides for a weaker criterion that does not require a renamed apart program, but still provides the advantages of SSA.

We start by defining semantically what it means that two programs have the same semantic interpretation. Then, building on liveness information, we give the inductive definition of coherence and show that it is sufficient for invariance. Finally, we discuss what it means to establish coherence.

10.1 Invariance

We call a program *invariant* if the functional and the imperative interpretation coincide.

Definition 10.1 ▪ Invariance
A closed program s is invariant if

$$\forall V,\ (\varnothing, V, s)_{\mathrm{F}} \sim (\varnothing, V, s)_{\mathrm{I}}.$$

Invariance is a non-trivial semantic property, and hence undecidable. We develop a syntactic, efficiently decidable criterion sufficient for invariance, which we call coherence. Coherence is a property of a IL program, that is, a property of IL syntax. If an IL program is coherent, the semantic interpretations of IL/F and IL/I coincide.

Coherence is based on the observation that some IL programs do not really depend on information from the closure. Assume $Lf = (V', \overline{x}, s)$ and consider the following IL reduction according to rule ILF-App:

$$(L, V, f\,\overline{e}) \quad \longrightarrow_{\mathrm{F}} \quad (L^f, V'[\overline{x} \mapsto \overline{v}], s)$$

If V agrees with V' on all variables X that are live in s, then the configuration could have equivalently reduced to $(L^f, V[\overline{x} \mapsto \overline{v}], s)$:

$$(L, V, f\,\overline{e}) \quad \longrightarrow_{\mathrm{F}} \quad (L^f, V[\overline{x} \mapsto \overline{v}], s)$$

This reduction does not require the closure V' and is similar to the rule ILI-App. Coherence is a syntactic criterion that ensures that at every at every function application, V and V' agree on a the live set X of the function body.

Example 10.2 ▪ A Coherent Program
In the following programs, the set of globals of f is $\{x\}$. The program on the left is not invariant, while the program on the right is coherent.

```
1 let x = 7 in
2 fun f () = x in
3 let x = 5 in f ()
```

```
1 let x = 7 in
2 fun f () = x in
3 let y = 5 in f ()
```

In the program on the left in line 3, the value of x is 5 and disagrees with the value of x in the closure of f. In the program on the right, x was not redefined, hence both IL and IL/I will compute 7. We say a function f is *available* as long as none of f's globals were redefined. The inductive definition of coherence ensures only available functions are applied.

10.2 Inductive Predicate

We specify coherence with the judgment **coh**, which is inductively defined by the rules shown in Figure 10.1 and has the following form:

$$\mathbb{A} \vdash \mathbf{coh}\ s : \mathbf{X} \quad \text{where} \quad \begin{array}{lll} \mathbb{A} : \mathit{context}\,(\mathsf{set}\,\mathcal{V}) & & \text{function globals} \\ \mathbf{X} : \mathsf{Ann}\,(\mathsf{set}\,\mathcal{V}) & & \text{liveness annotation} \\ \mathsf{s} : \mathsf{Exp} & & \text{program} \end{array}$$

The coherence judgment uses liveness information, in particular the globals of functions. $\mathbb{A}$ is a globals context similar to the one in the liveness judgment. We exploit that contexts realize a partial mapping, and maintain the invariant that $\mathbb{A}$ maps only *available* functions to their globals, and all other functions to $\bot$. The inductive definition given in Figure 10.1 ensures that only available functions are applied.

10.2.1 Description of the Rules

Coh-Op deals with binding a variable x. Every function that has x as a global (i.e. $x \in \mathbb{A}f$) becomes unavailable, and must be removed from $\mathbb{A}$. We write $\lfloor \mathbb{A} \rfloor_X$ to

$$\textsc{Coh-Op}\ \frac{\lfloor \mathbb{A} \rfloor_{X \setminus \{x\}} \vdash \mathbf{coh}\ s}{\mathbb{A} \vdash \mathbf{coh}\ \mathtt{let}\ x = \eta\ \mathtt{in}\ (s : X)} \qquad \textsc{Coh-Exp}\ \frac{}{\mathbb{A} \vdash \mathbf{coh}\ e} \qquad \textsc{Coh-App}\ \frac{\mathbb{A} f \neq \bot}{\mathbb{A} \vdash \mathbf{coh}\ f\ \overline{y}}$$

$$\textsc{Coh-Cond}\ \frac{\mathbb{A} \vdash \mathbf{coh}\ s \qquad \mathbb{A} \vdash \mathbf{coh}\ t}{\mathbb{A} \vdash \mathbf{coh}\ \mathtt{if}\ x\ \mathtt{then}\ s\ \mathtt{else}\ t}$$

$$\textsc{Coh-Fun}\ \frac{\overline{f : X \setminus \overline{x}}; \mathbb{A} \vdash \mathbf{coh}\ t \qquad \forall g,\ \lfloor \mathbb{A}; f : X \setminus \overline{x} \rfloor_{X_g \setminus \overline{x}_g} \vdash \mathbf{coh}\ s_g}{\mathbb{A} \vdash \mathbf{coh}\ \mathtt{fun}\ \overline{f\ \overline{x} = s : X}\ \mathtt{in}\ t}$$

Figure 10.1: Coherence: A sufficient and decidable criterion for invariance.

$$\begin{aligned}
\lfloor \emptyset \rfloor_X &= \emptyset \\
\lfloor \mathbb{A}; f : \bot \rfloor_X &= \lfloor \mathbb{A} \rfloor_X; f : \bot \\
\lfloor \mathbb{A}; f : X' \rfloor_X &= \lfloor \mathbb{A} \rfloor_X; f : X' && X' \subseteq X \\
\lfloor \mathbb{A}; f : X' \rfloor_X &= \lfloor \mathbb{A} \rfloor_X; f : \bot && X' \not\subseteq X
\end{aligned}$$

Figure 10.2: Definition of the restr operation on contexts of globals.

remove all definitions from $\mathbb{A}$ that require more globals than X. Trivally, $\lfloor \mathbb{A} \rfloor_{\mathcal{V}} = \mathbb{A}$. To remove all definitions from $\mathbb{A}$ that use x as global, we use $\lfloor \mathbb{A} \rfloor_{X \setminus \{x\}}$, where X is the live set of the continuation s. Formally, the definition of $\lfloor \mathbb{A} \rfloor_X$ in Figure 10.2 exploits the list structure of contexts.

Coh-App ensures only available functions can be applied, since $\mathbb{A}$ maps functions that are not available to $\bot$.

Coh-Fun deals with function definitions. When a function definition is encountered, the live-ins $\overline{X}$ are obtained from the annotation. We want to record the globals, hence we remove the function parameters to extend the context $\mathbb{A}$. In the function body s, only functions that require at most $X \setminus \overline{x}$ as globals are available, so the context is restricted accordingly.

10.2.2 Decidability and Translation Validation

We get an efficient decidability result for the coherence predicate. The result is efficient, because liveness information determines the globals of functions defined in a term s. The proof of Theorem 10.3 is constructive and yields an extractable decision procedure, which could be used to translation validate external tools that establish coherence. At the moment, we do not use any external components to establish coherence.

Theorem 10.3 ▪ Coherence is Decidable
For all $\mathbb{A}$ and s and $\mathbf{X}$, it is decidable whether $\mathbb{A} \vdash \mathbf{coh}\ s : \mathbf{X}$ holds.

10.3 Coherent Programs are Invariant

In this section we show that coherent programs are invariant, that mean, that their imperative and functional interpretations coincide.

10.3.1 Agreement Invariant

Given a configuration $(L, V, t)_{\mathrm{F}}$ such that $Lf = (V', \overline{x}, s)$, the agreement invariant describes a correspondence between the values of variables in the function closure environment V' and the environment V. If the closure of f is available, the closure environment V' agrees with the primary environment V on f's globals X: $V' =_X V$. We write $F, V \models \mathbb{A}$ if $\forall f \in \mathit{dom}\, F \cap \mathit{dom}\, \mathbb{A}$ we have that $V' =_X V$, where $\mathbb{A}f = X$ and $Ff = (V', \overline{x}, s)$.

10.3.2 Context Coherence

Function application continues evaluation with the function body from the closure. Assume $Ff = (V', \overline{x}, s)$ and consider the IL/F reduction:

$$(F, V, f\,\overline{e}) \longrightarrow_{\mathrm{F}} (F^f, V'[\overline{x} \mapsto \overline{v}]a, s)$$

If coherence is to be preserved under reduction, s must be coherent under suitable assumptions. We now define a predicate that to ensure this.

Definition 10.4 We say a globals context $\mathbb{A}$ approximates $\mathbb{A}'$ if whenever $\mathbb{A}f$ is defined, it agrees with $\mathbb{A}'$ and define $\mathbb{A} \preceq \mathbb{A}' :\leftrightarrow \forall f \in \mathit{dom}\, \mathbb{A},\ \mathbb{A}f = \mathbb{A}'f$.

The context coherence predicate $\mathbb{A} \vdash \mathbf{coh}\, F$ ensures that all function bodies in closures are coherent. It is defined inductively on the context: CohC-Con encodes two requirements: First, the body s_g of each g must be coherent under the context restricted to the globals X_g of g (cf. Coh-Fun). Second, $X_g \cup \overline{x}_g$ must suffice as live variables for the function body s_g under some globals context $\mathbb{A}'$ such that $\overline{f : X}; \mathbb{A}$

$$\textsc{CohC-Emp}\ \frac{}{\varnothing \vdash \mathbf{coh}\ \varnothing} \qquad \textsc{CohC-Con}\ \frac{\overline{f : X}; \mathbb{A} \preceq \mathbb{A}' \quad \mathbb{A} \vdash \mathbf{coh}\ F \qquad \forall g, X_g \neq \bot \to \mathbb{A}' \vdash \mathbf{live_I}\ s_g : X_g \cup \overline{x}_g \ \wedge\ \lfloor \overline{f : X}, \mathbb{A} \rfloor_{X_g} \vdash \mathbf{coh}\ s}{\overline{f : X}; \mathbb{A} \vdash \mathbf{coh}\ \overline{f : (V, \overline{x}, s)}; F}$$

Figure 10.3: Coherence for contexts.

approximates $\mathbb{A}'$. The approximation takes care of the fact that globals from $\mathbb{A}$ may have been replaced by $\bot$ because the corresponding function became unavailable, but for a liveness derivation to be possible, these globals cannot be $\bot$.

Lemma 10.5 Approximation ensures stability under restriction to any set of variables X, because $\lfloor \mathbb{A} \rfloor_X \preceq \mathbb{A}$.

$$\mathbb{A} \vdash \mathbf{coh}\ L \ \to\ \lfloor \mathbb{A} \rfloor_X \vdash \mathbf{coh}\ L.$$

Lemma 10.6 Context coherence is stable under rewinding to any function f:

$$\mathbb{A} \vdash \mathbf{coh}\ L \ \to\ \mathbb{A}^{-f} \vdash \mathbf{coh}\ L^{-f}.$$

10.3.3 Preservation Theorem

Definition 10.7 We define $\mathsf{strip}\,(V, \overline{x}, s) \ = \ (\overline{x}, s)$ and lift strip pointwise to contexts.

Theorem 10.8 ▪ **Coherence implies Invariance**
Let $\mathbb{A} \ \vdash \mathbf{coh}\ s : \mathbf{X}$ and $\mathbb{A} \preceq \mathbb{A}'$ and $\mathbb{A} \vdash \mathbf{coh}\ L$ and $L, V \models \mathbb{A}$ and $\mathbb{A}' \vdash \mathbf{live_I}\ s : \mathbf{X}$. Then:

$$(L, V, s)_{\mathrm{F}} \overset{1}{\sim}_r (\mathsf{strip}\ L, V, s)_{\mathrm{I}}.$$

Proof. The proof proceeds in a lock step fashion since the variable environments and the programs are the same. The let-binding case needs Lemma 10.5 to show that context coherence is preserved. The only other interesting case is function application, where we have to show that

$$(L, V, g\,\overline{e})_{\mathrm{F}} \overset{1}{\sim}_r (\mathsf{strip}\ L, V, g\,\overline{e})_{\mathrm{I}}$$

We can assume $L_g \ = \ (V', \overline{x}, t)$ and $[\![\overline{e}]\!]\, E \ = \ \overline{v}$, because otherwise both sides are stuck and we are done. After reducing both sides one step by applying the appropriate rule of the simulation, we have to show

$$(L^{-g}, V'[\overline{x} \mapsto \overline{v}], g\,\overline{e})_{\mathrm{F}}\ (r \cup \overset{1}{\sim}_r)\ (\mathsf{strip}\ L^{-g}, V[\overline{x} \mapsto \overline{v}], g\,\overline{e})_{\mathrm{I}}$$

From coherence of the application we get $X_g \neq \bot$, and in turn from context coherence we get coherence and soundness of liveness for the function body t. We use the transitivity property from Lemma 6.35 with the soundness result about liveness (Theorem 9.9) which allows us to change variable environments as long as they agree on the live variables. We use the agreement invariant to obtain that $V =_{X_g \setminus \overline{x}} V'$, from which we get that $V[\overline{x} \mapsto \overline{v}]$ agrees with $V'[\overline{x} \mapsto \overline{v}]$ on the live-ins X_g of g. It remains to show

$$(L^{-g}, V'[\overline{x} \mapsto \overline{v}], g\,\overline{e})_{\mathrm{F}}\ (r \cup \overset{1}{\sim}_r)\ (\mathsf{strip}\, L^{-g}, V'[\overline{x} \mapsto \overline{v}], g\,\overline{e})_{\mathrm{I}}$$

which follows from the cohypothesis with the fact that strip commutes with rewinding and Lemma 10.6. ■

Theorem 10.8 reduces the problem of translating between IL/I and IL to the problem of establishing coherence. For the translation from IL to IL/I, it suffices to establish coherence while preserving IL semantics. Theorem 10.8 also explains why we defined invariance on configurations with empty function evironments: Any function environment must satisfy the agreement invariant, which would have complicated the definition of invariance while not providing more insight.

Corollary 10.9 If $\varnothing \vdash \mathbf{coh}\ s : \mathbf{X}$ and $\varnothing \vdash \mathbf{live_I}\ s : \mathbf{X}$ then s is invariant.

10.4 Establishing Coherence

In this section, we discuss the simplest way to establish coherence for an IL program, and foreshadow two important transformations which we will prove correct. Two methods are of interest, one that preserves the program's IL/F semantics and one that preserves the program's IL/I semantics.

10.4.1 Establishing Coherence and Preserving IL/F Semantics

The simplest method to establish coherence while preserving IL/F semantics is α-renaming the program apart. A renamed-apart program is coherent, since every function is always available. The properties of α-conversion ensure semantic equivalence. This approach however, might introduce an excessive amount of variable names, and we discuss a method that uses substantially less names in §13.4, when we discuss register assignment.

10.4.2 Establishing Coherence and Preserving IL/I Semantics

The simplest method to establish coherence while preserving IL/I semantics is introducing all variables occurring in the program as formal parameters and as arguments to every function and every function application. This ensures that the globals are

the empty set for each function, and hence every function is always available. This approach, however, may introduce an excessive amount of parameters to the functions in the program, and we discuss a method that introduces substantially less parameters in §12 when we discuss SSA construction.

11 Reachability and Unreachable Code Elimination

This section introduces the notion of reachability. Reachability approximates an undecidable, semantic notion that is most easily explained by its complement: If a program point can be removed from the program without changing the program's semantics in any context, that program point is unreachable. This definition already hints at the most prominent use-case for reachability information, namely unreachable code elimination (UCE).

Reachability information annotates every program point with a boolean: **true** to mark it reachable, and **false** to mark it unreachable. There are two very useful intuitions about what soundness and completeness for reachability information means:

soundness If a program point is marked reachable, then all of its successors must be marked reachable. The exception to this rule are successors of conditionals if they are ruled out by a constant condition value.

completeness If a program point is marked reachable, then there must be a predecessor that is marked reachable. The exception to this rule is the initial program point, which is always marked reachable.

As reachability is a non-trivial semantic property and hence undecidable, we must settle for an overapproximation of reachability information; and indeed the definition above may mark branches of conditionals reachable, even if they are not. However, we can compute reachability information that is sound and complete in the sense of the definition above, but we postpone the verification of the analysis to §15.4.

We are interested in completeness of a reachability analysis, because we want to show that certain kinds of unreachable code are completely eliminated by UCE. In particular, we will show that UCE removes all functions that do not have a reachable application. This result about Reachability and UCE bridges an important gap between the liveness information for IL/I and IL/F. In Figure 9.1 we saw that the globals of an unreachable function (i.e. a function that is never applied) are live with respect to IL/I, but not live with respect to IL/I. In general, an unreachable function never impacts liveness of the surrounding program in IL/I, while it may impact the

$$\textsc{CallChain-Refl}\ \frac{}{\mathsf{callChain}\ F\ f\ f} \qquad \textsc{CallChain-Step}\ \frac{\mathsf{isCalled}\ s_i\ f_j \qquad \mathsf{callChain}\ \overline{f\ \overline{x}\ =\ s}\ f_j\ g}{\mathsf{callChain}\ \overline{f\ \overline{x}\ =\ s}\ f_i\ g}$$

$$\textsc{IsCalled-Op}\ \frac{\mathsf{isCalled}\ s\ f}{\mathsf{isCalled}\ (\mathtt{let}\ x = \eta\ \mathtt{in}\ s)\ f} \qquad \textsc{IsCalled-App}\ \frac{}{\mathsf{isCalled}\ f\ \overline{e}\ f}$$

$$\textsc{IsCalled-Cond-Left}\ \frac{\mathbf{true} \sqsubseteq \mathsf{cnd}\ e \qquad \mathsf{isCalled}\ s_1\ f}{\mathsf{isCalled}\ (\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2)\ f} \qquad \textsc{IsCalled-Cond-Right}\ \frac{\mathbf{false} \sqsubseteq \mathsf{cnd}\ e \qquad \mathsf{isCalled}\ s_2\ f}{\mathsf{isCalled}\ (\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2)\ f}$$

$$\textsc{IsCalled-Fun}\ \frac{\mathsf{isCalled}\ t\ g \qquad \mathsf{callChain}\ \overline{f\ \overline{x}\ =\ s}\ g\ f}{\mathsf{isCalled}\ (\mathtt{fun}\ \overline{f\ \overline{x}\ =\ s}\ \mathtt{in}\ t)\ f}$$

$$\mathsf{isCalledFrom}\ F\ t\ f\ :=\ \exists f', \mathsf{isCalled}\ t\ f' \wedge \mathsf{callChain}\ F\ f'\ f$$

Figure 11.1: Formal definition of call chain

liveness of the surrounding program in IL/F. At the end of this section we show that in a program without unreachable code, liveness information sound for IL/I's notion of liveness is also sound with respect to IL/F's notion of liveness (but not vice versa). This theorem is critical for switching semantic interpretations during register allocation.

11.1 Static Evaluation of Conditions

We use a static evaluation function $\mathsf{cnd} : \mathsf{Exp} \to \mathbb{B}_\bot^\top$ to statically evaluate the value of condition expressions in conditionals. cnd yields the value of the condition, $\top$ if the value of the condition can be either truth value, or $\bot$ to indicate that neither the consequence nor the alternative is reachable. The latter is the case if, for example, the evaluation of the condition gets stuck. The reachability annotations for a program can achieve both soundness and completeness relative to a static evaluation function cnd.

11.2 Call Chains and Completeness

There are different ways to formulate the completeness requirement for reachability of functions, that is, a condition that ensures that a function is called (up to a static evaluation function like cnd). Consider a mutually recursive function definition $\texttt{fun}\ \overline{f\,\overline{x} = s}\ \texttt{in}\ t$. Intuitively, reachability of some f_i can be justified by a call chain (again, up to a criterion cnd for handling conditionals) from t to f_i with an arbitary but fixed number of calls to other functions from $\overline{f}$ in between. There are other possibilities, but this formulation has the advantage that it allows proofs by induction on the call chain. The formal definition of a call chain is realized by the inductive predicate

$$\mathsf{callChain}\ F\ f\ g$$

given in Figure 11.1. The first rule says that every function reaches itself. The second rule says that f reaches h if there is a g such that the body of f calls g and there is a call chain from g to h. The inductive predicate

$$\mathsf{isCalled}\ s\ f$$

given in Figure 11.1 realizes the notation that a program s calls a function f which appears free in s. Finally, we define

$$\mathsf{isCalledFrom}\ F\ t\ f$$

in Figure 11.1, which formalizes the fact that f is called from t.

11.3 Inductive Reachability Judgment

We define the judgment reach inductively according to the rules given in Figure 11.2 to formalize soundness and completeness of reachability information. The rules are parametric in an index p, which indicates whether soundness, completeness, or both are desired.

$$R \vdash \mathbf{reach}_p\,\{\mathbf{r}\}\,s \quad \text{where} \quad \begin{array}{lll} R & : \mathit{context}\,\mathbb{B} & \text{reachability for functions} \\ s & : \mathsf{Exp} & \text{program} \\ \mathbf{r} & : \mathsf{Ann}\,\mathbb{B} & \text{reachability annotation} \\ p & \subseteq \{\mathrm{S},\mathrm{C}\} & \text{soundness/completeness} \end{array}$$

The parameter p of the predicate can be any subset of $\{\mathrm{S},\mathrm{C}\}$ and controls whether soundness (S), completeness (C), or both are enforced. This flexibility is realized by the relation $\overset{p}{\leftrightarrow} : \mathbb{B} \to \mathbb{B} \to \mathbf{P}$ that is defined as follows:

$$b \overset{p}{\leftrightarrow} b' := (\mathrm{S} \in p \to b \to b') \wedge (\mathrm{C} \in p \to b' \to b)$$

$$\textsc{Reach-Let}\ \frac{b \overset{p}{\leftrightarrow} b' \qquad R \vdash \mathbf{reach}_p\,\{b'\}\ s}{R \vdash \mathbf{reach}_p\,\{b\}\ \mathtt{let}\ x = \eta\ \mathtt{in}\ \{b'\}\ s} \qquad \textsc{Live-Exp}\ \frac{}{R \vdash \mathbf{reach}_p\,\{b\}\ e}$$

$$\textsc{Reach-App}\ \frac{b \to R_f}{R \vdash \mathbf{reach}_p\,\{b\}\ f\ \overline{e}}$$

$$\textsc{Reach-Cond}\ \frac{\begin{array}{c}\mathbf{true} \sqsubseteq \mathsf{cnd}\ e \to b \overset{p}{\leftrightarrow} b_1 \\ \mathbf{false} \sqsubseteq \mathsf{cnd}\ e \to b \overset{p}{\leftrightarrow} b_2 \qquad R \vdash \mathbf{reach}_p\,\{b_1\}\ s_1 \qquad R \vdash \mathbf{reach}_p\,\{b_2\}\ s_2\end{array}}{R \vdash \mathbf{reach}_p\,\{b\}\ \mathtt{if}\ e\ \mathtt{then}\ \{b_1\}\ s_1\ \mathtt{else}\ \{b_2\}\ s_2}$$

$$\textsc{Reach-Fun}\ \frac{\begin{array}{c}\forall g,\ \overline{f : b}; R \vdash \mathbf{reach}_p\,\{b_g\}\ s_g \qquad \overline{f : b}; R \vdash \mathbf{reach}_p\,\{c\}\ t \\ \mathsf{C} \in p \to \forall f, b_f \to b \qquad \mathsf{C} \in p \to \forall f, \mathsf{isCalledFrom}\,(\overline{f\ \overline{x} = s})\,t\,f \qquad a \overset{p}{\leftrightarrow} c\end{array}}{R \vdash \mathbf{reach}_p\,\{a\}\ \mathtt{fun}\ \overline{f\ \overline{x} = \{b\}\ s}\ \mathtt{in}\ \{c\}\ t}$$

Figure 11.2: Definition of the Reachability Predicate $R \vdash \mathbf{reach}_p\,\{\mathbf{r}\}\ s$. The context $R : \mathit{context}\ \mathbb{B}$ contains reachability information for functions, s is a program, and $\mathbf{r}$ is a reachability information annotation. The rules omit the annotation as described in §4.8.2. See Figure 11.1 for the definition of isCalledFrom.

In this way, $\overset{\{S\}}{\leftrightarrow}$ enforces forward propagation of reachability, which corresponds to soundness, and $\overset{\{C\}}{\leftrightarrow}$ enforces backwards propagation of reachability, which corresponds to completeness. The relation $\overset{\{S,C\}}{\leftrightarrow}$ enforces both. Note that Reach-App is not formulated in terms of $\overset{p}{\leftrightarrow}$, but only captures the soundness aspect by requiring that if the application is reachable, then the corresponding function is marked reachable in the context $R : \mathit{context}\ \mathbb{B}$. The completeness aspect is taken care of by the predicate isCalledFrom at function definitions.

11.3.1 Description of the Rules

Reach-Let propagates reachability information through let-bindings: If the let is reachable, then so is its successor.

Reach-App ensures that whenever a function application is reachable, then the function is also reachable.

Reach-Cond evaluates $\mathsf{cnd}\ e$, i.e. it uses the static evaluation function to evaluate

the condition. If cnd e is **false** or undefined ($\bot$) then reachability is not propagated into the consequence s_1. Propagation into the alternative s_2 is treated similarily.

Reach-Fun propagates reachability into t. The context R is extended with the reachability information of the function bodies $\overline{f : b}$. The topmost premise ensures that the reachability information for all function bodies is sound.

11.4 Unreachable Code Elimination

In this section, we verify an optimizations that falls into the category of dead code elimination (DCE): unreachable code elimination (UCE). The correctness proof uses the inductive proof method we developed in §8, and cannot easily be verified using coinduction in Coq because the transformation removes statements (cf. §6.2.4) from the program. The proofs are intentionally detailed to demonstrate the inductive proof method from §8.

UCE eliminates function definitions and branches of conditionals according to given reachability information. Note that we never specified what the annotations are supposed to *mean*. In our setting, analysis information receives its meaning through the transformations it permits. There is, of course, an intuition about the semantic meaning of the reachability annotations, but we do not have to make the intuition formal: The connection between the annotations and the semantics is in the correctness proof of the transformation.

11.4.1 Transformation

In Figure 11.4, we define a function uce that removes all code not marked reachable. If all functions from a mutually recursive function definition are removed, the `fun`-statement is removed, too. Conditionals are removed if the value of the condition can be statically evaluated.

11.4.2 Correctness for IL/I and IL/F

In this section we prove correctness of uce for both IL/I and IL/F. The proofs for both semantics are so similar, that we can just omit IL interpretation subscripts F and I, and present a proof that works out for both semantics.

Definition 11.1 We define the proof relation $\mathcal{P}_{\mathsf{uce}}$ where

$$
\begin{aligned}
A &:= \mathbb{B} \\
Param\ \_\ \overline{x}\ \overline{x}' &:= \overline{x} = \overline{x}' \\
Arg\ V\ V'\ a\ \overline{v}\ \overline{v}' &:= \overline{v} = \overline{v}' \\
Idx\ a\ f\ f' &:= a = \mathbf{true} \wedge f = f'
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{filterby} \;:\; \forall XY, (X \to \mathbb{B}) \to \mathsf{list}\, X \to \mathsf{list}\, Y \to \mathsf{list}\, Y \\
&\mathsf{filterby}\, p\, (x :: x')\, (y :: y') = \\
&\quad \mathsf{if}\; px \;\mathsf{then}\; y :: \mathsf{filterby}\, p\, x'\, y' \;\mathsf{else}\; \mathsf{filterby}\, p\, x'\, y' \\
&\mathsf{filterby}\, p\, \_, \_ = \mathsf{nil} \\
&\mathsf{filter} \;:\; \forall X, (X \to \mathbb{B}) \to \mathsf{list}\, X \to \mathsf{list}\, X \\
&\mathsf{filter}\, p\, x = \mathsf{filterby}\, p\, x\, x
\end{aligned}
$$

Figure 11.3: Definition of filterby.

$$
\begin{aligned}
\mathsf{uce} \;&:\; \mathsf{Ann}\; \mathsf{Exp}\, \mathbb{B} \to \mathsf{Exp} \\
\mathsf{uce}\, (\mathtt{let}\; x = \eta\; \mathtt{in}\; s) \;&=\; \mathtt{let}\; x = \eta\; \mathtt{in}\; (\mathsf{uce}\, s) \\
\mathsf{uce}\, (\mathtt{if}\; e\; \mathtt{then}\; s_1\; \mathtt{else}\; s_2) \;&=\; \mathsf{if}\; [\![e]\!]\emptyset = \mathbf{true}\; \mathsf{then}\; \mathsf{uce}\, s_1 \\
&\qquad \mathsf{else\ if}\; [\![e]\!]\emptyset = \mathbf{false}\; \mathsf{then}\; \mathsf{uce}\, s_2 \\
&\qquad \mathsf{else}\; \mathtt{if}\; e\; \mathtt{then}\; (\mathsf{uce}\, s_1)\; \mathtt{else}\; (\mathsf{uce}\, s_2) \\
\mathsf{uce}\, (f\, \overline{e}) \;&=\; f\, \overline{e} \\
\mathsf{uce}\, e \;&=\; e \\
\mathsf{uce}\, (\mathtt{fun}\; F\; \mathtt{in}\; t) \;&=\; \mathsf{let}\; F' = \mathsf{uceF}\, F\; \mathsf{in} \\
&\qquad \mathsf{if}\; |F'| = 0\; \mathsf{then}\; \mathsf{uce}\, t \\
&\qquad \mathsf{else}\; \mathtt{fun}\; F'\; \mathtt{in}\; (\mathsf{uce}\, t) \\
\mathsf{uceF}\, F \;&=\; \mathsf{let}\; K = \mathsf{filter}\, (\lambda(\overline{x}, \{b\}\; s).\, b)\, F\; \mathsf{in} \\
&\qquad \mathsf{map}\, (\lambda(\overline{x}, s).(\overline{x}, \mathsf{uce}\, s))\, K
\end{aligned}
$$

Figure 11.4: Definition of Unreachable Code Elimination.

Lemma 11.2 If $\mathit{dom}\, R = \mathit{dom}\, F$ then $F \parallel R; R' \parallel_{\mathcal{P}_{\mathsf{uce}}} \mathsf{uceF}\, F$.

Lemma 11.3 $F = \overline{f\, \overline{x} = \{b\}\ s} \to \mathit{Param}\, \overline{b}\, F\, (\mathsf{uceF}\, F)$.

Lemma 11.4 If $F' = \mathsf{uceF}\, F$ and $(\!|F|\!)_V; L, V, s) \sim_r (\!|F'|\!)_{V'}; L', V', \mathsf{uce}\, t)$ then

$$(L, V, \texttt{fun}\ F\ \texttt{in}\ s) \sim_r (L', V', \mathsf{if}\ |F'| = 0\ \mathsf{then}\ \mathsf{uce}\, t\ \mathsf{else}\ \texttt{fun}\ F'\ \texttt{in}\ \mathsf{uce}\, t).$$

Proof. If $|F'| = 0$, then $F' = \mathsf{nil}$ and after reducing only the left side one step by applying Sim-Expansion-Closed the assumption solves the goal. Otherwise we reduce both sides one step (Lemma 6.18), and the assumption solves the goal. ∎

Theorem 11.5 Let $R \vdash \mathbf{reach}_{\mathsf{s}}\, \{\mathbf{r}\}\, s$ such that $[\mathbf{r}] = \mathbf{true}$ and $L \sim_r L' :^{\mathcal{P}_{\mathsf{uce}}} R$. Then: $(L, V, s) \sim_r (L', V, \mathsf{uce}\, s)$.

Proof. Induction on s and in each case inversion of **reach**.

- The case for let follows from Sim-Let-Call and the inductive hypothesis.
- The case for the conditional follows by Lemma 8.2 and the inductive hypotheses.
- The application case follows from $L \sim_r L' :^{\mathcal{P}_{\mathsf{uce}}} R$ with Lemma 8.21, after discharging premises: Note that $R_f = \mathbf{true}$ by inversion on **reach**, so $\mathit{Idx}\, R_f\, f\, f$ holds by definition. Argument evaluation agrees, since parameters, and arguments and environments are identical.
- The case for return expressions is trivial, since the return expressions and environments are identical.
- In the function definition case, let $F' = \mathsf{uceF}\, F$. Lemma 11.4 lets us deal with both cases uniformly and requires

  $$(\!|F|\!)_V; L, V, t) \sim_r (\!|F'|\!)_E; L', E, \mathsf{uce}\, t).$$

  After applying the inductive hypothesis, we must show

  $$(\!|F|\!)_E; L \gtrsim_r (\!|F'|\!)_E; L' :^{\mathcal{P}} R'; R.$$

  We apply Lemma 8.18 and discharge its premises by using Lemma 11.2 and Lemma 11.3. The remaining premise requires us to show from

  $$(\!|F|\!)_E; L \sim_r (\!|F'|\!)_E; L' :^{\mathcal{P}} R'; R \qquad (*)$$

  that $\mathsf{Bdy}^{\mathcal{P}}_{L,L'}\, (\!|F|\!)_E\, (\!|F'|\!)_E\, (R'; R) \subseteq \sim_r$. Unfolding Bdy, we obtain $\mathit{Idx}\, (R'; R)\, f\, f'$ such that $F_f = (\overline{x}, s)$ and $F'_{f'} = (\overline{x}', s')$. Furthermore, we get $\mathit{Arg}\, E\, E\, R'_f\, \overline{v}\, \overline{v'}$. Unfolding those, we have to show

  $$(\!|F|\!)_E; L, E[\overline{x} \mapsto v], s) \sim_r (\!|F'|\!)_E; L', E[\overline{x} \mapsto v], \mathsf{uce}\, s)$$

  The inductive hypothesis solves the goal with $(*)$. ∎

$$\text{NUC-Op}\ \frac{\mathbf{noUC}_{\mathsf{cnd}}\ s}{\mathbf{noUC}_{\mathsf{cnd}}\ (\mathtt{let}\ x = \eta\ \mathtt{in}\ s)} \qquad \text{NUC-App}\ \frac{}{\mathbf{noUC}_{\mathsf{cnd}}\ \overline{e}} \qquad \text{NUC-Cond}\ \frac{\mathbf{noUC}_{\mathsf{cnd}}\ s_1 \qquad \mathbf{noUC}_{\mathsf{cnd}}\ s_2}{\mathbf{noUC}_{\mathsf{cnd}}\ (\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2)}$$

$$\text{NUC-Return}\ \frac{}{\mathbf{noUC}_{\mathsf{cnd}}\ e} \qquad \text{NUC-Fun}\ \frac{\forall i, \mathbf{noUC}_{\mathsf{cnd}}\ s_i \qquad \forall i, \mathsf{isCalledFrom}\ (\overline{f\ \overline{x} = s})\ t\ f_i}{\mathbf{noUC}_{\mathsf{cnd}}\ (\mathtt{fun}\ \overline{f\ \overline{x} = s}\ \mathtt{in}\ t)}$$

Figure 11.5: Definition of **noUC**.

11.5 Absence of Unreachable Code

We define a predicate **noUC** that specifies that all functions defined in a program are reachable in the sense of isCalledFrom in Figure 11.5.

Theorem 11.6 If $R \vdash \mathbf{reach}_{\mathrm{C,S}}\ s$ and s is marked reachable, then uce s contains no unreachable code: $\mathbf{noUC}_{\mathsf{cnd}}\ (\mathsf{uce}\ s)$.

11.5.1 Properties of Programs without Unreachable Code

If a program is coherent and does not contain unreachable code, then liveness information sound with respect to imperative liveness is also sound with respect to functional liveness.

Theorem 11.7 If $\mathbf{noUC}_{\mathsf{isCalled}}\ s$ and $\mathbb{A} \vdash \mathbf{coh}\ as$ and $\mathbb{A} \sqsubseteq \mathbb{A}'$ and $\mathbb{A}' \vdash \mathbf{live}_{\mathbf{I}}\ s : X$ then $\mathbb{A}' \vdash \mathbf{live}_{\mathbf{F}}\ s : X$.

Since every renamed apart program is coherent, we get:

Theorem 11.8 If $\mathbf{noUC}_{\mathsf{isCalled}}\ s$ and s is renamed apart and $\mathbb{A}$ and $\mathbb{A} \vdash \mathbf{live}_{\mathbf{I}}\ s : X$ then $\mathbb{A} \vdash \mathbf{live}_{\mathbf{F}}\ s : X$.

12 Establishing Coherence for IL/I: SSA Construction

In this section, we describe how to establish coherence while preserving IL/I semantics. This effectively yields an algorithm that translates an IL/I program to an equivalent IL/F program. Since IL/F programs can be easily renamed apart, and renamed apart IL/F programs are in SSA, the algorithm is a SSA construction algorithm.

The translation from IL/I to IL/F introduces parameters to the functions in an IL/I program in an equivalence-preserving manner to obtain a coherent program. For a given IL/I program, there are usually several possible translations depending on the number of introduced parameters. We motivate the idea behind the translation using coherence. As we discussed in §10.1, a coherent program may only apply functions with an available closure. Closures may become unavailable if registers are reassigned. The program in Listing 12.1, for example, is not coherent because x is reassigned in line 3, making the closure of f unavailable. Note that both programs below are to be interpreted imperatively.

Listing 12.1: An IL/I Program (not coherent)

```
1 let x = 7 in
2 fun f () = x in
3 if y then let x = 3 in f ()
4      else f ()
```

To make the program coherent, it must be ensured that the closure of f remains available after the assignment to x in line 3. This is accomplished by by making x a parameter of f:

Listing 12.2: An IL/I Program (coherent)

```
1 let x = 7 in
2 fun f x = x in
3 if y then let x = 3 in f x
4      else f x
```

At all applications, the parameter x itself is used as argument. In general, it is sufficient (but not necessary) to introduce every global of each function as additional identity argument to make a program coherent. In particular, we could have introduced y as an additional second parameter to f in Listing 12.2. For example, the

following program in which all free variables have been added as parameters to every function is also coherent:

Listing 12.3: An IL/I Program (coherent)

```
let x = 7 in
fun f (x,y) = x in
if y then let x = 3 in f (x,y)
     else f (x,y)
```

Adding parameters corresponds to placing ϕ-functions [Kel95; App98], and minimizing the number of ϕ-functions is desirable for practical purposes [Cyt+91]. Hence our translation makes an effort to require fewer parameters.

12.1 Adding Parameters

We begin by giving a translation that adds parameters according to annotations at function definitions. As usual, we maintain the annotations in a separate tree, but introduce the following notation to display inline that the additional parameter annotation is $\overline{z}$:

$$\texttt{fun}\ \overline{f\,\overline{x} = s \oplus \overline{z}}\ \texttt{in}\ t$$

Given such additional parameter annotations, the function addParams defined in Figure 12.1 adds the additional parameters and arguments to the program. For example, the mutually recursive function definition above would be translated to

$$\texttt{fun}\ \overline{f\,\overline{xz} = \ldots}\ \texttt{in}\ \ldots$$

Each application $f\,\overline{e}$ gets the corresponding additional parameters $\overline{z}$ as additional arguments:

$$f\,\overline{ez}$$

Example 12.1 The following two programs show that adding parameters does neither respect program equivalence nor the implementation relation.

```
fun f () =
 if true then 1
 else x
in
f ()
```

```
fun f (x) =
 if true then 1
 else x
in
f (x)
```

The program on the right has been obtained from the program on the left by adding parameter x to f. While the two programs are equivalent in variable environments where x is defined, the right-hand side program gets stuck in environments where x is undefined, while the left-hand side program terminates.

Example 12.1 shows that a correctness property of addParams involves assumptions about definedness of variables. We hence postpone the correctness statement to Theorem 12.5. The definition of addParams is given in Figure 12.1. The context Π : context (list $\mathcal{V}$) maps each function to the parameters that must be added at each of its call sites.

$$
\begin{aligned}
\mathsf{addParams} \;:\; \mathsf{context}(\mathsf{list}\,\mathcal{V}) \to \mathsf{Exp} \to \mathsf{Ann}\,(\mathsf{list}\,\mathcal{V}) \to \mathsf{Exp}& \\
\mathsf{addParams}\,\Pi\,(\mathtt{let}\;x = e\;\mathtt{in}\;s) &= \mathtt{let}\;x = e\;\mathtt{in}\;\mathsf{addParams}\,\Pi\,s \\
\mathsf{addParams}\,\Pi\,(\mathtt{if}\;x\;\mathtt{then}\;s\;\mathtt{else}\;t) &= \mathtt{if}\;x\;\mathtt{then}\;\mathsf{addParams}\,\Pi\,s \\
&\quad\;\mathtt{else}\;\mathsf{addParams}\,\Pi\,t \\
\mathsf{addParams}\,\Pi\,(f\,\overline{e}) &= f\,\overline{ez} \quad \text{where } \Pi_f = \overline{z} \\
\mathsf{addParams}\,\Pi\,x &= x \\
\mathsf{addParams}\,\Pi\,(\mathtt{fun}\;\overline{f\;\overline{x} \;=\; s \oplus \overline{z}}\;\mathtt{in}\;t) &= \mathtt{fun}\;\overline{f\;\overline{xz} \;=\; \mathsf{addParams}\,(\overline{f : \overline{z}}, \Pi)\,s} \\
&\quad\;\mathtt{in}\;\mathsf{addParams}\,(\overline{f : \overline{z}}, \Pi)\,t
\end{aligned}
$$

Figure 12.1: Definition of addParams, which adds parameters to a program. In the presentation, the annotation (4th parameter) is inlined into the third for readability as described in §4.8.2. For functions in the context, Π : context (list $\mathcal{V}$) contains the parameters to add.

12.2 Inductive Correctness Predicate

We now give an inductive judgment **ap** that will ultimately guarantee that the parameter annotations in s are sufficient for the program translated by addParams to be coherent. The judgment has the following form:

$$
\mathbb{A} \vdash \mathbf{ap}\; s \oplus \mathbf{a} : \mathbf{X}
$$

$\mathbb{A}$	:	$\mathcal{L} \rightharpoonup \mathsf{set}\,\mathcal{V}$	functions' globals
$\mathbf{a}$	:	$\mathsf{Ann}\,\mathsf{list}\,\mathcal{V}$	additional parameter annotation
$\mathbf{X}$	:	$\mathsf{Ann}\,\mathsf{set}\,\mathcal{V}$	liveness annotation
s	:	Exp	program

The predicate $\mathbb{A} \vdash \mathbf{ap}\; s \oplus \mathbf{a} : \mathbf{X}$ can be read as follows:

> Under the assumptions $\mathbb{A}$ about the globals of functions occurring free in s, and the additional parameter annotations $\mathbf{a}$ for functions defined in s are sufficient for addParams $\Pi\, s\, \mathbf{a}$ to be coherent.

$$\textsc{AP-Op}\ \frac{\lfloor \Lambda \rfloor_{X \setminus \{x\}} \vdash \mathbf{ap}\ s : X}{\Lambda \vdash \mathbf{ap}\ \mathtt{let}\ x = \eta\ \mathtt{in}\ s} \qquad \textsc{AP-Cond}\ \frac{\Lambda \vdash \mathbf{ap}\ s \qquad \Lambda \vdash \mathbf{ap}\ t}{\Lambda \vdash \mathbf{ap}\ \mathtt{if}\ x\ \mathtt{then}\ s\ \mathtt{else}\ t} \qquad \textsc{AP-Var}\ \frac{}{\Lambda \vdash \mathbf{ap}\ x}$$

$$\textsc{AP-App}\ \frac{\Lambda_f \neq \bot}{\Lambda \vdash \mathbf{ap}\ f\ \overline{y}}$$

$$\textsc{AP-Fun}\ \frac{\forall g, \lfloor \overline{f : X \setminus \overline{xz}}; \Lambda \rfloor_{X_g \setminus \overline{x}_g \overline{z}_g} \vdash \mathbf{ap}\ s_g \qquad \overline{f : X \setminus \overline{xz}}; \Lambda \vdash \mathbf{ap}\ t \qquad \forall g, \overline{z}_g \subseteq X_g \wedge \overline{x}_g \overline{z}_g \text{ duplicate-free}}{\Lambda \vdash \mathbf{ap}\ \mathtt{fun}\ \overline{f\ \overline{x} = s \oplus \overline{z} : X}\ \mathtt{in}\ t}$$

Figure 12.2: Correctness predicate for additional parameters. Both annotations are inlined into the program for readability as described in §4.8.2.

$\mathbf{a}$ annotates function definitions inside s with additional parameters. Λ is a globals environment used exactly as in the definition of coherence. Intuitively, **ap** is a variant of the coherence judgment that pretends that the additional parameters from the annotation have already been added to the program (according to addParams).

12.2.1 Description of the Rules

Each rule in Figure 12.2 corresponds to a rule of coherence (see Figure 10.1), with the difference that **ap** checks coherence for the program addParams $\Pi\ s\ \mathbf{a}$, i.e. the program in which the parameters have already been added. The rules AP-Op, AP-Cond, AP-Var and AP-App are analogous to Coh-Op, Coh-Cond, Coh-Var and Coh-App from the definition of coherence.

The rule AP-Fun takes care of handling the additional parameters. First note that for any function g, we have that $\overline{x}_g \overline{z}_g$ are the parameters after the translation. The globals of a function g after the translation are hence obtained from the live-ins X_g provided by the liveness annotation as $X_g \setminus \overline{x}_g \overline{z}_g$. For the judgment recursion, these globals are added to the globals context Λ, in the cases for the function bodies with appropriate restriction (cf. Coh-Fun).

The third premise of Coh-Fun handles liveness requirements for the additional parameters. First, the additional parameters must be a subset of the live-ins of the function, and second, the resulting parameter list $\overline{x}_g \overline{z}_g$ must be duplicate free. Note that duplicate parameters introduce shadowing, and hence cannot be accessed in the function. That uniqueness of the additional parameters is hence a minimal quality

requirement for the annotation.

12.3 Decidability and Translation Validation

It is decidable if a given set of parameters suffices to achieve coherence:

Theorem 12.2 ■ Correctness of Translation is Decidable
Given Λ, s, $\mathbf{a}$ and $\mathbf{X}$, it is decidable whether $\Lambda \vdash \mathbf{ap}\ s \oplus \mathbf{a} : \mathbf{X}$ holds.

The proof of Theorem 12.2 is constructive and yields a decision procedure. This decision procedure can be used to translation validate algorithms that establish coherence while preserving IL/I semantics, that is, to translation validate SSA construction algorithms.

12.4 Correctness

To show correctness of the predicate **ap**, we have to show that it guarantees that the additional parameter annotations are sufficient for addParams to yield a coherent program. Furthermore, we must establish a simulation result, which is necessarily weaker than program equivalence as Example 12.1 shows.

Theorem 12.3 ■ Translation Establishes Coherence
If $\Lambda \vdash \mathbf{ap}\ s \oplus \mathbf{a} : \mathbf{X}$ then $\Lambda \vdash \mathbf{coh}\ (\mathsf{addParams}\ \Pi\ s\ \mathbf{a}) : \mathbf{X}$.

Proof. By induction on the derivation of **ap**. ■

Note that the liveness annotation $\mathbf{X}$ in the coherence judgment in the conclusion of Theorem 12.3 is the same as in the premise, although we get coherence of the translated program. This result suffices, because AP-Fun ensures that all additional parameters are already live-ins of the function. The following lemma makes this connection precise:

Lemma 12.4 ■ Translation Preserves Liveness
If $\Lambda \vdash \mathbf{live}_p\ s : \mathbf{X}$ and $\Lambda \vdash \mathbf{ap}\ s \oplus \mathbf{a} : \mathbf{X}$ then $\Lambda \vdash \mathbf{live}_p\ (\mathsf{addParams}\ \Pi\ s\ \mathbf{a}) : \mathbf{X}$.

Proof. By induction on the derivation of **live** and inversion on **ap**. ■

What remains to be shown is that the translated program is observationally equivalent to the original program. The proof has to argue that introducing parameters does not change the semantics. We do the proof by coinduction, so we have to ensure that the invariant is maintained after function applications for the function bodies. For this purpose, we define the following predicate that relates two function contexts L, L' if L' is a translation of L according to globals Λ and additional parameters Π. The judgment has the following form, and the rules are given in Figure 12.3.

$$\mathbb{A} \mid \Pi \vdash\ L\ /\ L' \qquad \begin{array}{lll} \mathbb{A} & : & \mathcal{L} \rightharpoonup \mathsf{set}\,\mathcal{V} \quad \text{globals mapping} \\ \Pi & : & \mathcal{L} \rightharpoonup \mathsf{set}\,\mathcal{V} \quad \text{additional parameters} \\ L, L' & : & \mathsf{context} \quad \text{label contexts} \end{array}$$

AP-Ctx-Con ensures that the function bodies in L and L' are pairwise translations of each other.

AP-Ctx-Emp

$$\frac{}{\varnothing \mid \varnothing \vdash\ \varnothing\ /\ \varnothing}$$

AP-Ctx-Con

$$\frac{\mathbb{A} \mid \Pi \vdash\ L\ /\ L' \qquad \forall g, X_g \neq \bot \to \lfloor \overline{f : X}; \mathbb{A}, \rfloor_{X_g} \vdash \mathbf{ap}\ \overline{f : \overline{z}}; \Pi \oplus s_g : \overline{z}_g X_g}{\overline{f : X}; \mathbb{A} \mid \overline{f : \overline{z}}; \Pi \vdash\ \overline{f : (\overline{x}, s)}; L\ /\ \overline{f : (\overline{x}\overline{z}, \mathsf{addParams}\,(\overline{f : \overline{z}}; \Pi)\ s)}; L'}$$

Figure 12.3: Context translation relation.

Theorem 12.5 ▪ Translation is Correct
If $\mathbb{A} \vdash \mathbf{ap}\ s \oplus \mathbf{a} : \mathbf{X}$ and $\mathbb{A} \mid \Pi \vdash\ L\ /\ L'$ and $\mathbb{A} \sqsubseteq \mathbb{A}'$ such that $\mathbb{A}' \vdash \mathbf{live_I}\ s : \mathbf{X}$ and V is defined on $[\mathbf{X}]$ then

$$(L, V, s)_{\mathrm{I}} \approx_r^p (L', V, \mathsf{addParams}\,\Pi\ s)_{\mathrm{I}}.$$

Theorem 12.5 and Theorem 12.3 together with the main result about coherence (Theorem 10.8) ensure that we get the following corollary, which shows that we can change semantic interpretations.

Corollary 12.6 ▪ Translation to IL/F
If $\mathbb{A} \vdash \mathbf{ap}\ s \oplus \mathbf{a} : \mathbf{X}$ and $\mathbb{A} \mid \Pi \vdash\ L\ /\ \mathsf{strip}\ L'$ and $\mathbb{A} \sqsubseteq \mathbb{A}'$ such that $\mathbb{A}' \vdash \mathbf{live_I}\ s : \mathbf{X}$ and V is defined on $[\mathbf{X}]$ then

$$(L, V, s)_{\mathrm{I}} \approx_r^p (L', V, \mathsf{addParams}\,\Pi\ s)_{\mathrm{F}}.$$

12.4.1 Discussion of the Correctness Predicate

The correctness predicate **ap** requires the additional parameters to be live, which imposes requirements on the construction algorithms the predicate admits. The requirement that additional parameters must be live actually forces algorithms to translate to what corresponds to pruned SSA-form, which we discuss in §18.12.1. This means, for example, that the predicate cannot be used to justify correctness

of an algorithm that adds variables occurring in the program to every function as additional parameters.

In the Coq development, Theorem 12.5 is formulated separately of the assumption that liveness does not change before and after the translation. We think that, if desired, construction algorithms that introduce dead parameters could be integrated with minor effort. Since we prove correct a construction algorithm that doesn't introduce dead parameters in §12.5, however, the advantages of such an approach are unclear.

Another remarkable property is the definedness requirement in Theorem 12.5, which arises from the fact that the translation may introduce reads into the program that were not present in the original, as shown in Example 12.1. It is not possible to decide in general whether variable that occurs in a program is read (Rice), hence every SSA translation has this problem. The definedness requirement in Theorem 12.5 ultimately arises from a design choice in the semantics: Reading an undefined variable gets stuck. This means that because our semantics differentiates between defined and undefined variables, Theorem 12.5 must account for the fact that the translation may in general introduce reads that were not present in the original program.

12.5 SSA Construction via Adding Parameters

In this section, we given an algorithm deloc that computes additional parameter such that the predicate **ap** holds of the translation, i.e. that after adding these parameters the resulting program is coherent.

There are two reasons why an additional parameter might be necessary and which the algorithm accounts for:

1. A parameter x must be added to f, if x is a global of f and x may be redefined between the definition of f and the application of f
2. A parameter x must be added to g, if x is a global of g and there is a function f that may be called from g and x must be added as a parameter to f

Example 12.7 provides an example for each of these two cases.

Example 12.7 Consider Figure 12.4. The parameter x must be added to f, because x is redefined in line 4. This is an instance of case 1 from above. The parameter x must also be added to g, since g is called from f (which might have altered x), and x is a global of g. This is an instance of case 2 from above.

These two reasons provide the key to understanding the definition of the algorithm deloc in Figure 12.5, which we now describe in detail.

```
fun g () =
  fun f () =
    if x then
      let x = x - 1
      in f ()
    else g ()
  in if x then f ()
           else 5
in f ()
```

```
fun g x =
  fun f x =
    if x then
      let x = x - 1
      in f ()
    else g x
  in if x then f x
           else 5
in g x
```

Figure 12.4: Example programs for Example 12.7.

12.5.1 The Intuition behind deloc

We describe arguments and return values of deloc. Consider the following invocation:

$$\mathsf{deloc}\ \Lambda\ \Pi\ s\ \mathbf{X} = (\mathbf{a}, P)$$

The arguments of deloc are the program s and the corresponding liveness annotation $\mathbf{X}$, and two contexts. Λ is a globals context corresponding to the one maintained in the **live** judgment. The context Π maintains for each function the subset of its globals that may have been redefined; that means if s were to apply f immediately then Π_f would be the additional parameters that need to be added to f.

The result of the algorithm is a tuple. The first component $\mathbf{a}$ of the tuple is an annotation tree that contains the additional parameter annotations for s. The second component P an additional parameter context that contains for each function the parameters that must be added because of s and Π. The difference between Π and P is that first, P may contain more parameters per function because it also accounts for redefinitions that occur in s, and second, P may map the entry for a function f to $\bot$, even if $\Pi_f \neq \bot$, provided that the function f is not called in s. The algorithm is defined in Figure 12.5.

12.5.2 Description of deloc

In this section we describe the defining equations of deloc given in Figure 12.5 in detail.

The let-binding case modifies the context Π by adding x to the list of additional parameters for every function that has x as a global.

The conditional case computes the results for the two branches, and uses the corresponding first components to construct the program annotation for the conditional.

$$
\begin{aligned}
&\mathsf{deloc} : \mathsf{context}(\mathsf{set}\,\mathcal{V}) \to \mathsf{context}(\mathsf{list}\,\mathcal{V}) \to \mathsf{Exp} \to \mathsf{Ann}\,(\mathsf{set}\,\mathcal{V}) \\
&\qquad \to \mathsf{Ann}\,(\mathsf{list}\,\mathcal{V}) \times \mathsf{context}(\mathsf{set}\,\mathcal{V}) \\
&\mathsf{deloc}\,\mathbb{A}\,\Pi\,(\texttt{let}\,x = \eta\,\texttt{in}\,s) \quad = \\
&\quad \mathsf{let}\,(\mathbf{a}, P) = \mathsf{deloc}\,\mathbb{A}\,(\mathsf{add}\,x\,\mathbb{A}\,\Pi)\,s\,\mathsf{in} \\
&\quad (\emptyset \cdot \mathbf{a}, P) \\
&\mathsf{deloc}\,\mathbb{A}\,\Pi\,(\texttt{if}\,e\,\texttt{then}\,s\,\texttt{else}\,t) \quad = \\
&\quad \mathsf{let}\,(\mathbf{a}, P) = \mathsf{deloc}\,\mathbb{A}\,\Pi\,s\,\mathsf{in} \\
&\quad \mathsf{let}\,(\mathbf{a}', Q) = \mathsf{deloc}\,\mathbb{A}\,\Pi\,t\,\mathsf{in} \\
&\quad (\emptyset \cdot \mathbf{a}, \mathbf{a}', P \cup Q) \\
&\mathsf{deloc}\,\mathbb{A}\,\Pi\,e \quad = \quad (\emptyset, \emptyset) \\
&\mathsf{deloc}\,\mathbb{A}\,\Pi\,(f\,\overline{e}) \quad = \quad (\emptyset, f : \Pi_f) \\
&\mathsf{deloc}\,\mathbb{A}\,\Pi\,(\texttt{fun}\,\overline{f\,\overline{x} = s : X}\,\texttt{in}\,t) \quad = \\
&\quad \mathsf{let}\,\mathbb{A}' = \overline{f : X \setminus \overline{x}}; \mathbb{A}\,\mathsf{in} \\
&\quad \mathsf{let}\,\Pi' = \overline{f : \emptyset}; \Pi\,\mathsf{in} \\
&\quad \mathsf{let}\,(\mathbf{a}', Q) = \mathsf{deloc}\,\mathbb{A}'\,\Pi'\,t\,\mathsf{in} \\
&\quad \mathsf{let}\,\overline{(\mathbf{a}, P)} = \overline{\mathsf{deloc}\,\mathbb{A}'\,\Pi'\,s}\,\mathsf{in} \\
&\quad \mathsf{let}\,\overline{f : \overline{y}}; \Pi'' = Q \cup \bigcup P\,\mathsf{in} \\
&\quad \mathsf{let}\,Z = \bigcup^2 \overline{yx}\,\mathsf{in} \\
&\quad \mathsf{let}\,\overline{f : \overline{z}}; \Pi''' = \mathsf{add}\,Z\,\mathbb{A}'\,(\overline{f : y}; \Pi'')\,\mathsf{in} \\
&\quad (\overline{\overline{z}} \cdot \overline{\mathbf{a}}, \mathbf{a}', \Pi''') \\
&\mathsf{add}\,\overline{y}\,\mathbb{A}\,\Pi = \mathsf{zip}\,(\lambda X\,\overline{z}.\,(\overline{y} \cap X)) \cup \overline{z})\,\mathbb{A}\,\Pi
\end{aligned}
$$

Figure 12.5: The Delocation algorithm deloc.

The result context is constructed by point-wise union of the contexts P and Q. This collects the additional parameters required in both branches into one result context.

The expression case never requires additional variables.

The application case requires that the additional parameters Π_f collected so far for f are recorded in the result context, an ensures that the additional parameters for all functions different from f are $\bot$.

The function definition case first constructs extended contexts $\mathbb{A}'$ and Π', and computes the results for all subterms recursively. Note that $\overline{P}$ is a list of result contexts such that P_i corresponds to s_i. Then we compute the point-wise union of all result contexts $Q \cup \bigcup P$, and pattern match on the resulting additional parameter context to single out the part $\overline{f : \overline{y}}$ that holds the additional parameters for functions defined in this function definition, and the part Π'' that holds the additional parameters for

functions previously defined. Note that any two mutually recursively defined functions $\overline{f}$ may call each other; figuring out the call structure would be a dominance analysis. However, this information is encoded in part in the liveness information: the globals upper-bound the additional parameters. We now collect all parameters and additional parameters of any function in $\overline{f}$ together in Z, and then add the variable x in Z as additional parameter to each function if x is a global of that function. We pattern match on the resulting additional parameter context and obtain $\overline{f : \overline{z}}$ as additional parameters for newly defined functions, and Π''' as parameter context for the previously defined functions. From this information it is easy to construct the result value for the function definition case.

The helper function add takes additional parameter candidates $\overline{y}$, a globals context Λ and an additional parameter context Π, and yields a context where each entry is obtained by adding $\overline{y} \cap \Lambda_f$ to Π_f.

12.5.3 Correctness of deloc

For the correctness proof of deloc it suffices to derive an appropriate instance of the **ap** judgment. The properties we are after then follow from Theorem 12.3, which provides that adding the parameters yields a coherent program, and Theorem 12.5, which provides semantic equivalence.

Theorem 12.8 ▪ Correctness of deloc
Let $\Lambda \vdash \mathbf{live_I}\ \mathbf{X} : s$ and $\mathsf{deloc}\ \Lambda\ \Pi\ s\ \mathbf{X} = (\mathbf{a}, P)$ and $\Pi \subseteq \Lambda$ and $\mathbf{noUC}_{\mathsf{isCalled}}\ s$. Then: $\lfloor \Lambda \setminus P \rfloor_{[\mathbf{X}]} \vdash \mathbf{ap}\ s \oplus \mathbf{X} : \mathbf{a}$.

The Unreachable Code Requirement

The requirement to have no unreachable code present in Theorem 12.8 deserves comment.

First note that in the function definition case, both the coherence judgment **coh**and the derived judgment **ap** require that the judgment holds recursively for each function body that is syntactically present. This recursive requirement is, in particular, independent of reachability considerations. That means that even if a function f is never applied, its body must be coherent, or satisfy the **ap** requirement, respectively.

Example 12.9 The algorithm deloc avoids introducing a parameter x to a function f if f is never called after the redefinition x. Even though x is redefined on the following program, deloc does not introduce x as parameter for f, because f is never called after the redefinition of x.

```
1 fun f () =
2   if x then f ()
3   else let x = 5 in x
4 in f()
```

Example 12.9 implies that if a function is never called (not even from an unreachable function), it gets no additional parameters.

Example 12.10 In the following program, deloc adds no parameters to g:

```
1 fun f () =
2   if x then f ()
3   else let x = 5 in x
4 and g () =
5   if x then
6     let x = x+1 in f ()
7   else x
8 in f()
```

The consequence is that deloc introduces no parameters at all in the program from Example 12.10, and hence the translated program is not coherent (although invariant), because f is unavailable after its global x was redefined in line 6. The redefinition of x in line 6, however, does not have any impact, because g is never called.

For this reason, we have to forbid unreachable code in the correctness statement Theorem 12.8: Otherwise, the resulting programs would not be coherent (although invariant). Another approach would be weaken the requirement in the definition of coherence, and require that a function body only needs to be coherent, if it is called in the sense of isCalled as defined in §11. We chose not to incorporate the reachability requirement in the definition of coherence, for two reasons:

1. LVC uses the SSA construction algorithm twice: Once in the front-end to translate into IL/F, and once in the back-end during register allocation. On both occasions DCE recently ran, and by Theorem 11.6 DCE eliminates all unreachable code. We hence have no problem to satisfy the premises of Theorem 12.8 in the correctness proofs.

2. It is not clear where to draw the line on what exceptions to incorporate in the definition of coherence. Each of these exceptions, starting with the possible incorporation of reachability, ultimately has the goal of reducing the parameter requirements further. To this end, we could even start evaluating constant conditions, then we could incorporate a constant propagation, and so on. The ultimate definition of coherence would be bloated and it would be hard to expose

which premise serves what purpose. We hence opted for the simplest possible definition of coherence.

12.6 Minimality

The standard SSA-construction algorithm [Cyt+91] minimizes the number of ϕ-functions. Minimality is defined with respect to the dominance order of the blocks in the CFG of the program. Our translation predicate supports a minimal construction with respect to the definition of coherence, but we do not formally prove this. Relating a definition of minimality based on coherence to a definition of minimality based on dominance would require a formalization of dominance within the LVC framework, which we currently do not have.

We can, however, show that the number of additional parameters depends on the block-nesting structure.

Example 12.11 The four programs below show that the number of additional parameters required for coherence depends on block-nesting structure of the program. The bottom row contains coherent versions of the program in the top row which have been obtained by adding the minimal number of additional parameters required. The two programs in the top row differ only in their block nesting structure. The block nesting in the top right program allows g to take no argument, while x is required as additional parameter for both f and g in left program.

IL/I programs differing only in their block-nesting structure

```
fun g () = x in
fun f () = g () in
let x = 3 in
f()
```

```
fun f () =
  fun g () = x in
  g () in
let x = 3 in
f ()
```

TIF translations with minimal number of parameters

```
fun g x = x in
fun f x = g x in
let x = 3 in
f x
```

```
fun f x =
  fun g () = x in
  g () in
let x = 3 in
f x
```

13 Establishing Coherence for IL/F: Register Allocation

Register allocation transforms an IL/F program that uses an unbounded number of pseudo registers into an IL/I program that uses a fixed number of registers k and an unbounded number of memory locations which we call *slots*. The number of available registers k is called *register bound*.

To distinguish registers from slots, we partition the set of variables into two infinite subsets. It is important that both subsets are infinite, as this guarantees that we can construct a total function that, given a finite set of variables X finds a variable that is not in X. We arrange things such that the actual machine registers are the smallest k variables in the register partition.

We restrict the usage of slots such that they can only be used in variable to variable assignments, as parameters of functions, and as arguments in function applications. We call this restriction the *slot restriction*, which we formally defined in §13.1. Lowering operations on slots to the corresponding operations on the stack is straight-forward as described in §16.

More formally, the register allocation requires the input program to satisfy that

- function arguments are variable only
- for every let-binding $\texttt{let}\ x = e\ \texttt{in}\ \ldots$ appearing in the program the number of free variables in e is less than or equal to the register bound

The register allocation phase maintains the above invariants, and additionally establishes the following properties:

1. slots are only used in variable to variable assignments and as function parameters and arguments
2. at most k different registers are used
3. the semantics of the program is preserved

Following the work by Hack [HGG06], register allocation takes advantage of a key property only available to SSA-based register allocation schemes.

Definition 13.1 ▪ Register Demand/Pressure
The number of distinct registers required to implement an IL/I program is called *register demand*. The size of the largest live-set occurring in a program is called

register pressure.

The important properties is:

> In SSA register demand is less or equal to register pressure [HGG06].

This property allows to separate the register allocation phase into two separate sub-phases:

Spilling If register pressure exceeds the number of available registers, values must be *spilled* to memory so that they can later be *reloaded* when they are needed. The register allocation phase must hence insert code that copies values between registers and memory if more variables are live than registers available, and this task is called *spilling*. Note that the key advantage of SSA-based spilling is that register pressure can be used as a proxy for register demand. This allows the spilling algorithm to determine whether spilling is necessary at a program point without knowing the register assignment. Spilling then inserts spills and reloads to lower register pressure sufficiently, i.e. the result of spilling is that register pressure is less or equal to the register bound.

Register Assignment decides which variable resides in which register at which program point, and this phase is called *register assignment*. Register assignment starts from a program where the register pressure is less than or equal to the register bound. We then use Hack's approach to find an α-renaming of the program in which the number of distinct register variables is less than or equal to the register bound.

The spilling and register allocation phase work together to establish the three invariants above. The spilling phase already establishes invariant 1 and of course ensures that invariant 3 holds. Spilling does not establish invariant 2 directly, but establishes that the largest live set in the program is less or equal to the register bound. The register assignment phase then establishes invariant 2.

The implementation of register allocation involves the following phases, in order:

1. Spilling on IL/I (§13.2)
2. SSA construction transforming to coherent IL/F (§12)
3. Renaming apart the register partition on coherent IL/F
4. Register assignment on coherent IL/F (§13.4)
5. Argument elimination on IL/I by lowering to parallel moves, and consequent implementation of parallel moves with a sequence of assignments, thereby avoiding slot-to-slot assignments (§13.5)

$$\textsc{SlotLoad}\ \frac{x \in \mathcal{V}_R \qquad y \in \mathcal{V}_M \qquad \mathbf{slotRes}\, s}{\mathbf{slotRes}\ \texttt{let}\ x = y\ \texttt{in}\ s} \qquad \textsc{SlotLet}\ \frac{\mathsf{fv}\, e \subseteq \mathcal{V}_R \qquad \mathbf{slotRes}\, s}{\mathbf{slotRes}\ \texttt{let}\ x = e\ \texttt{in}\ s}$$

$$\textsc{SlotReturn}\ \frac{\mathsf{fv}\, e \subseteq \mathcal{V}_R}{\mathbf{slotRes}\, e} \qquad \textsc{SlotIf}\ \frac{\mathsf{fv}\, e \subseteq \mathcal{V}_R \qquad \mathbf{slotRes}\, s \qquad \mathbf{slotRes}\, t}{\mathbf{slotRes}\ \texttt{if}\, e\, \texttt{then}\, s\, \texttt{else}\, t} \qquad \textsc{SlotApp}\ \frac{}{\mathbf{slotRes}\, f\, y}$$

$$\textsc{SlotFun}\ \frac{\forall i, \mathbf{slotRes}\, s_i \qquad \mathbf{slotRes}\, t}{\mathbf{slotRes}\ \texttt{fun}\ \overline{f\, x\, =\, s}\ \texttt{in}\ t}$$

Figure 13.1: Definition of the predicate **slotRes** enforcing the slot restriction.

13.1 Variable Partitions and Slot Restriction

We partition the variables into two countably-infinite sets $\mathcal{V} = \mathcal{V}_R \uplus \mathcal{V}_M$, and require that the input program only contains variables from $\mathcal{V}_R$. We further parametrize the register allocation phase by an injection slot : $\mathcal{V}_R \to \mathcal{V}_M$ that generates the name of the spill slots for a given register variable (cf. CompCert [Ler09a]). We adopt the convention that in the source code, we use lower-case variables $x, y, \ldots$ for register variables, and upper-case variables $X, Y, \ldots$ for the (corresponding) slots. For example, we implement spilling of y by a let-binding of the corresponding spill slot $\texttt{let}\ Y = y\ \texttt{in}\ \ldots$ and use $\texttt{let}\ y = Y\ \texttt{in}\ \ldots$ to reload y later.

It is important that slots are not used by expressions in the program, as machine architectures require the arguments to most instructions to reside in registers. Some architectures allows memory operands for certain instructions, but we currently do not take advantage of this. We call this restriction the *slot restriction*, and define a predicate **slotRes** that guarantees that every variable is in a register whenever it is used. The predicate is inductively defined and the rules are given in Figure 13.1. The predicate also ensures that let-bindings with a memory slot on the left-hand side have a register on the right side, i.e. that slot to slot assignments are forbidden.

13.2 Spilling

Spilling transforms a program into an equivalent program by inserting spills and loads such that the number of registers in the maximal live set is afterwards bounded by a given integer k. The spilling phase relies on liveness information and involves three steps:

1. Spilling information is computed using a verified spilling algorithm, and associated with the orignal program in a way analogous to liveness annotations.
2. The spilled program is generated, i.e. the spilling information is materialized into let-bindings that implement the spills and loads
3. The new liveness annotation is computed. Note that it is enough to compute the liveness information at function definitions, as full liveness information can be reconstructed from it (see §9.1.6).

13.2.1 Spilling Information

Spilling annotations are annotation trees as described in §4.8, and the annotation at each program point is a tree-tuple. By convention, $\mathbf{c}$ ranges over spilling annotations, i.e. over the type of spilling annotation trees. Following the notational convention discussed in §4.8.2, we use the separator $:$ to indicate the spilling annotation at a certain program point. Hence, a statement with spilling annotation is written $s : (S, L, \_)$, where S is the set of variables to be spilled (**spill set**) and L is the set of variables to be loaded (**load set**). The third component is only required if s is a function applications or a function definition, and we discuss its purpose below.

13.2.2 Materializing Spills and Loads

A program s with spilling annotation $\mathbf{c}$ can be turned into a spilled program via the recursive function doSpill $Z\ \Lambda\ s\ \mathbf{c}$, which we now informally describe.

To generate the spilled program for $s : (S, L, \_)$, doSpill first prepends the statement s with spills for each variable in S, followed by the loads for each variable in L as depicted in Listing 1. For let statements, conditionals, and return statements this is all that needs to be done. Function definitions and applications require some additional work, which we describe next.

Function definitions take a pair of sets (R_f, M_f) as third component of the spilling annotation: $\texttt{fun}\ \overline{f\ \overline{x}\ =\ s_1}\ \texttt{in}\ s_2 : (S, L, (R_f, M_f))$. We call the pair (R_f, M_f) the **live-in cover** and require it to cover the live-ins X_f of f, i.e. $R_f \cup M_f = X_f$. The set R_f specifies the variables the function expects to reside in registers, and the set M_f specifies the variables the function expects to reside in memory. The sets R_f and M_f are not necessarily disjoint, as a function may want a variable to reside both

$$\mathsf{doSpillLocal}(s : (\underbrace{\{x_1, \ldots, x_n\}}_{\text{spills}}, \underbrace{\{y_1, \ldots, y_m\}}_{\text{loads}}, \_)) = \begin{array}{l} \texttt{let}\ \mathsf{slot}\ \ \mathtt{x_1} = \mathtt{x_1}\ \ \texttt{in}\ \ \ldots \\ \texttt{let}\ \mathsf{slot}\ \ \mathtt{x_n} = \mathtt{x_n}\ \ \texttt{in} \\ \texttt{let}\ \mathtt{y_1} = \mathsf{slot}\ \ \mathtt{y_1}\ \ \texttt{in}\ \ \ldots \\ \texttt{let}\ \mathtt{y_n} = \mathsf{slot}\ \ \mathtt{y_n}\ \ \texttt{in}\ s \end{array}$$

Listing 1: Definition of doSpillLocal

```
1  fun f x y z =                    R_f = {y, z}, M_f = {c, x, z}
2   if y > 0 then
3    let a = y+z in
4    f x a z                        R_app = {a, z}, M_app = {x, z}
5   else
6    if y = 0 then
7
8
9     x + c                         L = {c, x}
10   else
11    let w = y*y in
12    let a = y+w in
13    f x a z                       R_app = {a}, M_app = {x, z}
```

```
fun f X y z Z =
 if y > 0 then
  let a = y+z in
  f X a z Z
 else
  if y = 0 then
   let x = X in
   let c = C in
   x + c
  else
   let w = y*y in
   let a = y+w in
   f X a Z Z
```

Listing 2: A program with spilling annotations on the left (non-empty sets in spilling annotations are indicated by equations) and the resulting spilled program on the right. The live-ins of f are $\{x, y, z, c\}$. The variable c is free in f. Lowercase variables denote registers, uppercase variables denote spill slots. In line 4, z is passed in register and memory to avoid loading z in line 3. The application in line 12 implicitly loads z (3rd parameter).

in register and in memory when it is applied (see Listing 2). Besides inserting spills and loads according to S and L as already described, the function parameters must be modified to account for parameters that are passed in spill slots. For this purpose, every parameter $x_i \in M_f \setminus R_f$ is replaced by the name slot x_i in $\overline{x}$. Furthermore, for any parameter $x_i \in M_f \cap R_f$ an additional parameter with name slot x_i is inserted directly after x_i. The following function performs this task:

$$\mathsf{exparm}\,(R, M)\,\mathsf{nil} = \mathsf{nil}$$

$$\mathsf{exparm}\,(R, M)\,(x_1, \overline{x}) = \begin{cases} x_1, \mathit{slot}\,x_1, \mathsf{exparm}\,(R, M)\,\overline{x} & \text{if } z \in R \cap M \\ x_1, \mathsf{exparm}\,(R, M)\,\overline{x} & \text{if } z \in R \\ \mathit{slot}\,x_1, \mathsf{exparm}\,(R, M)\,\overline{x} & \text{otherwise} \end{cases}$$

Function applications have a pair of sets (R_{app}, M_{app}) as third component of spilling information and take the form $f\,y_1, \ldots, y_n : (S, L, (R_{app}, M_{app}))$. We require all function arguments y_i to be variables, and that $R_{app} \cup M_{app} = \{y_1, \ldots, y_n\}$. The sets R_{app} and M_{app} indicate the availability of argument variables at the function application. If an argument variable y_i is available in a register, then the spilling algorithm sets $y_i \in R_{app}$, if it is in memory, then $y_i \in M_{app}$. Besides inserting spills and loads according to S and L as already described, doSpill modifies the argument vector $y_1, \ldots, y_n$. For every parameter $x_i \in R_f$ such that the corresponding argument

variable y_i is not in R_{app} (i.e. not available in a register), the variable y_i is replaced by the name slot y_i in the argument vector. For every parameter $x_i \in M_f \setminus R_f$ such that the corresponding argument variable y_i is in M_{app} (i.e. available in memory), the variable y_i is replaced by the name slot y_i in the argument vector. Furthermore, for every parameter $x_i \in M_f \cap R_f$ an additional argument is inserted directly after the corresponding argument variable (y_i or slot y_i) in $y_1, \ldots, y_n$, and the name of the additional argument is slot y_i if $y_i \in M_{app}$ and y_i otherwise. In this way, R_f and M_f are used to avoid implicit loads and stores at function application if availability, as indicated in R_{app} and M_{app}, permits. Since spill slots are just a partition of the variables, parameter passing can copy between spill slots and registers if the argument variable y_i for a register parameter x_i is only available in memory, or vice versa. This fits nicely in our setting, as we handle the generation of these implicit spills and loads later on, when parameter passing is lowered to parallel moves. In line 12 of Listing 2, for example, the application implicitly loads z. In contrast, availability of z in both register and memory at the application in line 4 allows avoiding any implicit loads and stores. Assuming $y > 0$ holds for most executions, this is beneficial for performance.

13.3 A Correctness Criterion for Spilling

We define a correctness predicate for spilling on programs with spilling annotation of the form

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s : \mathbf{c} : \mathbf{X}$$

The correctness predicate is defined relative to sets R and M, which contain the variables currently in registers, and in memory, respectively. Additionally, the **parameter context** Z maps function names to their parameter list, and the **live-in cover context** Σ maps functions to their live-in cover. The parameter k is the register bound. The spilling predicate uses the spilling annotation $\mathbf{c}$ as well as the liveness annotation $\mathbf{X}$. The rules defining the predicate are given in Figure 13.2 and require the presence of liveness information for the formulation of the rule SpillFun.

13.3.1 Description of the Rules of the Inductive Predicate

The predicate consists of two generic rules that handle spilling and loading, and one rule for each statement. Rules for statements only apply once spills and loads have been handled. This is achieved by requiring empty spill and load sets in statement rules, and requiring an empty spill set in the load rule. SpillSpill requires $S \subseteq R$ to ensure only variables currently in registers are spilled. The new memory state is $M \cup S$. SpillLoad requires the spill set to be empty. Its second premise ensures there are enough free registers to load all values. The *kill set* K represents the variables that may be overwritten because they are not used anymore or are already spilled.

SpillSpill
$$\frac{S \subseteq R \qquad Z \mid \Sigma \mid R \mid M \cup S \vdash \mathbf{spill}_k\ s : (\emptyset, L, \_)}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s : (S, L, \_)}$$

SpillLoad
$$\frac{L \subseteq M \qquad |R \setminus K \cup L| \leq k \qquad Z \mid \Sigma \mid R \setminus K \cup L \mid M \vdash \mathbf{spill}_k\ s : (\emptyset, \emptyset, \_)}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s : (\emptyset, L, \_)}$$

SpillReturn
$$\frac{\mathsf{fv}\, e \subseteq R}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ e : (\emptyset, \emptyset)}$$

SpillApp
$$\frac{\Sigma_f = (R_f, M_f) \qquad R_f \setminus Z_f \subseteq R \qquad M_f \setminus Z_f \subseteq M \qquad \overline{y} = R_{app} \cup M_{app} \qquad M_{app} \subseteq M \qquad R_{app} \subseteq R}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ (f\ y) : (\emptyset, \emptyset, (R_{app}, M_{app}))}$$

SpillIf
$$\frac{\mathsf{fv}\, e \subseteq R \qquad Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s_1 \qquad Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s_2}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ (\texttt{if}\, e\, \texttt{then}\, s_1\, \texttt{else}\, s_2) : (\emptyset, \emptyset)}$$

SpillLet
$$\frac{\mathsf{fv}\, e \subseteq R \qquad |R \setminus K \cup \{x\}| \leq k \qquad Z \mid \Sigma \mid R \setminus K \cup \{x\} \mid M \vdash \mathbf{spill}_k\ s}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ (\texttt{let}\, x = e\, \texttt{in}\, s) : (\emptyset, \emptyset)}$$

SpillFun
$$\frac{|R_f| \leq k \qquad \forall i, f : \overline{x}; Z \mid f : (R_f, M_f); \Sigma \mid R_f \mid M_f \vdash \mathbf{spill}_k\ s_i \qquad R_f \cup M_f = X_f \qquad f : \overline{x}; Z \mid f : (R_f, M_f); \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ t}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ (\texttt{fun}\, \overline{f\ x\ =\ s\ \{X\}}\, \texttt{in}\, t) : (\emptyset, \emptyset, \overline{(R_f, M_f)})}$$

Figure 13.2: Inductive correctness predicate $\mathbf{spill}_k$. We follow the convention discussed in §4.8.2

$R \setminus K \cup L$ is the new register state after loading. Clearly, K is most useful if $K \subseteq R$ because only then variables are removed from the register set R, but our proofs do not require this restriction. We also do not include K in the spilling annotation, as the spilling algorithm would have to compute liveness information to provide it. Simple spilling algorithms, such as the one we verify in §13.3.4, never need to compute liveness information.

SpillReturn requires that the free variables are in the registers. SpillIf requires the consequence and the alternative to fulfill the predicate on the same configuration,

and that the variables used in the condition are in registers. SpillLet deals with the new variable x, which needs a register. The resulting register state is $R \setminus K \cup \{x\}$, the size of which must be bounded by the register bound k. This imposes a lower bound on k. The kill set K reflects that there might be a variable y holding the value of a variable required to evaluate the expression e, that is then overwritten to store the value of x. In this case $K = \{y\}$.

SpillApp uses the sets R_f and M_f from the corresponding function definition. The premises $R_f \setminus Z_f \subseteq R$ and $M_f \setminus Z_f \subseteq M$ require that all live-ins of the function except parameters are available in registers and memory at the application. The remaining premises require that all argument variables are available either in the registers (R_{app}) or in the memory (M_{app}), as discussed in §13.2. Note that the argument vector $\overline{y}$ is variables only, i.e. applications can only have variables as arguments. SpillFun refers to the live-ins X_f to require that the live-in cover (R_f, M_f) covers the live-ins X_f of the program: $R_f \cup M_f = X_f$. The rule also requires the function to expect at most k variables in registers: $|R_f| \leq k$. The parameters and the live-in cover are recorded in the context. The condition for the function body s_1 uses R_f and M_f as register and memory sets, respectively.

13.3.2 Formalization of the Spill Predicate in Coq

The predicate **spill**$_k$ is realized with five rules in the Coq development instead of the seven rules presented here. Each of the five rules corresponds to a consecutive application of SpillSpill, SpillLoad and one of the statement-specific rules. The five-rule system behaves better under inversion and induction in Coq, but we think the formulation with seven rules provides more insight. The Coq development contains a formal proof of the equivalence of the two systems.

13.3.3 Soundness of the Correctness Predicate

In this subsection we show that our spilling predicate is sound. We show that if s is renamed apart and all variables in s are in $\mathcal{V}_R$, and the spilling liveness annotations in s are sound, the following holds for the spilled program s':

(§13.3.3) every variable in s' is in a register when used in a computation

(§13.3.3) the maximal live set in s' is bounded by k

(§13.3.3) s and s' have the same behavior

Variables in Registers

We define $\mathbb{M}\,(R, M) = R \cup M$ and analogously its pointwise lifting.

Lemma 13.2 Let $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s \vdots \mathbf{c} : \mathbf{X}$ and $\mathbb{M}\Sigma \setminus Z \vdash \mathbf{live_I}\ s : \mathbf{X}$ and let the program s be renamed apart and let all variables occurring in s be in $\mathcal{V}_R$. If $R \cup M \cup \bigcup Z \subseteq \mathcal{V}_R$ then **slotRes** (doSpill $Z\ \Sigma\ s\ \mathbf{c}$).

Proof. The conditions follow directly by induction on $\mathbf{spill}_k s$. ∎

Register Bound

After the spilling phase, the liveness information in the program changed tremendously. Spills and loads introduce new live ranges, and shorten live ranges of already defined variables. To prove correctness of the spilling predicate, we must show that after spilling the register pressure is lowered to k. To formally establish the bound, we show that the number of variables from $\mathcal{V}_R$ in each liveness annotation in the spilled program is bounded by k. The following observation is key to this proof: The live-ins of a function after spilling can be obtained from the live-ins of the function before spilling by keeping the variables passed in registers, and adding the slots of the variables passed in memory. This property can be seen in the rule SpillFun, where we require $R_f \cup M_f = X_f$.

In the Coq development, the statements of the following lemmas involve the algorithm that reconstructs minimal liveness information from the annotations at function definitions, which we omitted in this presentation for the sake of simplicity.

Lemma 13.3 Let $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s \vdots \mathbf{c} : \mathbf{X}$ and $\mathbb{M}\Sigma \setminus Z \vdash \mathbf{live_I}\ s : \mathbf{X}$ and let s be renamed apart and let all variables in s be in $\mathcal{V}_R$. If $\bigcup Z \cup R \cup M \subseteq \mathcal{V}_R$ then there is a liveness annotation $\mathbf{X}'$ such that $\mathbb{M}\Sigma \setminus Z \vdash \mathbf{live_I}$ (doSpill $Z\ \Sigma\ s\ \mathbf{c}$) : $\mathbf{X}'$.

Proof. By induction on $\mathbf{spill}_k s$; mostly simple but tedious set constraints. ∎

Lemma 13.4 Let $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s \vdots \mathbf{c} : \mathbf{X}$ and $\mathbb{M}\Sigma \setminus Z \vdash \mathbf{live_I}\ s : \mathbf{X}$ and let s be renamed apart and let all variables in s be in $\mathcal{V}_R$ and $|R| \leq k$ and $\bigcup Z \cup R \cup M \subseteq \mathcal{V}_R$. Let $\mathbb{M}\Sigma \setminus Z \vdash \mathbf{live_I}$ (doSpill $Z\ \Sigma\ s\ \mathbf{c}$) : $\mathbf{X}'$ be the liveness derivation from the conclusion of Lemma 13.3. Then for every live set X in $\mathbf{X}'$ the bound $|\mathcal{V}_R \cap X| \leq k$ holds.

Proof. By induction on s. The proof uses a technical lemma about the way the liveness reconstruction deals with forward-propagation that was difficult to find. ∎

Semantic Equivalence

In this subsection we show that the spilled program is semantically equivalent to the original program. Semantic equivalence means trace-equivalence à la CompCert. As proof tool we use a co-inductively defined simulation relation. See our

previous work [SSH15; SSH16] for details on simluation and proof technique. The verification is done with respect to the imperative semantics of IL. This allows for a simple treatment of of the new variables that each spill and each load introduces. A typical spill and reload looks as follows:

```
let x = 5 in
fun f () = x in

...

f()
```

```
let x = 5 in
fun f () = x in
...
let X = x in    ⟵ spill
...
let x = X in    ⟵ load
f()
```

Note that in a semantics with binding, serious effort would be requires to introduce additional function parameters after spilling and loading. In the above example, f would need to take x as a parameter. We postpone the introduction of additional parameters to a phase after spilling, where we switch to the functional semantics again to do register allocation. Changing the semantics from imperative to functional corresponds to SSA construction and is in line with practical implementations of SSA-based register allocation [BH09] that break the SSA invariant during spilling, and then perform some form of SSA (re-)construction.

We need the following function to replicate argument values for arguments that are passed in both register and spill slot positions.

$$\begin{aligned}
&\mathsf{exarg}\,(R, M)\,\mathsf{nil} = \mathsf{nil}\\
&\mathsf{exarg}\,(R, M)\,(v_1, \overline{v}) = \begin{cases} v_1, v_1, \mathsf{exarg}\,(R, M)\,\overline{x} & \text{if } z \in R \cap M\\ v_1, \mathsf{exarg}\,(R, M)\,\overline{x} & \text{otherwise}\end{cases}
\end{aligned}$$

Definition 13.5 We define the proof relation $\mathcal{P}_{\text{spill}}$ where

$$\begin{aligned}
A &:= \mathsf{set}\,\mathcal{V} \times \mathsf{set}\,\mathcal{V} \times \mathsf{list}\,\mathcal{V}\\
\mathit{Param}\,(R, M, \overline{x})\,\overline{y}\,\overline{y'} &:= \overline{y} = \overline{x} \wedge \overline{y'} = \mathsf{exparm}\,(R, M)\,\overline{x}\\
\mathit{Arg}\,V\,V'\,(R, M, \overline{x})\,\overline{v}\,\overline{v'} &:= V =_{R\setminus\overline{x}} V' \wedge V =_{M\setminus\overline{x}} (\lambda x.V'(\mathsf{slot}\ x))\\
&\quad \wedge |\overline{x}| = |\overline{v}| \wedge V' \text{ defined on } R \setminus \overline{x} \cup \mathsf{slot}\,(M \setminus \overline{x})\\
&\quad \wedge \overline{v'} = \mathsf{exarg}\,\overline{v}\,\overline{x}\,(R, M)\\
\mathit{Idx}\,_\,f\,f' &:= f = f'
\end{aligned}$$

Lemma 13.6 Let s be a program where all variables are renamed apart and in $\mathcal{V}_R$, and $\mathbf{c}$ be the corresponding spilling annotation. Let $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s : \mathbf{c} : \mathbf{X}$ and $\mathbb{M}\Sigma \setminus Z \vdash \mathbf{live}_\mathbf{I}\ s : \mathbf{X}$ and $V =_R V'$ and $V =_M (\lambda x.V'(\mathsf{slot}\ x))$. If V' is defined on $R \cup \mathsf{slot}\ M$ and $R \cup M \subseteq \mathcal{V}_R$ and $L \gtrsim_r L' :^{\mathcal{P}_{\text{spill}}} \mathsf{zip}\ \Sigma\,\zeta$ then $(L, V, s)_\mathrm{I} \gtrsim_r (L', V', \mathsf{doSpill}\ Z\,\Sigma\,s\,\mathbf{c})_\mathrm{I}$.

13.3.4 Case Study: Verified Spilling Algorithms

A spilling algorithm generates a valid spilling annotation from a program with sound liveness annotation. The following algorithms are implemented in Coq and verified using the correctness predicate. The compactness of the correctness proofs shows how effectively our spilling framework reduces the correctness proof size for spilling algorithms.

SimpleSpill

The naive spilling algorithm simpleSpill loads the required values before each statement, without considering that the value might still be available in a register. After a variable is assigned, the algorithm immediatelly spills the variable. This is a very simple algorithm, and it corresponds to the spilling strategy used in the very first version of CompCert [Ler09a].

Theorem 13.7 Let $\mathbb{M}\Sigma \setminus Z \vdash \mathbf{live_I}\ s : \mathbf{X}$ and let s be renamed apart and let all variables in s be in $\mathcal{V}_R$ and let every expression in s contain at most k different variables. If every live set X in s is bounded by $R \cup M$ and the first component in Σ_f is empty for every f then $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill}_k\ s : (\text{simpleSpill}\ R\, s\, \mathbf{X}) : \mathbf{X}$.

Proof. By induction on s in less than 100 lines. ■

SplitSpill

The spilling algorithm splitSpill follows three key ideas: Variables are loaded as late as possible, but in contrast to simpleSpill, only values not already available in registers. If a register must be freed for a reload, the algorithm lets an oracle choose the variable to be spilled from the list of variables live and currently in a register. The correctness requirement for the oracle is trivial. The oracle enables live range splitting based on an external heuristic, similar to the approach of Braun [BH09]. In contrast to Braun's algorithm, splitSpill cannot hoist reloads from their uses.

Theorem 13.8 Let $\mathbb{M}\Sigma \setminus Z \vdash \mathbf{live_I}\ s : \mathbf{X}$ and let s be renamed apart and let all variables in s be in $\mathcal{V}_R$ and let every expression in s contain at most k different variables. If every live set X in s is bounded by $R \cup M$ and for every f such that $(R_f, M_f) = \Sigma_f$ we have $|R_f| \leq k$ then $\Sigma \mid Z \mid R \mid M \vdash \mathbf{spill}_k\ s : (\text{splitSpill}\ k\, Z\, (\mathbb{M}\Sigma \setminus Z)\, R\, M\, s\, \mathbf{X}) : \mathbf{X}$.

Proof. By induction on s in less than 500 lines. ■

$$\textsc{Inj-Op}\ \frac{\rho \rightarrowtail X \qquad \rho \vdash \mathbf{inj}\ s}{\rho \vdash \mathbf{inj}\ \mathtt{let}\ x = \eta\ \mathtt{in}\ s : X} \qquad \textsc{Inj-Cond}\ \frac{\rho \rightarrowtail X \qquad \rho \vdash \mathbf{inj}\ s \qquad \rho \vdash \mathbf{inj}\ t}{\rho \vdash \mathbf{inj}\ \mathtt{if}\ x\ \mathtt{then}\ s\ \mathtt{else}\ t : X}$$

$$\textsc{Inj-Val}\ \frac{\rho \rightarrowtail X}{\rho \vdash \mathbf{inj}\ e : X} \qquad \textsc{Inj-App}\ \frac{\rho \rightarrowtail X}{\rho \vdash \mathbf{inj}\ f\ \overline{y} : X} \qquad \textsc{Inj-Fun}\ \frac{\rho \rightarrowtail X \qquad \rho \vdash \mathbf{inj}\ s \qquad \rho \vdash \mathbf{inj}\ t}{\rho \vdash \mathbf{inj}\ \mathtt{fun}\ f\ \overline{x} = s\ \mathtt{in}\ t : X}$$

Figure 13.3: Local Injectivity. $\rho : \mathcal{V} \to \mathcal{V}$ is the register assignment, s is a program, and X is a set of live variables. We use the notational convention described in §4.8.2.

13.4 Register Assignment

We present an algorithm that establishes coherence while preserving IL/F semantics and improves over the naive approach outlined in §10.4.1 by using no more different names than the size of the maximal live set in the program. This algorithm corresponds to the assignment phase of SSA-based register allocation [HGG06]. The algorithm requires a renamed-apart program as input to ensure that every consistent renaming can be expressed as a function from $\mathcal{V} \to \mathcal{V}$. We proceed in two steps:

1. We define the notion of *local injectivity* for a function $\rho : \mathcal{V} \to \mathcal{V}$ relative to the liveness information for an IL program s. We show that renaming s with a locally injective ρ yields an α-equivalent and coherent program $\rho\, s$.
2. We give an algorithm rassign and show that it constructs a locally injective ρ that uses the minimal number of different names.

13.4.1 Local Injectivity

We use the following notation for injectivity on X:

$$f \rightarrowtail X \;:\leftrightarrow\; \forall x\, y \in X,\ f\, x = f\, y \to x = y$$

We define local injectivity judgment $\rho \vdash \mathbf{inj}\ s : \mathbf{X}$ inductively according to the rules in Figure 13.3, where $\rho : \mathcal{V} \to \mathcal{V}$ is a register assignment, s is a program, and X is liveness annotation. The rules of the judgment in Figure 13.3 and require ρ to be injective on every live set X annotating any subterm.

Let $\mathcal{V}_B(s)$ be the set of variables that occur in a binding position in s, and fv s be the set of free variables of s. For our theorems, several properties are required:

(1) A variable in $\mathcal{V}_B(s)$ must not occur in a set of globals in Λ. We define $\Lambda \subseteq U \;:\leftrightarrow\; \forall f \in \mathit{dom}\,\Lambda,\ \Lambda\, f \subseteq U$.

(2) A variable in $\mathcal{V}_B(s)$ must not occur in the annotation $[s]$. We write $s \subseteq U$ if for every subterm t of s it holds that every $x \in [t]$ is either in U or bound at t in s.

For renamed-apart programs, these conditions ensure that the live set X in Inj-Fun always contains the globals X_1 of f (cf. Live-App).

Theorem 13.9 Let s be a renamed-apart program such that $\mathbf{noUC}_{\mathsf{isCalled}}\ s$ and such that $\Lambda \vdash \mathbf{live_I}\ s : \mathbf{X}$ and $\Lambda \subseteq \mathsf{fv}\ s$ and $s \subseteq \mathsf{fv}\ s$. Then

$$\rho \vdash \mathbf{inj}\ s : \mathbf{X} \;\to\; \rho\,(\lfloor \Lambda \rfloor_{[s]}) \vdash \mathbf{coh}\ (\rho\, s) : (\rho\, \mathbf{X})$$

Theorem 13.9 states that the renamed program $\rho\, s$ is coherent under the assumptions $\rho\,(\lfloor \Lambda \rfloor_{[s]})$ and $\rho\, \mathbf{X}$, i.e. the point-wise image under ρ.

Renaming with a locally injective renaming produces an α-equivalent program, and hence preserves program equivalence:

Theorem 13.10 Let s be a renamed-apart program such that $\mathbf{noUC}_{\mathsf{isCalled}}\ s$ and $\Lambda \vdash \mathbf{live_I}\ s : \mathbf{X}$ and $\Lambda \subseteq \mathsf{fv}\ s$ and $s \subseteq \mathsf{fv}\ s$. Let $\rho, d : \mathcal{V} \to \mathcal{V}$ such that ρ is the inverse of d on $\mathsf{fv}\ s$. Then $\rho \vdash \mathbf{inj}\ s : \mathbf{X} \;\to\; \rho, d \vdash \rho\, s \sim_\alpha s$

The formal definition of α-equivalence can be found in the Coq development, together with a soundness proof and a formal treatment of renaming, renaming apart, and a formal definition of being renamed apart.

13.4.2 A Simple SSA-based Register Assignment Algorithm for IL

The algorithm rassign is parametrized by a function $\mathsf{fresh} : \mathsf{set}\,\mathcal{V} \to \mathcal{V}$ of which we require $\mathsf{fresh} X \notin X$ for all finite sets of variables X. Based on fresh, we define a function $\mathsf{freshlist}\, X\, n$ that yields a list of n pairwise-distinct variables such that $(\mathsf{freshlist}\, X\, n) \cap X = \emptyset$. SSA algorithm must process the program in an order compatible with the dominance order to work [HGG06]. In our case it suffices to simply recurse on s as follows:

$$\begin{array}{lcl}
\mathsf{rassign}\,\rho\,(\mathtt{let}\ x = \eta\ \mathtt{in}\ s) & = & \mathsf{rassign}\,(\rho[x \mapsto y])\,s \\
 & & \quad \text{where } y = \mathsf{fresh}\,(\rho([s] \setminus \{x\})) \\
\mathsf{rassign}\,\rho\,(\mathtt{if}\ e\ \mathtt{then}\ s\ \mathtt{else}\ t) & = & \mathsf{rassign}\,(\mathsf{rassign}\,\rho\,s)\,t \\
\mathsf{rassign}\,\rho\,(e) & = & \rho \\
\mathsf{rassign}\,\rho\,(f\,\overline{e}) & = & \rho \\
\mathsf{rassign}\,\rho\,(\mathtt{fun}\ f\,\overline{x} = s\ \mathtt{in}\ t) & = & \mathsf{rassign}\,(\mathsf{rassign}\,(\rho[\overline{x} \mapsto \overline{y}])\,s)\,t \\
 & & \text{where } \overline{y} = \mathsf{freshlist}\,(\rho([s] \setminus \overline{x}))\,|\overline{x}|
\end{array}$$

We prove in Theorem 13.11 that the algorithm is correct for any choice of *fresh* and *freshlist*, as long as they satisfy the specifications above.

Theorem 13.11 Let s be renamed-apart such that $\mathbf{noUC}_{\mathsf{isCalled}}\ s$ and $\Lambda \vdash \mathbf{live}_{\mathbf{I}}\ s : \mathbf{X}$ and $\Lambda \subseteq \mathsf{fv}\ s$ and $s \subseteq \mathsf{fv}\ s$. Let ρ be injective on $[X]$. Then: $\mathsf{rassign}\ \rho\ s \vdash \mathbf{inj}\ s : \mathbf{X}$.

Our implementation of fresh implements the heuristic of simply choosing the smallest unused variable. Corollary 13.13 shows that for this choice of *fresh*, the largest live set determines the number of required names. We use $\mathcal{S}(k)$ to denote the set of the k smallest variables, and $\mathcal{V}_O(s)$ to denote the set of variables occurring (free or in a binding position) in s.

Theorem 13.12 Let fresh X always yield a variable less or equal to $|X|$. Let s be renamed-apart and $\Lambda \vdash \mathbf{live}_{\mathbf{F}}\ s : \mathbf{X}$ and $\Lambda \subseteq \mathsf{fv}\ s$ and $s \subseteq \mathsf{fv}\ s$ and $\rho \rightarrowtail [\mathbf{X}]$. Let k be the size of the largest set of live variables in s, and $\mathsf{rassign}\ \rho\ s = \rho'$. If $\rho'[\mathbf{X}] \subseteq \mathcal{S}(k)$, then $\mathcal{V}_O(s) \subseteq \mathcal{S}(k)$.

Proof. By induction on s. ■

For LVC, we need Theorem 13.12 to hold for imperative liveness information. We get this as corollary using results about the absence of unreachable code §11.5, and the fact that unreachable code was eliminated before register allocation.

Corollary 13.13 Let fresh X always yield a variable less or equal to $|X|$. Let s be renamed-apart such that $\mathbf{noUC}_{\mathsf{isCalled}}\ s$ and $\Lambda \vdash \mathbf{live}_{\mathbf{I}}\ s : \mathbf{X}$ and $\Lambda \subseteq \mathsf{fv}\ s$ and $s \subseteq \mathsf{fv}\ s$ and $\rho \rightarrowtail [\mathbf{X}]$. Let k be the size of the largest set of live variables in s, and $\mathsf{rassign}\ \rho\ s = \rho'$. If $\rho'[\mathbf{X}] \subseteq \mathcal{S}(k)$, then $\mathcal{V}_O(s) \subseteq \mathcal{S}(k)$.

Proof. From Theorem 13.12 with Theorem 11.8. ■

Theorem 13.12 provides an opportunity for translation validation with repair: The fresh function has some leeway to choose which variable to use as a register, as long as the variable is small enough. Hence, an unverified function could be used instead of fresh to choose a candidate variable, which is accepted if it is small enough and not currently in use; otherwise the default implementation of fresh currently included in LVC can be used to obtain a safe choice.

13.5 Argument Elimination and Lowering to Parallel Moves

In this section we informally discuss the elimination of arguments on IL/I to parallel moves, and the lowering of the parallel moves to sequences of assignments. We implement a transformation that performs both steps at once and relies on CompCert's component [RSL08] for lowering parallel moves to sequences of assignments.

We modify the approach such that slot-to-slot assignments, which register allocation might introduce, are lowered to loads and stores.

13.5.1 Parameter Passing as Parallel Moves

The first important intuition is that argument passing in IL/I effectively constitutes a parallel move. Consider a function f with formal parameters $\overline{x}$ and the corresponding application $f\overline{y}$. Assuming $Lf = (\overline{x}, s)$, executing the application in IL/I results in the following reduction step:

$$(L, V, f\,\overline{e}) \quad \longrightarrow_{\mathrm{F}} \quad (L^f, V[\overline{x} \mapsto V\,\overline{y}], s)$$

The environment $V[\overline{x} \mapsto V\,\overline{y}]$ could have equivalently been obtained from V by executing the parallel moves $\overline{x} \leftarrow \overline{y}$. We can hence implement IL/I function application parallel moves plus a function application without arguments.

13.5.2 Lowering Parallel Moves to Assignments

CompCert's component [RSL08] for implementing parallel moves requires a temporary register to break cyclic dependencies in the parallel moves. The use of a temporary can in general not be avoided, if one does not assume the availability an instruction or an instruction sequence that realizes a swap of two registers.

Example 13.14 To lower the parallel moves $X, Y, r \leftarrow Y, r, X$ to a sequence of assignments, a temporary register$t \notin \{r, X, Y\}$ is required:

$$t := X;\ X := Y;\ Y := r;\ r := t$$

Furthermore, as both sides of the parallel moves may contain slot variables as per our spilling pass, the resulting sequence of assignments may contain slot to slot assignments, as also show in Example 13.14, where the slot-to-slot assignment $X := Y$ appears. Target architectures cannot be assumed to have instructions for memory to memory moves, so slot-to-slot moves must be lowered into two assignments which use a temporary register. Our current approach is to use another register r and implement a slot-to-slot move $X := Y$ with two assignments $r := Y$ and $X := r$.

Example 13.15 To lower the parallel moves $r, X, Y \leftarrow X, Y, r$ to a sequence of assignments without slot-to-slot moves, two temporaries $u, t \notin \{r, X, Y\}$ are required, of which at least one is a register. We assuming t, u are both registers, the resulting sequence of assignments is

$$t := X;\ u := Y;\ X := u;\ Y := r;\ r := t$$

Assuming u is a register, and T is a slot, the resulting sequence of assignments is

$$u := X;\ T := u;\ u := Y;\ X := u;\ Y := r;\ r := T$$

We take the following approach to implement the parallel moves $\overline{x} \leftarrow \overline{y}$:

1. We reserve two unused temporary slots T_{pm} and T_{r} such that $x, y \notin \overline{x} \cup \overline{y}$.
2. We obtain a temporary register as follows:
   a. If there is a register r that is not live before the parallel move, and $r \notin \overline{x}$, then we use CompCert's component with T_{pm} as temporary to obtain a sequence of assignments, from which we eliminate slot-to-slot assignments as described above using r as temporary.
   b. If there is a register r that is not live before the parallel move, but $r \in \overline{x}$, we run CompCert's component with T_{pm} as temporary on the parallel move

      $$\overline{x}[r \mapsto T_{\text{r}}] \leftarrow \overline{y}$$

      that is, the parallel move where r has been replaced by the temporary T_{r}. Note that since r is not live, $r \notin \overline{y}$, which means that r is not read. We implement slot-to-slot assignments in the result using the temporary r, and append an additional assignment $r := T_{\text{r}}$ at the end.
   c. If there is no register r that is not live before the parallel move, we run CompCert's component with T_{pm} as temporary on the parallel move

      $$\overline{x}[r \mapsto T_{\text{r}}] \leftarrow \overline{y}[r \mapsto T_{\text{r}}]$$

      that is, the parallel move where r has been replaced by the temporary T_{r}. We implement slot-to-slot assignments in the result using the temporary r, prepend an additional assignment $T_{\text{r}} := r$ at the beginning, and append an additional assignment $r := T_{\text{r}}$ at the end.

Note that we must distinguish between case b and c, as the prepended assignment $T_{\text{r}} := r$ in case c gets stuck if r is not defined, which might, in general, be true in case b.

The informal description we gave here is realized in Coq and verified.

Theorem 13.16 Lowering of parallel moves as described informally here is correct.

Note that the proof of Theorem 13.16 requires function parameters to be duplicate free. This stems from a slight mismatch in the semantics of parallel moves and IL's parameter passing. Parameter passing is specified as a two phase process, where first all arguments are evaluated to values, and then the values are assigned to the parameters from right to left, i.e. if a variable is appears twice in on the left-hand side, the value of the earliest occurrence "wins". In contrast, the specification of parallel moves from CompCert does not give semantics to a parallel assignment where a variable appears twice on the right-hand side.

13.5.3 On Optimality and Possible Improvements

While the solution we chose is not optimal with respect to the number of assignments and the number of temporaries used in some cases, Example 13.15 shows that there are cases which require two temporaries. CompCert's component for parallel moves also does not produce optimal code in all cases. Its authors give an example in the conclusion of their paper [RSL08]. Optimal approaches are documented in literature [May89], but are not verified yet.

The particular problem that arises in LVC, in contrast to the problem CompCert needs to solve, also needs handling of slot-to-slot moves. We would be interested in an algorithm that lowers parallel moves that may contain slot-to-slot moves in such a way that the resulting sequence of assignments does not contain slot-to-slot assignments, and uses as few as possible temporary registers if no registers are available. Also, we would like to allow the lowering algorithm to deal with constants as well, i.e. on the right-hand sides of the initial parallel move certain very simple expressions should be allowed.

As discussed earlier, an algorithm for lowering parallel moves could handle duplicate occurrences of the same variable on the left hand side by, for example, only considering the first assignment to the variable and disregarding the others.

Furthermore, as the implementation of parallel moves is performance critical simply because copies are very frequent in generated code, a good lowering algorithm could produce an order that allows the processor to dispatch several assignments to different functional units of the processor in parallel. This can be achieved by simply avoiding dependencies between consecutive assignments whenever possible.

14 Value Optimizations

In this section, we devise a simple program logic for IL especially tailored towards the verification of value optimizations. We call the logic *value optimization logic*, or vopt for short. The statements of the logic express equivalence of two programs. The logic has syntactic proofs for these statements, which do not rely on the IL's small step semantics but only on the evaluation semantics of IL's expressions. The first main result is the soundness of the logic, which shows that whenever a statement that relates two programs is provable in the logic, then the two programs are equivalent with respect to IL semantic equivalence.

The main motivation for the logic is the observation that many SSA optimizations simply replace an expression e with another expression e'. Sparse conditional constant propagation [WZ91a], for example, replaces variables that are known to be constant with the respective value. Usual formulations of SSA optimizations are also concerned with the removal of definitions that become unused, but removal can delegated to DVE/UCE, which we discussed in §9.3 and §11.4. Ignoring the code removal part, the transformation a typical SSA optimization performs is to replace expressions by other expressions. We now preset a program logic that abstracts this principle through a notion of approximation $\sqsubseteq$ and $\equiv$ between expressions. In the logic, it is sound to replace an expressions e with an expressions e' if $e \sqsubseteq e'$, that is, if whenever e' yields a value, then e yields the same value. The main advantage of using the logic over a direct simulation proof is that the logic deals with the correctness argument for function definitions once and for all in the soundness lemma for the logic.

We implemented the logic with a deep embedding, because our initial goal was to automate the verification of optimizations with the help of an SMT solver. In joint work with Heiko Becker, we showed that the notions of approximation $\sqsubseteq$ and $\equiv$ are decidable with the help of an SMT solver. Unfortunately, we did not have time to finish integration of the two parts, so at the moment LVC does not offer SMT-based translation validation.

As case studies, we use the logic to verify two value optimizations, namely copy propagation and sparse conditional constant propagation [WZ91a]. The verification of these optimiztions consists of devising a procedure that constructs a proof of the appropriate equivalence statement in the logic.

14.1 The Value Optimization Logic

The program logic is based on assertions γ over IL expressions e of the form

$$\gamma : \Gamma \ ::= \ e \equiv e \mid e \sqsubseteq e \mid \bot \mid \top \mid \gamma \wedge \gamma$$

We say a variable environment E models an assertion γ if the judgment $E \models \gamma$ holds, which is defined as follows:

$$\begin{aligned}
E \models e \equiv e' &:\leftrightarrow [\![E]\!]\, e = [\![E]\!]\, e' \wedge [\![E]\!]\, e \neq \bot \\
E \models e \sqsubseteq e' &:\leftrightarrow [\![E]\!]\, e \sqsubseteq [\![E]\!]\, e' \\
E \models \bot &:\leftrightarrow \bot \\
E \models \top &:\leftrightarrow \top \\
E \models \gamma \wedge \gamma' &:\leftrightarrow E \models \gamma \wedge E \models \gamma'
\end{aligned}$$

The definition of equality on the assertions requires definedness: Undefined values are not equal to themselves, and assertion equality is not reflexive. However, this is an advantage, as we can require an expression e to be defined by asserting $e \equiv e$. The definition of assertion approximation $\sqsubseteq$, however, permits the left assertion to be undefined. This allows value optmizations to resolve underspecification: If the left side is undefined, any expression satisfies the assertion.

Note that the formulas γ we define are essentially sets of assertions, as we only allow conjunction.

We define the semantic notions of implication and validity for assertions:

$$\begin{aligned}
\gamma \Rightarrow \gamma' &:\leftrightarrow \forall E,\ E \models \gamma \to E \models \gamma' && \text{implication} \\
\models \gamma &:\leftrightarrow \forall E,\ E \models \gamma && \text{validity}
\end{aligned}$$

14.1.1 Substitutivity

The main ingredient for the soundness proof of the logic are the substitution lemmas in this section. The following lemmas state that we can replace the variables in an expression with either a value, or an expression that evaluates to that value.

Lemma 14.1 ■ Substution for Expressions
$[\![e_1]\!]\,(E[\overline{x} \mapsto [\![\overline{e_2}]\!]\, E]) = [\![e_1[\overline{x} \mapsto \overline{e_2}]]\!]\, E.$

Proof. Induction on e_1. ■

Lemma 14.2 ■ Substitution for Assertions
$E \models \gamma[\overline{x} \mapsto \overline{e}] \ \leftrightarrow \ E[\overline{x} \mapsto [\![\overline{e}]\!]\, E] \models \gamma.$

Proof. With Lemma 14.1 by induction on γ. ∎

We define the free variables $\mathsf{fv}\ \gamma$ of an assertion γ as the union of the free variables of the contained expressions.

Lemma 14.3 ▪ Invariance under Agreement
$E =_{\mathsf{fv}\ \gamma} E' \ \rightarrow\ E \models \gamma \leftrightarrow E' \models \gamma$.

14.1.2 Inductive Predicate for ValueOpt

We define a judgment $Z \mid H \mid \Gamma \vdash \mathbf{vopt}\ s \,/\, s'$, which will serve to assert that s and s' are equivalent. The judgment does not allow the program s' to differ much from s: Basically, s' must be obtained from s by replacing any number of expressions e in s with expression e' such that whenever e is evaluated to a value during execution, e' yields the same value. To statically ensure this is the case, the judgment maintains an assertion γ that records about the variable bindings. Whenever an expression is replaced, the judgment requires that $\gamma \Rightarrow e \sqsubseteq e'$. To simplify the handling of variable names, we will assume all programs s to be renamed apart. The rules defining the judgment are given in Figure 14.1.

Lemma 14.4 ▪ Monotonicity of vopt
Let s be a renamed apart program. If the judgment $Z \mid H \mid \gamma \vdash \mathbf{vopt}\ s \,/\, s'$ holds and $\gamma' \Rightarrow \gamma$ then $Z \mid H \mid \gamma' \vdash \mathbf{vopt}\ s \,/\, s'$.

14.1.3 Soundness of the logic

We are now concerned with the soundness proof for the logic. That is, we want to show that if $Z \mid H \mid \Gamma \vdash \mathbf{vopt}\ s \,/\, s'$ holds, then s' implements s. We use the inductive method and start by defining the proof relation $\mathcal{P}_{\mathbf{vopt}}$.

Definition 14.5 We define the proof relation $\mathcal{P}_{\mathbf{vopt}}$ where

$$
\begin{aligned}
A &:= \mathsf{list}\ \mathcal{V} \times \Gamma \\
\mathit{Param}\ (\overline{y}, \gamma_f)\ \overline{x}\ \overline{x}' &:= \overline{y} = \overline{x} \wedge \overline{x} = \overline{x}' \\
\mathit{Arg}\ E\ E'\ (\overline{y}, \gamma_f)\ \overline{v}\ \overline{v}' &:= |\overline{y}| = |\overline{v}| \wedge \overline{v} = \overline{v}' \wedge \\
&\quad\ \exists \gamma\ \overline{e}\ V,\ \overline{e} \equiv \overline{e} \wedge \gamma \Rightarrow \gamma_f[\overline{y} \mapsto \overline{e}] \wedge \\
&\quad\ V \models \gamma \wedge [\![\overline{e}]\!]\ V = \overline{v} \wedge E =_{\mathsf{fv}\ \gamma_f \setminus \overline{y}} V \\
\mathit{Idx}\ a\ f\ f' &:= f = f'
\end{aligned}
$$

The parameter relation and the index relation encode that function parameters and names cannot be changed. Recall that the argument relation encodes what must hold at a call site. Here, a function can be applied in an environment V that satisfies an assertion γ with argument expressions $\overline{e}$ if $\overline{e}$ evaluates, V agrees with the closure

$$\textsc{VOpt-Let}\quad \frac{\gamma \Rightarrow e \sqsubseteq e' \qquad Z \mid H \mid \gamma \wedge x \equiv e \vdash \textbf{vopt}\ s \,/\, s'}{Z \mid H \mid \gamma \vdash \textbf{vopt}\ \texttt{let}\ x = e\ \texttt{in}\ s \,/\, \texttt{let}\ x = e'\ \texttt{in}\ s'}$$

$$\textsc{VOpt-If}\quad \frac{\gamma \Rightarrow e \sqsubseteq e' \qquad Z \mid H \mid \gamma \wedge ?e \equiv \textbf{true} \vdash \textbf{vopt}\ s \,/\, s' \qquad Z \mid H \mid \gamma \wedge ?e \equiv \textbf{false} \vdash \textbf{vopt}\ s \,/\, s'}{Z \mid H \mid \gamma \vdash \textbf{vopt}\ \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t \,/\, \texttt{if}\ e'\ \texttt{then}\ s'\ \texttt{else}\ t'}$$

$$\textsc{VOpt-App}\quad \frac{\gamma \Rightarrow \overline{e} \sqsubseteq \overline{e}' \qquad \gamma \wedge \overline{e} \equiv \overline{e} \Rightarrow H_f[Z_f \mapsto \overline{e}']}{Z \mid H \mid \gamma \vdash \textbf{vopt}\ f\,\overline{e} \,/\, f\,\overline{e}'} \qquad \textsc{VOpt-Return}\quad \frac{\gamma \Rightarrow e \sqsubseteq e'}{Z \mid H \mid \gamma \vdash \textbf{vopt}\ e \,/\, e'}$$

$$\textsc{VOpt-Extern}\quad \frac{\gamma \Rightarrow \overline{e} \sqsubseteq \overline{e}' \qquad Z \mid H \mid \gamma \wedge x \equiv x \vdash \textbf{vopt}\ s \,/\, s'}{Z \mid H \mid \gamma \vdash \textbf{vopt}\ \texttt{let}\ x = f\,\overline{e}\ \texttt{in}\ s \,/\, \texttt{let}\ x = f\,\overline{e}'\ \texttt{in}\ s'}$$

$$\textsc{VOpt-Fun}\quad \frac{\forall i,\ \mathsf{fv}\,(\gamma_i)\ \text{in scope} \qquad \forall i \in [1,n],\ Z \mid \overline{\gamma}; H \mid \gamma' \wedge \gamma_i \vdash \textbf{vopt}\ s_i \,/\, s_i' \qquad Z \mid H \mid \gamma' \vdash \textbf{vopt}\ t \,/\, t'}{Z \mid H \mid \gamma' \vdash \textbf{vopt}\ \texttt{fun}\ \overline{f\,\overline{x} = s}\ \texttt{in}\ t \,/\, \texttt{fun}\ \overline{f\,\overline{x} = s'}\ \texttt{in}\ t'}$$

$$\textsc{VOpt-Unsat}\quad \frac{\gamma \Rightarrow \bot}{Z \mid H \mid \gamma \vdash \textbf{vopt}\ s \,/\, s'}$$

Figure 14.1: Defining equations of the value optimization logic.

environment on the free variables of the closure assertion γ_f without the function parameters. Further the assertion that the argument expressions evaluate together with the assertion γ must imply that the closure assertion holds with the argument expressions substituted in.

We need another definition to record an invariant about closures. We say the closure invariant holds for L, V, H and s if for all f we have that $\mathsf{fv}\ H_f$ is disjoint from variables bound in s, and for all $L_f = (E, \overline{x}, t)$ we have that $E =_{\mathsf{fv}\ H_f \setminus \overline{x}} V$.

Theorem 14.6 ▪ Soundness of VOpt
If $V \models \gamma$ and $Z \mid H \mid \gamma \vdash \textbf{vopt}\ s \,/\, s'$ and s is renamed-apart and and $\mathsf{fv}\,(\gamma)$ are disjoint from the bound vars of s, and closure invariant holds for L, V, H and s then

$$\forall r,\ L \approx_r^{\mathsf{sim}} L' :^{\mathcal{P}_{\mathbf{vopt}}} Z, H \ \rightarrow\ (L, V, s)_{\mathrm{F}} \approx_r^{\mathsf{sim}} (L', V, s')_{\mathrm{F}}$$

Proof. The proof is by induction on the derivation of $Z \mid H \mid \gamma \vdash \textbf{vopt}\ s \,/\, s'$. Note that γ is still satisfied in updated environments that arise in the proof, because the program is renamed apart. It is clear that we can satisfy the argument relation at application, because of the rule VOpt-App. The closure invariant holds obviously for new closures. ■

In the bigger picture, Theorem 14.6 establishes the value optimization logic as an abstraction between the correctness proofs of transformations and the technical definitions of bisimilarity. We think such an abstraction is desirable given the fragility and technicality of coinductive proofs in Coq in general, and the likelyhood of the existence of a different and better semantic foundation with better properties. Once such a semantic foundation is available, the migration cost is mainly in reproving an apropriate version of Theorem 14.6, but the correctness proofs of the transformations building on the value optimization logic remain the same.

14.2 Copy Propagation

In this section we verify a simple optimization with the value optmization logic as a case study. The optimization is called copy propagation, and its ultimate goal is to remove let bindings of the form $\texttt{let}\ x = y\ \texttt{in}\ s$ where both x and y are variables. To achieve this, copy propagation replaces x with y in s, but keeps the let binding. A DVE post-pass can then take care of eliminating now unused binding.

We define copy propagation with a simple recursive algorithm. Note that this form of copy propagation cannot detect parameters that are copies of other variables, such as in the following example, where a more agressive version of copy propagation would eliminate the parameter x.

```
let y = z
fun f x = x
in if z then f y
        else f y
```

```

fun f x = x
in if z then f z
        else f z
```

The implementation of copy propagation is given in Figure 14.2.

14.2.1 Correctness

We show correctness of copy propagation by constructing a derivation of

$$Z \mid H \mid \gamma \vdash \textbf{vopt}\ s \,/\, \mathsf{cp}\ \varrho\ s$$

for an appropriate assertion γ. The assertion γ must encode the invariant that if $\varrho\, x = y$ and x is defined, then y has the same value. This invariant can be encoded

$$
\begin{aligned}
\mathsf{cp}\,\varrho\,(\texttt{let}\ x = y\ \texttt{in}\ s) &= \mathsf{cp}\,\varrho[x \mapsto \varrho y]\,s \\
\mathsf{cp}\,\varrho\,(\texttt{let}\ x = \eta\ \texttt{in}\ s) &= \texttt{let}\ x = \varrho\eta\ \texttt{in}\ \mathsf{cp}\,\varrho[x \mapsto x]\,s \\
\mathsf{cp}\,\varrho\,(\texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t) &= \texttt{if}\ \varrho e\ \texttt{then}\ \mathsf{cp}\,\varrho\,s\ \texttt{else}\ \mathsf{cp}\,\varrho\,t \\
\mathsf{cp}\,\varrho\,(f\,\overline{e}) &= f\,\varrho\overline{e} \\
\mathsf{cp}\,\varrho\,(e) &= \varrho e \\
\mathsf{cp}\,\varrho\,(\texttt{fun}\ \overline{f\,\overline{x} = s}\ \texttt{in}\ t) &= \texttt{fun}\ \overline{f\,\overline{x} = \mathsf{cp}\,\varrho[\overline{x} \mapsto \overline{x}]\,s}\ \texttt{in}\ \mathsf{cp}\,\varrho\,t
\end{aligned}
$$

Figure 14.2: Implementation of Copy Propagation.

as

$$\gamma = \bigwedge\{x \sqsubseteq \varrho x \mid x \in \mathsf{fv}\ s\}$$

Showing correctness of cp is fairly simple and uses only trivial properties of the assertion semantics. In the following, $\varrho\,e$ is the expression e where all variables in e have been replaced according to ϱ. The main lemma is the following property, which shows that it is sound to replace variables in an expression.

Lemma 14.7 If $\mathsf{fv}\ e \subseteq D$ then $\bigwedge\{x \sqsubseteq \varrho x \mid x \in D\} \Rightarrow e \sqsubseteq \varrho e$.

Proof. Induction on e. ■

Let $\hat{\top}$ be an environment that maps every function name to the assertion $\top$.

Theorem 14.8 Let s be a renamed apart program, and D be a set containing at least the free variables of s and disjoint from any variable bound in s. Let Z be a parameter environment for s. If $\varrho D \subseteq D$ then $Z \mid \hat{\top} \mid \bigwedge\{x \sqsubseteq \varrho x \mid x \in D\} \vdash \mathbf{vopt}\ s\ /\ \mathsf{cp}\,\varrho\,s$.

Proof. By size-induction on s with Lemma 14.7. We only show the two let-cases. For the first we have to show

$$Z \mid \hat{\top} \mid \bigwedge\{x \sqsubseteq \varrho x \mid x \in D\} \vdash \mathbf{vopt}\ \texttt{let}\ x = y\ \texttt{in}\ s\ /\ \texttt{let}\ x = y\ \texttt{in}\ \mathsf{cp}\,\varrho\,[x \mapsto \varrho y]s$$

We apply VOpt-Let and have to show its premises:

- The first premise requires us to show that under the current assumptions y can be replaced by y on the right-hand side of the assignment:

  $$\bigwedge\{x \sqsubseteq \varrho x \mid x \in D\} \Rightarrow y \sqsubseteq y$$

  This trivially holds because $\sqsubseteq$ is reflexive.

- The second premise requires us to show

$$Z \mid \hat{\top} \mid \bigwedge\{x \sqsubseteq \varrho x \mid x \in D\} \wedge x \equiv y \vdash \textbf{vopt}\ s \,/\, \mathsf{cp}\ \varrho\,[x \mapsto y]s$$

  We apply Lemma 14.4 and the inductive hypothesis with $\{x; D\}$. We have to show
  - $\varrho[x \mapsto \varrho y]\{x; D\} \subseteq \{x; D\}$, which holds because we have $\varrho D \subseteq D$ and $y \in D$ (because y is free) and from this $\varrho y \in D$.
  - For the use of monotonicity, we have to show

$$x \equiv y \wedge \bigwedge\{x \sqsubseteq \varrho x \mid x \in D\} \Rightarrow \bigwedge\{x \sqsubseteq \varrho[x \mapsto \varrho y]x \mid x \in \{x; D\}\}$$

    We have $\bigwedge\{x \sqsubseteq \varrho[x \mapsto \varrho y]x \mid x \in \{x; D\}\} = x \sqsubseteq \varrho y \wedge \bigwedge\{x \sqsubseteq \varrho x \mid x \in D\}$. It suffices to verify that $x \equiv y \wedge y \sqsubseteq \varrho y \Rightarrow x \sqsubseteq \varrho y$.

The second let-case follows similarly with the use of Lemma 14.7. ■

The use of vopt for the verification of copy propagation has the advantage that the semantics of IL does not appear in the proof, beyond, of course, the semantics of expression evaluation, which is encapsulated in Lemma 14.7. The proof is factorized along two independent axes. First, copy propagation leaves in the now unused bindings, and leaves the cleanup to a DVE pass. In particular, the correctness argument for $\mathsf{cp}\ \varrho\,(\texttt{let}\ x = y\ \texttt{in}\ s)$ does not need the invariant that x does not appear free in $\mathsf{cp}\ \varrho\,[x \mapsto \varrho y]s$, even though this invariant holds. Second, the use of the value optimization logic allows us to focus on the expression replacement.

14.3 Sparse Conditional Constant Propagation

As a second case study, we verify a more elaborate optimizations using the value optimization logic. Sparse Conditional Constant Propagation (SCCP) [WZ91a] is a seminal SSA-based optimization that remains very relevant in modern compilers. The optimization combines two types of analysis information: information about constantness of variables and reachability information. The two types of information interact with each other at conditionals: information about constant variables may be used to make reachability information more precise at conditionals by providing the value of the condition. In turn, information about constants becomes more precise because assignments in unreachable branches are not factored in.

14.3.1 Static Evaluation of Expressions

A key ingredient of SCCP is a static evaluation function ceval that evaluates expressions with respect to an environment that maps every variable to an element of the lattice $\mathbb{V}^{\top}_{\bot}$ depicted in Figure 14.3. The evaluation function

$$\mathsf{ceval} \ : \ (\mathcal{V} \to \mathbb{V}^{\top}_{\bot}) \to \mathsf{Exp} \to \mathbb{V}^{\top}_{\bot}$$

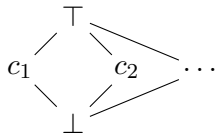

Figure 14.3: Lattice $\mathbb{V}_\bot^\top$ used as value domain in SCCP.

evaluates expressions as usual, but looks up the values of variables in an environment $\kappa : \mathcal{V} \to \mathbb{V}_\bot^\top$. If evaluation looks up a variable x and obtains $\top$, then the expression evaluation also returns $\top$. If evaluation looks up a variable x and obtains $\bot$, then the expression evaluation also returns $\bot$. Otherwise, all variables map to constants, and the value of the expression can be determined accordingly. We say an environment $E : \mathcal{V} \to \mathbb{V}_\bot$ conforms to an abstract environment $\kappa : \mathcal{V} \to \mathbb{V}_\bot^\top$ if forall x, $E\,x \sqsubseteq \kappa\,x$; here we implicitly life the co-domain $\mathbb{V}_\bot$ of E to $\mathbb{V}_\bot^\top$. We write $E \sqsubseteq \kappa$ if E conforms to κ, alluding to the point-wise lifting that is taking place.

Lemma 14.9 If $E \sqsubseteq \kappa$, then $[\![E]\!]\, e \sqsubseteq \mathsf{ceval}_\kappa\, e$.

14.3.2 Analysis

We specifying soundness for the analysis information with the inductive predicate $Z \mid K \vdash \mathbf{const}_\kappa\ s$.

14.3.3 Transformation

The SCCP transformation does not need reachability information, but relies only on the environment κ which maps variables to elements of $\mathbb{V}_\bot^\top$. We define a function cprop that selects the replacement value for an expression. cprop descends recursively through the expression and evaluates its subexpressions. If both subexpressions are constant, the result is computed and the corresponding constant is returned. Otherwise, the same operation with the two resulting subexpressions is returned.

The program tranformation proceeds as defined in Figure 14.5. Note that the transformation in fact eliminates constant parameters, but leaves the parameters in the program and relies on dead variable elimination to remove them in a later pass. This simplifies the correctness proof because we do not need an invariant that tracks the removed variables. In particular, we do not have to show that none of the variables that were removed are still needed.

$$\textsc{ConstProp-Let}\quad \frac{b_1 \to \mathsf{ceval}_c\, e = cx \qquad Z \mid K \vdash \mathbf{const}_\kappa\ s}{Z \mid K \vdash \mathbf{const}_\kappa\ \mathtt{let}\ x = e\ \mathtt{in}\ s}$$

$$\textsc{ConstProp-If}\quad \frac{Z \mid K \vdash \mathbf{const}_\kappa\ s \qquad Z \mid K \vdash \mathbf{const}_\kappa\ t}{Z \mid K \vdash \mathbf{const}_\kappa\ \mathtt{if}\, e\, \mathtt{then}\, s\, \mathtt{else}\, t}$$

$$\textsc{ConstProp-App}\quad \frac{|Z_f| = |\overline{e}| \qquad b \to \mathsf{ceval}_c\, \overline{e} \sqsubseteq C_f}{Z \mid K \vdash \mathbf{const}_\kappa\ f\, \overline{e}\{b\}}$$

$$\textsc{VOpt-Return}\quad \frac{}{Z \mid K \vdash \mathbf{const}_\kappa\ e}$$

$$\textsc{VOpt-Fun}\quad \frac{\forall i \in [1,n],\ \overline{\overline{x}}, Z \mid \overline{c\,\overline{x}}, K \vdash \mathbf{const}_\kappa\ s_i \qquad \overline{\overline{x}}, Z \mid \overline{c\,\overline{x}}, K \vdash \mathbf{const}_\kappa\ t}{Z \mid K \vdash \mathbf{const}_\kappa\ \mathtt{fun}\ \overline{f\,\overline{x}\ =\ s\{b\}}\ \mathtt{in}\ t}$$

Figure 14.4: Defining rules of the soundness predicate for constant propagation.

$$\begin{aligned}
\mathsf{cop}\,\kappa\,(\mathtt{let}\ x = e\ \mathtt{in}\ s) &= \mathtt{let}\ x = \mathsf{cprop}_\kappa\, e\ \mathtt{in}\ \mathsf{cop}\,\kappa\, s \\
\mathsf{cop}\,\kappa\,(\mathtt{let}\ x = f\,\overline{e}\ \mathtt{in}\ s) &= \mathtt{let}\ x = f\,(\mathsf{cprop}_\kappa\,\overline{e})\ \mathtt{in}\ \mathsf{cop}\,\kappa\, s \\
\mathsf{cop}\,\kappa\,(\mathtt{if}\, e\, \mathtt{then}\, s\, \mathtt{else}\, t) &= \mathtt{if}\ \mathsf{cprop}_\kappa\, e\ \mathtt{then}\ \mathsf{cop}\,\kappa\, s\ \mathtt{else}\ \mathsf{cop}\,\kappa\, t \\
\mathsf{cop}\,\kappa\,(f\,\overline{e}) &= f\,(\mathsf{cprop}_\kappa\,\overline{e}) \\
\mathsf{cop}\,\kappa\,(e) &= \mathsf{cprop}_\kappa\, e \\
\mathsf{cop}\,\kappa\,(\mathtt{fun}\ \overline{f\,\overline{x}\ =\ s}\ \mathtt{in}\ t) &= \mathtt{fun}\ \overline{f\,\overline{x}\ =\ \mathsf{cop}\,\kappa\, s}\ \mathtt{in}\ \mathsf{cop}\,\kappa\, t
\end{aligned}$$

Figure 14.5: Implementation of Constant Propagation.

14.3.4 Correctness

To show the transformation cop sound, we show an appropriate statement in the value optimization logic. The semantic correctness of SCCP then follows follows from the soundness of the program logic (Theorem 14.6). In this example, we see again the advantage of this approach: The proof will not require difficult invariants, even though the SCCP also detects constant function arguments.

Definition 14.10 We define the interpretation

$$\hat{\cdot} : (\mathcal{V} \to \mathbb{V}_\bot^\top) \to \mathsf{set}\,\mathcal{V} \to \mathsf{set}\,\Gamma$$

of an abstract environment κ as follows:

$$\begin{aligned} \hat{\kappa}\,X = \quad & \bigwedge\{x \equiv c \mid x \in X \wedge \kappa\,x = c\} \\ \wedge\ & \bigwedge\{\bot \mid x \in X \wedge \kappa\,x = \bot\} \end{aligned}$$

The following lemma connects abstract evaluation of expressions ceval with statements in the logic, which is useful for proving the conditional case in the correctness theorem for SCCP.

Lemma 14.11 Let $\hat{\kappa}\,(\mathsf{fv}\ e) \Rightarrow \gamma$. Then $v \sqsubseteq \mathsf{ceval}_\kappa\,e \to Z \mid H \mid e \equiv v \wedge \gamma \vdash \mathbf{vopt}\ s\ /\ s'$ implies $Z \mid H \mid e \equiv v \wedge \gamma \vdash \mathbf{vopt}\ s\ /\ s'$.

Theorem 14.12 Let s be a renamed apart program, and D be a set containing at least the free variables of s and disjoint from any variable bound in s. Let Z be a parameter environment for s. Then:

$$Z \mid \kappa\,Z \vdash \mathbf{const}_\kappa\ s \to Z \mid \hat{\kappa}\ Z \mid \hat{\kappa}\,D \vdash \mathbf{vopt}\ s\ /\ \mathsf{cop}\,\kappa\,s$$

Proof. Induction on the derivation of $Z \mid \kappa\,Z \vdash \mathbf{const}_\kappa\ s$. ■

15 Static Analysis

This section describes the static analysis framework we build for LVC. The framework has several building blocks. The basis is a type-class based library for partial orders and lattices (§3.9, §3.10) and a solver for fixed-point iteration for monotone functions on partial orders with finite height (§15.2). We use these building blocks to construct a general frameworks for forward dataflow analyses (§15.3), backward dataflow analyses (§15.5), and a framework specialized to SSA-based dataflow analyses (§15.7). The ideas uses here are not new; data-flow analyzers have been constructed in Coq before [Cac+05; Pic08; Jou+15].

15.1 Termination

Definition 15.1 ▪ **Termination**
Let X be a type and $R : X \to X \to \mathbf{P}$ be a relation. We define the predicate **ter** inductively:

$$\frac{\forall y, R\,x\,y \to \mathbf{ter}_R\,y}{\mathbf{ter}_R\,x}$$

Definition 15.2 Let X be a type and $R : X \to X \to \mathbf{P}$ be a relation. R is terminating if for all $x : X$ it holds $\mathbf{ter}_R\,x$. We say R is well-founded if its inverse R^{-1} is terminating.

The following lemmas establish termination of $\sqsubset$ for structures such as pairs, lists, etc.

Lemma 15.3 Let X, Y be a preordered types and let the respective relations $\sqsubset$ on X and Y be terminating. Then $\sqsubset$ on $X \times Y$ is also terminating.

Lemma 15.4 Let X be a preordered type and let the relation $\sqsubset$ on X be terminating. Then $\sqsubset$ on list X is also terminating.

Lemma 15.5 The relation $\sqsubset$ on $\mathbb{B}$ is terminating.

15.2 Finite Fixed-point Iteration

In this section we describe a finite fixed-point iteration scheme based on the Kleene fixed-point theorem.

Definition 15.6 A finite fixed-point iteration problem is a tuple (X, f, i) such that X is a preordered type, $f : X \to X$, and $i : X$ such that

1. $i \sqsubseteq fi$
2. $\sqsubset$ is terminating
3. f is monotone

Definition 15.7

Theorem 15.8 For every finite fixed-point iteration problem (X, f, i) there is $n \in \mathbb{N}$ and $x : X$ such that $f^n i = x$ and $fx = x$. Furthermore, there is a monotone function fix such that $x = \mathsf{fix}\, f\, i$ for all finite fixed-point iteration problems.

Proof. It is clear the following definition of fix' computes the limit of the Kleene iteration of f.

$$\mathsf{fix}'\, f\, y = \begin{cases} y & \text{if } fy \sqsubseteq y \\ \mathsf{fix}\, f\, (fy) & \text{otherwise} \end{cases}$$

However, we must enhance the type of fix' to make termination obvious. fix' terminates because every strict upward chain of X is finite by condition (3) of the iteration problem. Formally, we can give fix the type

$$\mathsf{fix} : \forall x : X, x \sqsubseteq fx \to \mathbf{ter}_{\sqsubset}\, x \to \Sigma\, y : X, n : \mathbb{N}, y = f^n i \wedge fy \equiv y$$

that recurs on the proof $\mathbf{ter}_{\sqsubset}\, x$ and extracts to fix'. ■

It is crucial to require f only to be monotone, but not expansive in the sense that $x \sqsubseteq fx$. We will use the finite fixed-point iteration to compute dataflow analyses, the transformers of which are monotone, but not expansive.

Although f is not expansive, monotonicity and the fact that the start value i satisfies $i \sqsubseteq fi$ are enough to ensure that each intermediate step of the Kleene fixed-point iteration lies on an upward chain.

Lemma 15.9 Let (X, f, i) be a finite fixed-point iteration problem and $n : \mathbb{N}$. Then: $f^n i \sqsubseteq f^{n+1} i$.

Lemma 15.9 yields an induction principle for fixed-point iteration:

Lemma 15.10 Let (X, f, i) be a finite fixed-point iteration problem and $n : \mathbb{N}$ and let $P : X \to \mathbf{P}$ be a property. To show $P\,(f^n i)$ it suffices to show the following:

- $P\,i$
- for all x such that $P\,x$ and $x \sqsubseteq fx$, it holds $P\,(fx)$

Proof. Induction on n and Lemma 15.9. ∎

15.3 A Framework for Forward Data Flow Analyses on IL

In general, the result of a data-flow analysis on an IL program is a annotation of the program that associates with every program point a value of some analysis domain D. We require the type of D to be a lower-bounded join-semi-lattice to ensure we have a bottom element, a join operation, and a notion of less-than. We will arrange things such that the result of the data-flow analysis can be computed by finite fixed-point iteration. Our framework provides a function fwd that monotonically transforms the whole annotation for a program and is suitable for finite fixed-point annotation. The function fwd is parametric in the type of the analysis domain D and local transformers.

The forward analysis function fwd maintains a parameter context ζ that holds the parameters each defined function. Additionally, it takes a program and a corresponding annotation as arguments, and produces a new annotation and a list of annotations (corresponding to analysis results for functions occurring free in the program) as result. The function fwd propagates the value of the analysis domain D in a forward fashion through the program, that is to say, in the direction of execution. In IL, the direction of execution corresponds to a recursive traversal of the program structure. We hence implement a forward analysis fwd as a recursive traversal of the program and its annotation. During traversal, fwd records new analysis information to produce the new annotation. The information at applications is propagated upwards and accumulated (joined) at function definitions. This is the purpose of the second return value of fwd: It is a context of analysis information that associates an analysis value to every defined function, and $\bot$ to functions that have not been called. The only communication between iterations happens at function definitions: All other annotations are overwritten with the new values (not joined). However, at function definitions all information from appliations is accumulated (joined) and subsequently recorded in the annotation. The next iteration uses this information as initial abstract value for the recursive analysis of the corresponding function bodies.

The function fwd is designed to be reused for forward analyses in general. For this porpose, the transformation is handled by a local transformer function t that is a parameter of fwd. The function t gets a program s and a value of type D and is

$$
\begin{aligned}
&\mathsf{fwd}\,\zeta\,(\texttt{let}\,x = \eta\,\texttt{in}\,s)\,(X \cdot a)\ = \\
&\quad \mathsf{let}\ X' = t\,\zeta\,(\texttt{let}\,x = \eta\,\texttt{in}\,s)\,X\ \mathsf{in} \\
&\quad \mathsf{let}\ a', A = \mathsf{fwd}\,\zeta\,s\,(\mathsf{setTopAnn}\,a\,X')\ \mathsf{in} \\
&\quad (X \cdot a', A) \\
&\mathsf{fwd}\,\zeta\,(\texttt{if}\,e\,\texttt{then}\,s_1\,\texttt{else}\,s_2)\,(X \cdot a_1, a_2)\ = \\
&\quad \mathsf{let}\ X_1', X_2' = t\,\zeta\,(\texttt{if}\,e\,\texttt{then}\,s_1\,\texttt{else}\,s_2)\,X\ \mathsf{in} \\
&\quad \mathsf{let}\ a_1', A_1 = \mathsf{fwd}\,\zeta\,s_1\,(\mathsf{setTopAnn}\,a_1\,X_1')\ \mathsf{in} \\
&\quad \mathsf{let}\ a_2', A_2 = \mathsf{fwd}\,\zeta\,s_2\,(\mathsf{setTopAnn}\,a_2\,X_2')\ \mathsf{in} \\
&\quad (X \cdot a_1', a_2', A_1 \sqcup A_2) \\
&\mathsf{fwd}\,\zeta\,(f\,\overline{e})\,X\ = \\
&\quad \mathsf{let}\ \mathsf{X}' = t\,\zeta\,(f\,\overline{e})\,X\ \mathsf{in} \\
&\quad (X, \bot[f \mapsto X']) \\
&\mathsf{fwd}\,\zeta\,e\,X\ =\ (X, \bot) \\
&\mathsf{fwd}\,\zeta\,(\texttt{fun}\,\overline{f\,\overline{x}\ =\ s}\,\texttt{in}\,t)\,(X \cdot \overline{a}, b)\ = \\
&\quad \mathsf{let}\ \zeta' = \overline{\overline{x}}; \zeta'\ \mathsf{in} \\
&\quad \mathsf{let}\ b', B = \mathsf{fwd}\,\zeta'\,t\,(\mathsf{setTopAnn}\,b\,X)\ \mathsf{in} \\
&\quad \mathsf{let}\ \overline{a'}, \overline{A} = \mathsf{fwdF}\,\zeta'\,\overline{s}\,\overline{a}\ \mathsf{in} \\
&\quad \mathsf{let}\ A' = B \sqcup \bigsqcup \overline{A}\ \mathsf{in} \\
&\quad (X \cdot \mathsf{setTopAnn}\,\overline{a'}\,A', b', \mathsf{drop}\,|\overline{f}|\,A') \\
&\mathsf{fwdF}\,\zeta\,\overline{s}\,\overline{a}\ = (\overline{a'}, \overline{A'})\ \text{such that} \\
&\quad \forall i, (a_i', A_i') = \mathsf{fwd}\,\zeta\,s_i\,a_i
\end{aligned}
$$

Figure 15.1: Definition of the forward analysis framework.

expected to transform the analysis value according to the top-most construction in s. Additionally, fwd provides the transformer t with the current parameter context ζ, which holds the parameter names of each defined function. For example, $t\,\zeta\,(\texttt{let}\ x = \eta\ \texttt{in}\ s)\,d$ yields a value of type D which corresponds to the analysis information before s, but after incorporating the effect of the variable definition. In the Coq development we made the retrun type of t depend on the program to allow t to the effect that t must return a tuple of two analysis values at conditionals, one for the consequence and one for the alternative. In the presentation of fwd in Figure 15.1 we omit types entirely, and give an essentially untyped version of the analysis function. The untyped version communicates the computational part clearly, and hides the book-keeping in the dependent type structure.

15.3.1 Forward Analysis as Finite Iteration Problem

In this section we show that we the forward analysis problem can be cast as a finite fixed-point iteration problem. The first ingredient is the fact that fwd is monotone.

Definition 15.11 We say a transformer t is monotone if $d \sqsubseteq d' \to t\,\zeta\,s\,d \sqsubseteq t\,\zeta\,s\,d'$.

Lemma 15.12 If the transformer t is monotone, then fwd is monotone in the sense that $d \sqsubseteq d' \to \mathsf{fwd}\,\zeta\,s\,d \sqsubseteq \mathsf{fwd}\,\zeta\,s\,d'$.

The last ingredient we need to cast data-flow analysis as a finite fixed-point iteration problem is the termination of the relation $\sqsubset$ on the analysis domain. The analysis domain must be chosen carefully to ensure this property. For example, in the case of a liveness analysis the domain is a set of variables and $\sqsubset$ is not terminating without further assumptions. Changing the domain to sets of variables bounded by the variables occuring in the program ensures that $\sqsubset$ is terminating. This, however, requires the transformer to stay inside the bound. The dependent type of fwd ensures that such properties can be easily established: The type of the analysis domain may depend on the program under analysis. The type for the transformer t is also dependent. The dependent types can be used to encode, for example, the transformer stays withing the variable bound for liveness analysis.

Theorem 15.13 Let D be a lower-bounded join-semi-lattice, and s be a program and $i : D$ such that $i \sqsubseteq \mathsf{fwd}\,\mathsf{nil}\,i$, and let t be monotone in the sense of Definition 15.11. Then $(\mathsf{Ann}\,D, \lambda d.\mathsf{fst}\,(\mathsf{fwd}_t\,\mathsf{nil}\,s\,d), i)$ is a finite fixed-point iteration problem.

15.4 Case Study: Reachability Analysis

An example for a forward analysis is reachability analysis. The domain of the reachability analysis is simply the lattice of booleans, where true signifies that a program

point is reachable.

The transformer takes a program s and a boolean b. The transformer t_{rch} inspects the top-level statement and must produce analysis information for the next statement:

$$t_{\text{rch}}\,\zeta\,s\,b \;=\; \begin{cases} (\beta([\![e]\!]\emptyset) \neq \mathbf{false} \wedge b, \beta([\![e]\!]\emptyset) \neq \mathbf{true} \wedge b) & \text{if } s \text{ is a conditional} \\ b & \text{otherwise} \end{cases}$$

For conditionals, the transformer evaluates the value of the condition in an empty variable environment. Recall that the return type of the transformer depends on the top-level statement, hence it is acceptable that t_{rch} returns a pair for conditionals. The pair expresses that the consequence is potentially reachable if the condition does not evaluate to **false** and b is true, which means that, s was reachable. Similarily, the alternative is potentially reachable if the condition does not evaluate to **true** and b is true. In all other cases the next statement is reachable if and only if s was reachable.

Definition 15.14 ▪ Reachability Analysis
We define $\mathsf{rch}\,\zeta\,s\,d := \mathsf{fst}\,(\mathsf{fwd}\,t_{\text{rch}}\,\zeta\,s\,d)$.

Lemma 15.15 Let s be a program. Then

$$(\mathsf{Ann}\,\mathbb{B},\ \mathsf{rch}\,\varnothing\,s,\ \mathsf{setTopAnn}\,(\mathsf{initAnn}\,s\,\mathbf{false})\,\mathbf{true})$$

is a finite fixed-point iteration problem.

Proof. From Theorem 15.13 with monotonicity of t_{rch}. ∎

Remark 15.16 The function t_{rch} is a good example for a naturally arising transformer that is monotone, but not expansive.

15.4.1 Soundness

We show that soundness of the reachability analysis follows from the fixed-point property by induction on the program structure.

Theorem 15.17 ▪ Soundness
Let $d : \mathsf{Ann}\,\mathbb{B}$ be an annotation for s and let ζ be a parameter context defining all free labels in s. If $\mathsf{fst}\,(\mathsf{fwd}\,t_{\text{rch}}\,\zeta\,s\,\mathbf{r}) \equiv d$ and furthermore $\mathsf{snd}\,(\mathsf{fwd}\,t_{\text{rch}}\,\zeta\,s\,\mathbf{r}) \sqsubseteq \Lambda$ then $\Lambda \vdash \mathbf{reach}_{\{S\}}\,\{\mathbf{r}\}\,s$.

Proof. By induction on s and inversion on $\mathsf{fst}\,(\mathsf{fwd}\,t_{\text{rch}}\,\zeta\,s\,\mathbf{r}) \equiv \mathbf{r}$ with properties of the join operation. ∎

Note that the soundness proof (Theorem 15.17) only needs the information that a fixed-point is reached, but not that it is the smallest. This is the usual behavior for a static analysis: All fixed-points are sound, the smallest fixed-point is just the one with the most precise information. In the case of reachability, the analysis information at the smallest fixed-point claims the least number of program points reachable.

15.4.2 Relative Completeness

We prove that the smallest fixed-point of the analysis is complete in the sense of the correctness predicate for reachability from §11. While soundness of the analysis required only the fixed-point property, we must establish completeness by showing that the completeness property is an invariant throughout the fixed-point iteration.

The first important observation is that the initial value of the reachablility analysis is complete:

Lemma 15.18 Let $d : \mathsf{Ann}\,\mathbb{B}$ annotate s and let ζ be a parameter context defining all free labels in s. Then $\Lambda \vdash \mathbf{reach}_{\{\mathrm{C}\}}\,\{\mathsf{setTopAnn}\,(\mathsf{initAnn}\,s\,\mathbf{false})\,\mathbf{true}\}\,s$.

Lemma 15.19 below shows that reachability analysis only claims functions live if they are called in the subterm. This is an essential requirement for completeness.

Lemma 15.19 Let $\Lambda \vdash \mathbf{reach}_{\{\mathrm{C}\}}\,\{\mathbf{r}\}\,s$ and $\bar{b} = \mathsf{snd}\,(\mathsf{fwd}\,t_{\mathrm{rch}}\,\zeta\,s\,(\mathsf{setTopAnn}\,d\,a))$. If $d \sqsubseteq a$ and $b_i = \mathbf{true}$ then $\mathsf{isCalled}\,s\,f_i$.

Lemma 15.20 below establishes completeness as an invariant of the program transformer fwd.

Lemma 15.20 Let $\mathbf{r}' = \mathsf{fst}\,(\mathsf{fwd}\,t_{\mathrm{rch}}\,\zeta\,s\,(\mathsf{setTopAnn}\,\mathbf{r}\,b))$ and let $b, b' : \mathbb{B}$ such that $b \sqsubseteq b'$. If it holds that $\Lambda \vdash \mathbf{reach}_{\{\mathrm{C}\}}\,\{\mathbf{r}\}\,s$ and we have $\mathbf{r} \sqsubseteq \mathsf{setTopAnn}\,\mathbf{r}'\,b'$ then $\Lambda \vdash \mathbf{reach}_{\{\mathrm{C}\}}\,\{\mathsf{setTopAnn}\,\mathbf{r}'\,b'\}\,s$.

Proof. By induction on $\Lambda \vdash \mathbf{reach}_{\{\mathrm{C}\}}\,\{\mathbf{r}\}\,s$ with Lemma 15.19. The generalization to $\mathsf{setTopAnn}\,\mathbf{r}'\,b'$ in the conclusion is critical to handle the recursion in the function definition case. ∎

Theorem 15.21 ▪ **Relative Completeness**
Let $d : \mathsf{Ann}\,\mathbb{B}$ annotate s and let ζ be a parameter context defining all free labels in s. Then

$$\Lambda \vdash \mathbf{reach}_{\{\mathrm{C}\}}\,\{\mathsf{fix}(\mathsf{rch}\,\varnothing\,s)\,(\mathsf{setTopAnn}\,(\mathsf{initAnn}\,s\,\mathbf{false})\,\mathbf{true})\}\,s$$

Proof. By chain induction (Lemma 15.10). The base case is Lemma 15.18. The inductive step is Lemma 15.20. ∎

$$
\begin{aligned}
&\mathsf{bwd}\,\zeta\,\Delta\,(\texttt{let}\,x = \eta\,\texttt{in}\,s)\,(d \cdot a)\;=\\
&\quad \mathsf{let}\;a' = \mathsf{bwd}\,\zeta\,\Delta\,s\,a\;\mathsf{in}\\
&\quad \mathsf{let}\;d' = t\,\zeta\,\Delta\,(\texttt{let}\,x = \eta\,\texttt{in}\,s)\,[a']\;\mathsf{in}\\
&\quad (d' \cdot a', A)\\
&\mathsf{bwd}\,\zeta\,\Delta\,(\texttt{if}\,e\,\texttt{then}\,s_1\,\texttt{else}\,s_2)\,(d \cdot a_1, a_2)\;=\\
&\quad \mathsf{let}\;a_1' = \mathsf{bwd}\,\zeta\,\Delta\,s_1\,a_1\;\mathsf{in}\\
&\quad \mathsf{let}\;a_2' = \mathsf{bwd}\,\zeta\,\Delta\,s_2\,a_2\;\mathsf{in}\\
&\quad \mathsf{let}\;\mathsf{d} = t\,\zeta\,\Delta\,(\texttt{if}\,e\,\texttt{then}\,s_1\,\texttt{else}\,s_2)\,[a_1']\,[a_2']\;\mathsf{in}\\
&\quad (d \cdot a_1', a_2')\\
&\mathsf{bwd}\,\zeta\,\Delta\,(f\,\overline{e})\,d\;=\;t\,\zeta\,\Delta\,(f\,\overline{e})\,d\\
&\mathsf{bwd}\,\zeta\,\Delta\,e\,d\;=\;t\,\zeta\,\Delta\,e\,d\\
&\mathsf{bwd}\,\zeta\,\Delta\,(\texttt{fun}\,\overline{f\,\overline{x}\;=\;s}\,\texttt{in}\,t)\,(d \cdot \overline{a}, b)\;=\\
&\quad \mathsf{let}\;\zeta' = \overline{\overline{x}}; \zeta\;\mathsf{in}\\
&\quad \mathsf{let}\;\Delta' = [\overline{a}]; \Delta\;\mathsf{in}\\
&\quad \mathsf{let}\;\overline{a'} = \mathsf{bwdF}\,\zeta'\,\Delta'\,\overline{s}\;\mathsf{in}\\
&\quad \mathsf{let}\;\Delta'' = [\overline{a'}]; \Delta\;\mathsf{in}\\
&\quad \mathsf{let}\;b' = \mathsf{bwd}\,\zeta'\,\Delta''\,t\,b\;\mathsf{in}\\
&\quad (d' \cdot \overline{a'}, b')\\
&\mathsf{bwdF}\,\zeta\,\Delta\,\overline{s}\;=\overline{a'}\;\text{such that}\\
&\quad \forall i, a_i' = \mathsf{bwd}\,\zeta\,\Delta\,s_i\,a_i
\end{aligned}
$$

Figure 15.2: Definition of the backward analysis framework.

15.5 A Framework for Backward Data Flow Analyses on IL

As previously in the forward analysis framework, the result of a backwards dataflow analysis is an annotation of the program that associates with every program point a value of some analysis domain, which we require to be a lower-bounded join-semi-lattice. The result is obtained by finite fixed-point analysis of a program transformer.

The backward analysis function bwd maintains a parameter context ζ that holds the parameters of each defined function, and an analysis domain context Δ that maps every function to a value from the analysis domain. Additionally, bwd takes a program and a corresponding annotation as arguments, and produces a new annotation as

result. The function bwd first descends recursively through the program, and then propagates the value of the analysis domain D in a backward fashion upwards in the opposite direction of execution. In IL, the direction of execution corresponds to a recursive traversal of the program structure, so bwd can simply return the current analysis value. We hence implement a backward analysis program transformer as a recursive traversal of the program and its annotation. During traversal, bwd records new analysis information to produce the new annotation. The information at applications is taken from the parameter Δ. At function definitions, Δ is extended with values for the newly defined functions taken out of the annotations for the corresponding function bodies. As in the forward analysis, the only communication between program-level iterations happens at function definitions: All other annotations are overwritten with the new values (not joined).

The function bwd is designed to be reused for backward analyses in general. For this porpose, the local transformation is handled by a parameter function t that transforms a value of type D according to the top-most construction in a given IL program. The analysis provides the transformer with the current contexts ζ and Δ that holds the parameters and the current analysis value, respectively, for each defined function. Additionally, the transformer gets the current term s and the analysis value that corresponds to the program "after" the top-level statement in s, and must in turn produce the locally transformed analysis value for the program point before s. For example, bwd might evaluate $t\,\zeta\,\Delta\,(\texttt{let}\,x = \eta\,\texttt{in}\,s)\,d$, where d is the analysis value at s, to obtain the analysis value for $\texttt{let}\,x = \eta\,\texttt{in}\,s$. In the Coq development we made the type of t depend on the program to allow t to take two arguments in case of a conditional, that is one for the consequence and one for the alternative, to improve analysis precision. In the presentation of bwd in Figure 15.1 we omit types entirely, and give an essentially untyped version of the analysis function. The untyped version communicates the computational part clearly, and hides the intricate dependent type structure required for book-keeping purposes.

15.5.1 Backward Analysis as Finite Iteration Problem

A backward dataflow analysis can be cast as a finite fixed-point iteration problem. The first ingredient is the fact that bwd is monotone.

Definition 15.22 We say a transformer t is monotone if $d \sqsubseteq d' \to t\,\zeta, \Delta\,s\,d \sqsubseteq t\,\zeta\,\Delta\,s\,d'$.

Lemma 15.23 If the transformer t is monotone, then bwd is monotone in the sense that $d \sqsubseteq d' \to \Delta \sqsubseteq \Delta' \to \mathsf{bwd}\,\zeta\Delta'\,s\,d \sqsubseteq \mathsf{bwd}\,\zeta\,\Delta'\,s\,d'$.

As already discussed in the context of the forward analysis in §15.3.1, we use dependent types for bwd to be able to use finite-height lattices.

Theorem 15.24 Let D be a lower-bounded join-semi-lattice, and s be a program and $i : D$ such that $i \sqsubseteq \mathsf{bwd}\,\varnothing\,\varnothing\,i$, and let t be monotone in the sense of Definition 15.22. Then $(\mathsf{Ann}\,D, \mathsf{bwd}_t\,\varnothing\,\varnothing\,s, i)$ is a finite fixed-point iteration problem.

15.6 Case Study: Liveness Analysis

The domain of a liveness analysis of a closed program s is the type of subsets of the set of variables occuring in s, written $\{x : \mathsf{set}\,\mathcal{V} \mid x \subseteq \mathcal{V}_{\mathcal{O}}\,s\}$. It is clear that $\sqsubseteq := \subseteq$ on $\{x : \mathsf{set}\,\mathcal{V} \mid x \subseteq \mathcal{V}_{\mathcal{O}}\,s\}$ is terminating for any s. Liveness analysis relies on the dependent type structure of fwd to maintain this type. The arising types are involved and we omit them in this paper presentation.

15.6.1 Local Transformer and Program Transformer

The local transformer for liveness analysis is defined as follows:

$$
\begin{aligned}
t_{\text{live}}\,\zeta\,\Delta\,(\texttt{let}\,x = e\,\texttt{in}\,s)\,d &= d \setminus \{x\} \cup \mathsf{if}\ x \in d\ \mathsf{then}\ \mathsf{fv}\,(e)\ \mathsf{else}\ \emptyset \\
t_{\text{live}}\,\zeta\,\Delta\,(\texttt{let}\,x = f\,\overline{e}\,\texttt{in}\,s)\,d &= d \setminus \{x\} \cup \mathsf{fv}\,(\overline{e}) \\
t_{\text{live}}\,\zeta\,\Delta\,(\texttt{if}\,e\,\texttt{then}\,s_1\,\texttt{else}\,s_2)\,(d_1, d_2) &= \begin{cases} d_1 & \text{if } \beta([\![e]\!]\emptyset) = \mathbf{true} \\ d_2 & \text{if } \beta([\![e]\!]\emptyset) = \mathbf{false} \\ d_1 \cup d_2 & \text{otherwise} \end{cases} \\
t_{\text{live}}\,\zeta\,\Delta\,(f\,\overline{e})\,d &= \bigcup \{\mathsf{fv}\,(e_i) \mid (\zeta_f)_i \in \Delta_f\} \\
t_{\text{live}}\,\zeta\,\Delta\,e\,d &= \mathsf{fv}\,(e) \\
t_{\text{live}}\,\zeta\,\Delta\,(\texttt{fun}\,\overline{f\,\overline{x} = s}\,\texttt{in}\,t)\,d &= d
\end{aligned}
$$

The local transformer gets the liveness information for the successor statements as arguments, and has to return the liveness information before the statement. For let-statements, x is removed from the live set and the free variables of the expression e are only added if x is live in the continuation. The situation is similar for external events, with the notable difference that free variables of arguments are always considered live. For conditionals, we check whether the condition evaluates statically (i.e. is a constant expression), and then return the live information from the branch. Otherwise, both branches are assumed reachable and the transformer returns the union of the two live sets. For an application, the transformer puts the free variables of those argument expressions in the live set, if the corresponding parameter is live. At return statements, the live set is the set of free variables of the expression. At function definitions, the the transformer just propagates the live set from the continuation.

Lemma 15.25 t_{live} is monotone.

Definition 15.26 ▪ **Liveness Analysis**
We define $\mathsf{live}\,\zeta\,\Delta\,s\,d := \mathsf{bwd}\,t_{\mathrm{live}}\,\zeta\,\Delta\,s\,d$.

15.6.2 Liveness Analysis as Finite Fixed-Point Iteration Problem

Lemma 15.27 Let s be a program. Then

$$(\mathsf{Ann}\,\{x : \mathsf{set}\,\mathcal{V} \mid x \subseteq \mathcal{V}_{\mathcal{O}}\,s\}, \mathsf{live}\,\varnothing\,\varnothing\,s, \mathsf{initAnn}\,s\,\emptyset)$$

is a finite fixed-point iteration problem.

Proof. From Theorem 15.24 with monotonicity of t_{live}. ■

15.6.3 Soundness

Theorem 15.28 Let s be a program and $d : \mathsf{Ann}\,\{x : \mathsf{set}\,\mathcal{V} \mid x \subseteq \mathcal{V}_{\mathcal{O}}\,s\}$ be an annotation for s. Let ζ be a parameter s-context and ζ be an analysis domain s-context. If $\mathsf{live}\,\zeta\,\Delta\,s\,d \equiv d$ then $\zeta \mid \Delta \vdash \mathbf{tlive_I}\,s : d$.

Proof. Induction on s. ■

15.7 A Framwork for SSA-based Forward Analyses on IL

The third and last framework we developed is a framework supporting SSA-based forward analyses. The main point of SSA-based program analyses is to keep the asymptotic size of analysis information linear in the program size n. The key idea is that the analysis domain should be a mapping from SSA variables to some per-variable domain. Note that this approach saves a factor of n in analysis size. Liveness analysis, which we discussed earlier, associates with every program point a element from the analysis domain, which results in the size of analysis information growing with n^2.

Our framework for SSA-based program analyses requires the analysis domain to be a mapping $\mathcal{V} \to D$, where D can be any lower-bounded join-semi-lattice. Additionally, the framework always performs a reachability analysis, and provides the results to the analysis. Reachability information has constant space requirements per program point (one boolean), and the overall analysis information size remains linear in the program size.

The SSA-based framework is parameterized by two local transformers. One transforms the variable domain of type $\mathcal{V} \to D$, which we call the local value transformer, and the other transforms reachability information at conditionals and is called the

$$
\begin{aligned}
&\mathsf{fwdSSA}\,\zeta\,(\texttt{let}\;x = \eta\;\texttt{in}\;s)\,d\,(b \cdot a) \;=\\
&\quad \mathsf{let}\;d' = d[x \mapsto t\,b\,d\,\eta]\;\mathsf{in}\\
&\quad \mathsf{let}\;d'', a', A = \mathsf{fwdSSA}\,\zeta\,s\,d'\,(\mathsf{setTopAnn}\,a\,b)\;\mathsf{in}\\
&\quad (d'', b \cdot a', A)\\
&\mathsf{fwdSSA}\,\zeta\,(\texttt{if}\;e\;\texttt{then}\;s_1\;\texttt{else}\;s_2)\,d\,(d \cdot a_1, a_2) \;=\\
&\quad \mathsf{let}\;b_1', b_2' = r\,b\,d\,e\;\mathsf{in}\\
&\quad \mathsf{let}\;d', A_1 = \mathsf{fwdSSA}\,\zeta\,s_1\,d\,(\mathsf{setTopAnn}\,a_1\,b_1')\;\mathsf{in}\\
&\quad \mathsf{let}\;d'', A_2 = \mathsf{fwdSSA}\,\zeta\,s_2\,d'\,(\mathsf{setTopAnn}\,a_2\,b_2')\;\mathsf{in}\\
&\quad (d'', b \cdot a_1', a_2', A_1 \sqcup A_2)\\
&\mathsf{fwdSSA}\,\zeta\,(f\,\overline{e})\,d\,b \;=\\
&\quad \mathsf{let}\;d' = d[\zeta_f \mapsto t\,b\,d\,\overline{e}]\;\mathsf{in}\\
&\quad (\mathsf{if}\;b\;\mathsf{then}\;d'\;\mathsf{else}\;d, b, \bot[f \mapsto b])\\
&\mathsf{fwdSSA}\,\zeta\,e\,d \;=\; (d, b, \bot)\\
&\mathsf{fwdSSA}\,\zeta\,(\texttt{fun}\;\overline{f\,\overline{x} = s}\;\texttt{in}\;t)\,d\,(b \cdot \overline{a}, c) \;=\\
&\quad \mathsf{let}\;\zeta' = \overline{\overline{x}}; \zeta'\;\mathsf{in}\\
&\quad \mathsf{let}\;d', c', B = \mathsf{fwdSSA}\,\zeta'\,t\,d\,(\mathsf{setTopAnn}\,c\,b)\;\mathsf{in}\\
&\quad \mathsf{let}\;d'', \overline{a'}, \overline{A} = \mathsf{fwdSSAF}\,\zeta'\,d'\,\overline{s}\,(\mathsf{joinTopAnn}\,\overline{a}\,B)\;\;\mathsf{in}\\
&\quad (d'', b \cdot \mathsf{setTopAnn}\,\overline{a'}\,A, c', \mathsf{drop}\,|\overline{f}|\,\overline{A})\\
&\mathsf{fwdSSAF}\,\zeta\,A\,d\,(s_1, \ldots, s_n)\,(a_1, \ldots, a_n) \;=\\
&\quad \mathsf{let}\;d', a_1', B = \mathsf{fwdSSA}\,\zeta'\,s_1\,d\,a_1\;\mathsf{in}\\
&\quad \mathsf{let}\;d'', (a_2', \ldots, a_n'), C = \mathsf{fwdSSAF}\,\zeta'\,A\,d'\,(s_2, \ldots, s_n)\,(a_2, \ldots, a_n)\;\mathsf{in}\\
&\quad (d'', (a_1', \ldots, a_n'), B \sqcup C)\\
&\mathsf{fwdSSAF}\,\zeta\,A\,d\,\varnothing\,\varnothing \;= (d, \varnothing, A)
\end{aligned}
$$

Figure 15.3: Definition of the SSA-based forward analysis framework.

local reachability transformer. In this way, value information and reachability information can be combined to make the analysis more precise. In the next section, we evaluate the framework with the analysis for sparse conditional constant propagation.

The key property of fwdSSA is that fwdSSA only changes certain variables in the analysis mapping. This reflects the SSA property.

Lemma 15.29 Let $d', \_, \_ = \mathsf{fwdSSA}\,\zeta\,s\,d\,a$. If x is a variable not defined in s then $d\,x \sqsubseteq d'\,x$. If x is not defined in s and not in ζ, then also $d'\,x \sqsubseteq d\,x$.

15.7.1 SSA-based Forward Analysis as Finite Iteration Problem

In this section, we want to show that we can cast the forward analysis problem as a finite fixed-point iteration problem. The first ingredient is the fact that fwdSSA is monotone.

Definition 15.30 We say a transformer t is monotone if $b \sqsubseteq b' \to d \sqsubseteq d' \to t\,b\,d\,\eta \sqsubseteq t\,b'\,d'\,\eta$. Likewise, we say the transformer r is monotone if $b \sqsubseteq b' \to d \sqsubseteq d' \to r\,b\,d\,e \sqsubseteq t\,b'\,d'\,e$.

Lemma 15.31 If the transformers t and r are monotone, then fwdSSA is monotone in the sense that $d \sqsubseteq d' \to a \sqsubseteq a' \to \mathsf{fwdSSA}\,\zeta\,s\,d\,a \sqsubseteq \mathsf{fwdSSA}\,\zeta\,s\,d'\,a'$.

Theorem 15.32 Let s be a program. Let D be a upper-bounded and lower-bounded join-semi-lattice with finite height. Let t be a monotone local value transformer, and r be a monotone local reachability transfomer. Then

$$(\mathsf{Ann}\,D, \lambda(d,a).\mathsf{fst}\,(\mathsf{fwdSSA}_{t,r}\,\mathsf{nil}\,\mathsf{nil}\,s\,d\,a), \bot[\mathsf{fv}\,(s) \mapsto \overline{\top}])$$

is a finite fixed-point iteration problem.

15.8 Case Study: SCCP Analysis

The analysis for sparse conditional constant propagation (SCCP) is defined in the framework for SSA-based forward analyses. The value transformer for constant propagation t look as follows:

$$\begin{aligned} t\,b\,d\,e &= \mathsf{ceval}_d\,e \\ t\,b\,d\,(\alpha(\overline{e})) &= \top \end{aligned}$$

For expressions e, we use the evaluation operation ceval for abstract values from SCCP (§14.3.1). If the expression belongs to an external event, the transformer knows nothing about the result value and returns $\bot$.

The reachability transformer takes the constant propagation information into account. It evaluates the condition e abstractly, and then deems the consequence reachable if $\mathbf{true} \sqsubseteq \mathsf{ceval}_d\,e$, and the alternative if $\mathbf{false} \sqsubseteq \mathsf{ceval}_d\,e$.

$$r\,b\,d\,e = (\mathbf{true} \sqsubseteq \mathsf{ceval}_d\,e, \mathbf{false} \sqsubseteq \mathsf{ceval}_d\,e)$$

Theorem 15.33 Let s be renamed apart and let ζ be a parameter context for s, and let ζ be disjoint from the variables defined in s, and let each parameter list in ζ be unique. If $\mathsf{fst}\,(\mathsf{fwdSSA}_{t,r}\,\zeta\,s\,d\,a) \equiv (d, a)$ then $Z \mid (\kappa\,Z) \vdash \mathbf{const}_\kappa\ s$.

16 Assembly

The translation to assembly is in our case the translation to CompCert's "Linear" intermediate language. The translation is straight-forward and we describe it in this section. The main point is to show that it is indeed realistic to compile with IL, since after register allocation and parallel move elimination, we are low-level enough to compile into one of the last translation phases of CompCert. Verifying correctness of the translation is also straight-forward, as was to be expected because the two languages closely correspond as can be see in Figure 2.15.

The integration of the correctness proofs of LVC and CompCert, however, poses difficulties, because CompCert's equivalence is not compositional, which will be addressed by an upcoming paper []. In particular, we are missing the integration of our correctness proof, which relates an IL/I program to the function body of a C function in CompCert's linear, into the end-to-end correctness statement from CompCert. That means we can prove that the C function behaves like the IL program it is generated from, but we have no statement about the whole translation unit.

16.1 Properties of the Translation

Verifying the correctness of the translation to CompCert poses several difficulties, and we discuss the key problems here in detail.

16.1.1 System Calls

The first problem is that while our system calls are there mainly as a benchmark for our semantic methods, CompCert implements actual system calls which require calling conventions. At the moment, our register allocation phase does not support register constrains which are required to correctly implement the calling convention. For this reason, we do not verify system calls in the translation, but require that the IL/I program contains no system calls. While this is a severe restriction in practice, we do not expect any principal problems in extending our approach to system calls.

16.1.2 Functions vs Translation Units

Another problem is related to CompCert's correctness properties. At the moment, LVC only supports compilation of one C-level function (that is, LVC can only translate what corresponds to a C function body, but not a whole translation unit, as CompCert does it). Currently, we integrate the result of LVC as one function into one of CompCert's C translation units. To even specify what correctness should mean, we would have to be able to specify the behavior of a CompCert translation unit where one C-level function is given in terms of LVC's input program. It is our understanding that the current setup in CompCert does not easily allow this. If we had such a specification, we could then show that LVC produces a valid implementation.

We instead prove a different theorem that shows that the translation step to Linear is correct in the sense that the body of the generated C function implements the IL program, but do not bother with the integration part. We show that the Linear code we generate is bisimilar to the IL/I program. To be able to show this, we modify CompCert's Linear semantics in such a way that function return terminates the program with the result value corresponding to the value in the return register. We expect that this result could be easily integrated in a correctness proof once compositional equivalence methods for CompCert are available.

16.2 Machine Registers and Stack Slots

We define a set $\mathcal{V}_{\text{PPC}} \subseteq \mathcal{V}_R$ that contains those registers from the register partition that are to be mapped to machine registers. An invariant from the register allocation phase will provide that the register allocated IL/I program only uses registers in $\mathcal{V}_{\text{PPC}}$. Currently, LVC only use the caller-saved PowerPC registers $r_3, \ldots, r_{12}$. We arrange things such that $\mathcal{V}_{\text{PPC}}$ contains the $10 = |\{r_3, \ldots, r_{12}\}|$ smallest variables $\{x_0, \ldots, x_9\}$ in $\mathcal{V}_R$, and use x_i to stand for machine register r_{i+3}.

16.2.1 Memory Relation

After register allocation, all variables in IL/I are either registers from $\mathcal{V}_{\text{PPC}}$ or stack slots from $\mathcal{V}_M$, and a configurations of the IL/I semantics records the values of both types of variables in the variable environment V. A configuration of CompCert's Linear language (which we will introduce shortly) has a designated component `rs`, which records both the contents of registers and the contents of spill slots. While IL/I currently only deals with integer variables, `rs` can store other types of values, too. Integer variables in `rs` are tagged with a type, and the tag for integer values is `Vint`. Since in both IL/I and CompCert's Linear registers and stack slots are handled as different kinds of variables, it is easy to define a function toLinVar such

$$\frac{\mathsf{State}\ \mathtt{sf}\ \mathtt{fn}\ \mathtt{sp}\ \mathtt{code}\ \mathtt{rs}\ \mathtt{mem} \longrightarrow \sigma'}{\mathsf{State}\ \mathtt{sf}\ \mathtt{fn}\ \mathtt{sp}\ \mathtt{code}\ \mathtt{rs}\ \mathtt{mem} \longrightarrow_A \sigma'}$$

Figure 16.1: Rules that adapt Linear's small step semantics for use in our IDRS. The point is that the adapted relation $\longrightarrow_A$ only allows in-function transitions. Any CallState is stuck, any Returnstate is a final state.

that $\mathsf{toLinVar}\, x$ yields the machine register corresponding to x if $x \in \mathcal{V}_{\mathsf{PPC}}$, and Linear's designation for the spill slot corresponding to x if $x \in \mathcal{V}_M$.

Definition 16.1 We say an IL/I variable environment V and a register state `rs` from CompCert's Linear are in relation, written $\mathsf{mRel}\, V\ \mathtt{rs}$ if whenever $v \in \mathbb{V}$ and $x \in \mathcal{V}_{\mathsf{PPC}} \cup \mathcal{V}_M$ and $Vx = v$, then $\mathtt{rs}\ (\mathsf{toLinVar}\, x) = \mathtt{Vint}\ v$.

16.3 Linear's Semantic as IDRS

To be able to reason about CompCert's Linear, we define an IDRS based on Linear's semantics. Let us first examine the Linear's states. There are three types of states: Regular states that are used for executions within a C-level function. Call states that are used to indicate that a function call is about to be executed, and return states that indicate that a function return is about to be executed. A regular state has the form

$$\mathsf{State}\ \mathtt{sf}\ \mathtt{fn}\ \mathtt{sp}\ \mathtt{code}\ \mathtt{rs}\ \mathtt{mem}$$

where `sf` is the list of stack frames, `fn` is the current function, `sp` is the stack pointer, `code` is the Linear code currently executing, `rs` maps registers and stack slots to values, and `mem` is the memory. A return state has the form

$$\mathsf{Returnstate}\ \mathtt{sf}\ \mathtt{rs}\ \mathtt{mem}$$

where `sf` is the list of stack frames, `rs` maps registers and stack slots to values, and `mem` is the memory.

We first define an adapter relation that restricts Linear's semantics to stay within one function invocation. In particular, the adapter relation ensures any Returnstate of CompCert's Linear is a final state of the IDRS with the result value corresponding to the value in r_3, the return register for PowerPC.

The IDRS is obtained by using the adapter relation from Figure 16.1 with CompCert's step relation for Linear, and defining the return function

$$\mathsf{res}\ (\mathsf{Returnstate}\ \mathtt{sf}\ \mathtt{rs}\ \mathtt{mem}) = v$$

where v is the value of the return register r_3 according to `rs`.

Assignment	Instruction
$\texttt{let}\, x = y\, \texttt{in} \ldots$	`mov y x`
$\texttt{let}\, x = Y\, \texttt{in} \ldots$	`getstack Y x`
$\texttt{let}\, X = y\, \texttt{in} \ldots$	`setstack X y`

Figure 16.2: Translation table for variable-to-variable assignments. The register r must be unused. Note that these are Linear's assembly instructions that are later translated to PowerPC assembly instructions by CompCert.

16.4 Translating Let-Bindings

We require all let-bindings to fall into one of the following two classes, which is guaranteed by the spilling phase.

1. The right-hand side is a variable, and at least one of the variables involved is a register variable.
2. The left hand side is a register variable, and the right-hand side is a unary or binary expression which only involves register variables and for which a corresponding instruction exists.

The function toLinExp handles the translation of let-bindings. The table Figure 16.2 shows let-bindings in category 1 are translated by toLinExp depending on whether the left and right-hand sides are registers or slots. toLinExp translates let-bindings from category 2 into the corresponding Linear instruction.

Lemma 16.2 ■ Correctness of toLinExp
Let $\mathsf{mRel}\, V\, \texttt{rs}$ and $\mathsf{fv}\, e \subseteq \mathcal{V}_{\mathsf{PPC}}$ and $x \in \mathcal{V}_{\mathsf{PPC}}$ and

$$\forall\, V\, \texttt{rs}, \mathsf{mRel}\, V\, \texttt{rs} \to (L, V, s)(\approx_r^p \cup\, r)(\mathsf{State}\, \texttt{sf fn sp code rs mem})$$

then

$$(L, V, \texttt{let}\, x = e\, \texttt{in}\, s) \approx_r^p (\mathsf{State}\, \texttt{sf fn sp}\, (\mathsf{toLinExp}\, x\, e; \texttt{code})\, \texttt{rs mem}).$$

16.5 Translating Conditions of Conditionals

We require conditions to either be a single variable, a less than comparison, or a comparison for equality. Translation is handled by toLinCond as defined in Figure 16.3.

$$\begin{aligned}
\mathsf{toLinCond}\, l\, (y) &= \texttt{beq y 0 l} \\
\mathsf{toLinCond}\, l\, (x < y) &= \texttt{ble y x l} \\
\mathsf{toLinCond}\, l\, (x = y) &= \texttt{bne x y l}
\end{aligned}$$

Figure 16.3: Definition of the translation function toLinCond for conditions. l denotes the label of the alternative branch of the conditional. Note that the translation negates the condition because we want the code to branch to the alternative if the condition is false. Note also that these are Linear's assembly instructions that are later translated to PowerPC assembly instructions by CompCert.

$$\begin{aligned}
\mathsf{toLin}\, D\, (\texttt{let}\, x = e\, \texttt{in}\, s) &= \mathsf{toLinExp}\, x\, e; \mathsf{toLin}\, D\, s \\
\mathsf{toLin}\, D\, (\texttt{if}\, e\, \texttt{then}\, s\, \texttt{else}\, t) &= \mathsf{toLinCond}\, l\, e; \mathsf{toLin}\, D\, s; l : \mathsf{toLin}\, D\, t \quad l \text{ fresh} \\
\mathsf{toLin}\, D\, (f\, ()) &= \texttt{goto}\, D_f \\
\mathsf{toLin}\, D\, (e) &= \mathsf{toLinExp}\, \texttt{r3}\, e; \texttt{return} \\
\mathsf{toLin}\, D\, (\texttt{fun}\, \overline{f\, () = s}\, \texttt{in}\, t) &= \mathsf{let}\, D' = \bar{l}; D\, \mathsf{in} \\
&\quad\; l' : \mathsf{toLin}\, D\, t; \overline{l : \mathsf{toLin}\, D'\, s} \qquad \bar{l}, l' \text{ fresh}
\end{aligned}$$

Figure 16.4: Translation from IL/I without parameters to CompCert's Linear

Lemma 16.3 ■ Correctness of toLinCond
Let e be a simple condition, $\mathsf{mRel}\, V\, \texttt{rs}$ and $\mathsf{fv}\, e \subseteq \mathcal{V}_{\mathsf{PPC}}$ and

$$(L, V, s_1)(\approx_r^p \cup\, r)(\mathsf{State}\, \texttt{sf fn sp}\, (c_1; (l : c_2); c_3)\, \texttt{rs mem})$$

and

$$(L, V, s_2)(\approx_r^p \cup\, r)(\mathsf{State}\, \texttt{sf fn sp}\, c_2; c_3\, \texttt{rs mem})$$

and $\mathsf{find\_label}\, l\, (\mathsf{fn\_code}\, \texttt{fn}) = c_2; c_3$ then

$$\begin{aligned}
&(L, V, \texttt{if}\, e\, \texttt{then}\, s_1\, \texttt{else}\, s_2) \\
\approx_r^p\ &(\mathsf{State}\, \texttt{sf fn sp}\, (\mathsf{toLinCond}\, l\, e; c_1; (l : c_2); c_3)\, \texttt{rs mem}).
\end{aligned}$$

16.6 Translating IL/I to CompCert's Linear

The translation procedure for IL/I to Linear is given in Figure 16.4. The translation of let bindings is handled by toLinExp. A conditional is translated by first generating

instructions that do the branching, and then generating the code for the consequence followed by the code for the alternative. We need a new label l at the beginning of the code for the alternative. The code for function application is a simple goto instruction; the label is looked up in the environment D. The IL/I return statement evaluates the return expression, stores the result in the return register (here $r3$ because according to PowerPC ABI for integer values). The code for a function definition $\texttt{fun}\ \overline{f\,() = s}\ \texttt{in}\ t$ is generated by first generating code for the continuation t, followed by the code for the function bodies, in order, and each prepended with a fresh label. The function labels are then recorded in the environment D, where they can be looked up.

We generate labels in toLin in a strictly increasing fashion, and this is important for the formulation of the invariant for the correctness proof. However, for the sake of simplicity, we state the correctness result only for programs without free function variables. In the Coq development, we prove a generalized version for open programs, which requires more invariants.

Lemma 16.4 ▪ Correctness of toLin
Let $\mathsf{mRel}\, V\ \texttt{rs}$ and $\mathsf{fn\_code}\ \texttt{fn} = c$ and s is linearizable and s has no parameters and s satisfies the register bound and two technical requirements for the function return hold then

$$(L, V, \texttt{if}\, e\, \texttt{then}\, s_1\, \texttt{else}\, s_2) \gtrsim_r (\mathsf{State}\ \texttt{sf}\ \texttt{fn}\ \texttt{sp}\ (\mathsf{toLin}\ \varnothing\ s)\ \texttt{rs}\ \texttt{mem}).$$

17 Coq Development

This section gives an overview over the components of the LVC development, and discusses issues of the implementation of LVC in Coq. We provide some numbers that help quantify the size of the project.

17.1 Lines of Code

The Coq development of LVC consists of 58,344 lines of code (LoC). The Paco libary (3,591 LoC) and the containers library (19,799 LoC), which LVC uses, are not included in this total. The extracted OCaml source code of LVC is between 20-30k LoC, but includes parts extracted from CompCert and the containers library. A detailed breakdown by component is given in Figure 17.1. Note that the basic definitions such as injectivity, agreement of functions, etc. make up a major part (15,156LoC, 26%) of the development and could, possibly, in the future, be provided by the standard library. The semantic definitions are also rather lengthy (15,211LoC, 26%), which is in part due to the insufficient support for coinduction in Coq, and in part because basic definitions such as transitive, reflexive closure and their properties had to be formalized as well. The Coq code of the transformation phases of LVC, which this thesis focuses on, is then rather compact. For example, the proof for coherence (683LoC, 1) is very small. The proof for the whole register assignment phase (assignment and spilling) is relatively compact (9,127LoC, 16%).

17.2 Effort

LVC uses git as source code management system (SCM) since mid 2012. The initial commits contain material from my Master's thesis [Sch13]. Over the course of the development, there were approximately 1500 change sets committed to the git repository. Figure 17.2 provides a rough estimate about the contributors to the LVC effort by showing the number of change sets per author, and the sum of LoC the change sets modify.

Component	Lines of Code		Number of			%
	Spec.	Proofs	Lemmas	Definitions	Tactics	
Preliminaries	Σ 7,549	7,607	1,421	243	254	26
- Sets and Maps	2,958	3,383	634	61	54	11
- Utilities and Tactics	4,591	4,224	787	182	200	15
IL	Σ 6,751	8,460	921	337	80	26
- Semantics	3,231	2,571	438	210	45	10
- Equivalence	1,937	3,097	261	94	19	9
- α-conversion	1,252	2,440	176	25	12	6
- Coherence	331	352	46	8	4	1
Register Allocation	Σ 3,083	6,044	304	65	4	16
- Assignment	509	1,302	64	13	1	3
- Spilling	2,574	4,742	240	52	3	13
Analyses	2,875	3,442	365	100	42	11
SSA Construction	620	1,087	66	20	3	3
Value Optimizations	1,396	1,632	191	71	6	5
DCE	667	1,226	88	14	2	3
Lowering	1,168	2,104	140	68	10	6
OCaml Integration	1099					
Coq Plugin	127					
LVC in total	Σ 25,077	32,620	3,586	960	401	100

Figure 17.1: Overview over distribution of the 58,344 lines of code (LoC) in LVC. The table excludes the Paco libary (3,591 LoC) and the containers library (19,799 LoC) which LVC uses. The table includes framworks for SMT-based translation validation (1,407 LoC) and translation validation with repair (1,823 LoC) which are part of LVC but not described in this thesis.

Contributor	Commits	Lines Changed
Julian Rosemann	120	33,467
Heiko Becker	137	24,045
Maximilian Zöllner	9	1,589
Sigurd Schneider	1,228	642,655

Figure 17.2: Change sets and lines changed per author.

17.3 De-Bruijn Variables

LVC uses De-Bruijn indices for the function alphabet $\mathcal{F}$. This caused much trouble, and, in our setting, probably did not have a lot of benefits. A De-Bruijn approach can work with automation, and if extraction is not a concern [STS15]. However, De-Bruijn's main advantage is the fact that all terms are in a certain α-normal form. This is convenient when using a substitution semantics (which LVC does not). Transformations on De-Bruijn terms, however, always have to re-establish α-normality. We found that this is mainly a complication. For example, unreachable code elimination (§11.4) is essentially about renaming De-Bruijn indices, both on the implementation and on the correctness proof side. In a named setting, UCE would be a very simple transformation. Another draw-back of De-Bruijn is that the indices cannot be used as globally unique identifiers. This makes our implementation of program analyses slower than an implementation with explicit names. Unfortunately, we were too far into LVC's development to move away from the design decision for De-Bruijn indices.

17.4 Mutual Recursion

At some point during the development of LVC, we wanted to investigate the relationship between block-nesting structure in IL, the dominance-order, and the number of parameters required to make a program coherent (see also §12.6). We introduced mutually recursive definitions into LVC to be able to express an "unnested" program, and consider a procedure that re-nests the function in such a way that the number of parameters required to make the program coherent is minimal. However, just adding mutually recursive definitions took almost a year to implement, because it introduced an unexpected amount of complication in every part of LVC. In retrospect, introducing mutually recursive definitions was a grave mistake, especially because of the interaction with De-Bruijn indices. In a named setting, this would probably have been less painful. We never got to investigate what originally had motivated us to introduce mutually recursive definitions, because there was no time left.

17.5 Use of Axioms in Coq

The constructive type theory used in Coq allows to assume additional axioms, such as excluded middle. In the development of LVC we generally tried to avoided using additional axioms. We require two forms forms of serious classical reasoning in the program equivalence meta-theory. In particular, we use excluded middle to obtain Theorem 3.29, which shows that our IDRS either terminate or diverge. Theorem 3.29 is critical to show that prefix trace equivalence coincides with bisimulation.

We further use informed excluded middle and indefinite description to obtain an Theorem 3.30, which is an informative version of Theorem 3.29. We use need this theorem to show that every IDRS configuration produces a co-trace (Lemma 5.8), which is ultimately used to show that infinite trace equivalence coincides with bisimulation.

Our end-to-end correctness theorems do not depend on strong classical axioms. The only assumptions our end-to-end theorems[1] depend on are the following:

- The library Paco uses uniqueness of identity proofs (UIP) in the form of John Major's equality [McB00], on which our results transitively depend.
- Schäfer's and Smolka's approach to coinduction [SS17] depends on functional extensionality. We use their approach to show-case that their approach enables the use of a transitivity property in a simulation proof in Theorem 10.8, and hence transitively depend on functional extensionality.

17.6 A Wish-list for Coq

During the development of LVC, Coq improved tremendously. LVC was first developed on Coq 8.3 and the current version of the development requires Coq 8.7. It is difficult to exaggerate how much Coq improved and matured between these to versions. The Coq development team has done fantastic work. Nevertheless, there are still unsatisfactory aspects of Coq. We discuss issues that pertain to the following general areas:

1. An Interactive Proof Mode for Guardedness
2. Equality vs Equivalence
3. Automation
4. Interfacing with tools and proof general

Many of the problems we describe are exacerbated by the size of the LVC development, and probably do not surface in developments that have only a few thousand lines. Before we introduced the `smpl` plugin described below, a recompile of LVC took 45 minutes. This made the development painful, and it made it especially costly to experiment with different definitions, as any alteration would result in hours spend waiting on Coq to recompile. LVC currently takes roughly 25 minutes to recompile.

[1] See https://sigurdschneider.github.io/lvc/Lvc.Compiler.html#Compiler.

17.6.1 An Interactive Proof Mode for Guardedness

Coq should support an interactive mode where guardedness of a definition can be shown by transforming a term with tactics, similar to the usual interactive proofs for theorems in Coq.

Coq currently features an automated guardedness check, the purpose of which is to decide whether a given function is total or not. There are two different guardedness checkers, one for fixpoints and another for cofixpoints. These checkers are trusted components that essentially return a single boolean to the Coq kernel indicating whether the (co)fixpoint definition under consideration is total or not. Each guardedness checkers use a syntactic criterion that is sufficient for totality. In the case of fixpoints, for example, the function is examined to determine whether the recursive calls are on a strict subterm of the decreasing argument.

The general design philosophy seems to be that fixpoints that are not accepted by the guardedness checker should be encoded using induction on an appropriate instance of the general accessibility predicate. Coq Plugins (i.e. the Program package [Soz07]) exist that help with this task. The drawback of encoding definitions is that conversion is usually much weaker for such definitions, which especially is a problem in inductive proofs. The situation for cofixpoints is even more severe, because the guardedness checker accepts only few definitions and no general accessibility predicate is available.

As a first step, the user should be able to interactively proof totality of fixpoints and cofixpoints. To enable this, the rules by which the guardedness checker operates must be explicated to the Coq user, which is probably valuable in its own right. The old guardedness checkers could be provided as tactics. As a second step, the set of guarded (co)fixed-points that can be directly defined in Coq should be increased beyond what guardedness today admits. The advantage would be that more definitions could be expressed without encodings, and are hence not cluttered by additional proof arguments.

This could also be used as an opportunity to improve the guard condition for cofixpoints. We were very unsatisfied with having to resort to encodings for our simulation definitions, as discussed in §5. We think that Coquand [Coq93] explanation of productivity is more intuitive than encodings, and would like Coq to adopt Coquand's approach in a more explicit way.

17.6.2 Equality and Equivalence

The fact that there is a fundamental issue with equality in Coq has spawned the area of HoTT [Uni13]. The LVC development has to deal with equivalence relations. A prime example is the type of finite sets that may use different trees to represent the same set. All set operations, however, yield the same results on different represen-

tations of the same set. The library comes with proofs that (almost) all operations respect this extensional abstraction boundary, and the ones that do not can be change to do so. However, Coq does not provide a means to put up an abstraction barrier, and allow all outside users to identify the extensional equivalence with equality, which would be justified because the interface to the abstraction provably cannot distinguish sets on a more fine-grained basis. At the moment, the only sharp weapon against the equivalence problem is setoid rewriting. The importance of this issue is exacerbated by the fact that almost all libraries and tactics in Coq treat equality specially. For example, the standard library uses equality is the only equivalence relation that matters and defines order theoretic structures only with respect to equality. This means that the set library we use has to provide its own order theoretic framework which allows to provide the equivalence relation on a given type.

17.6.3 Automation

Interoperability of Tactics

The tactics that come with Coq are *island solutions*, that do not interact well with each other. Prime example are the tactic Ω, that solves Presburger arithmetic, the `congruence` tactic, that reasons up to a syntactic congruence relation, and the backward chaining `eauto` proof search. These three tactics, which each for itself are already powerful, do unfortunately not interact. For example, a goal which requires to show $fx = fy$ cannot be solved by Ω even if $x = y$ can be shown by Ω, and so on. Some level of interaction can be obtained by integrating them into an custom Ltac tactic, but this is brittle and slow.

Caching for `eauto`

The tactic `eauto` ignores the relationship between goals during proof search. In fact, proof search is completely redone for each subgoals and there is no caching. During interactive proof construction, the changes to the proof goal are usually minimal. Hence the set of derivable statements has a considerable overlap between a goal and a successor goal obtained by applying a tactic. It is also reasonable to assume that the congruence closure (also with respect to user-defined equivalences) does not change drastically by applying a tactic. None of this is used to improve proof automation. Caching eauto search trees would probably yield great speed improvements in many contexts, even if memory usage would increase. A partial eauto search tree is essentially a lemma, and thus much infrastructure for managing such a cache likely already exists. A concrete application for caching would be the instances of compatibility proofs generated during setoid rewriting.

Better Tooling for Setoid Rewriting

In LVC, we observed that certain instances are re-proven in every setoid rewrite, which slowed down Coq compilation time tremendously. We manually added the instances that took the longest to discover. This, however, this was a frustrating task as it involved reading setoid rewrite logs, which could easily grow to over 60k lines and are hard to read even when short. This issue has been recognized and there is Coq bug #6141 with discussion.

Additionally, one has to understand the proof search setoid performs, because sometimes adding more instances slows down rewriting, sometimes unification is slowing things down, and so on. We also discovered that wrapping rewrites in matches on the term to rewrite and then rewriting with a (partially) instantiated lemma speeds up the rewrite considerably. Discovering these intricacies takes a lot of time, and setoid could do a better job reporting about the time it spends, and how it spends it. For example, if setoid would also associate the time it took to first generate an instance, and report the type of the instances that took the longest to discover on request, this would save a lot of time and effort. Additionally, setoid could cache instances that took long to discover automatically. Setoid could also cache instances that it failed to prove, although this cache would have to be invalidated whenever a new instance becomes available or setoid search parameters are changed. If such features would have been available, LVC would probably have been a lot faster to compile in the first place.

17.6.4 Interfacing with Tools and Proof General

At the scale of LVC (including the libraries it embedds), the lack of certain features in the IDE is a huge problem: Simple tasks become a huge time sink; this problem is especially dire because Coq recompilation times are high (at some point LVC took 45 minutes to compile).

The main interface to Coq is Emacs' Proof General. There is the `company-coq` plugin for emacs, which supports the user with convenience features. While the emacs plugin is a great help, it does not go far enough. For example, there source navigation "go-to definition" has only been added recently to `company-coq`. It is impossible to get a list of uses of a definition, one has to resort to `grep` with all its limitations. These are probably things that need fixing inside of Coq, not on the Emacs side.

Especially painful is the fact that there are no refactoring tools. Definitions cannot be automatically renamed, it is not possible to easily determine if a lemma is still used, if one changes the arguments of a constructor many proofs break, etc. Even if all this refactoring was available, the text-only regime of emacs makes it very difficult to display information in a sensible way.

Name generation in Coq is a problem. Unfortunately, given the number of names required in the proofs of LVC, manual naming is not an option. Coq could provide more control over the automatically generated names. For example, a command that ensures that all names a tactic generates receive a certain prefix would be useful.

17.7 Custom Tactic Support

To improve tooling for the proofs for LVC, we developed a Coq plugin, several native tactics (i.e. implemented in Ocaml) and over 500 Ltac tactics.

17.7.1 The `smpl` Coq Plugin

We developed a Coq plugin that helps cleaning up Coq goals. In our development, we often have premises that need to be inverted to be useful. Examples include:

- Terms related by a partial order. For example, if $x \sqsubseteq (y, z)$ we can conclude that there must be a, b such that $x = (a, b)$ and $a \sqsubseteq y$ and $b \sqsubseteq z$.
- The development contains a relation that realizes an "at-n" predicate for lists. Often, the list appearing in the predicate is generated by map or zip. An important inversion lemma for map is that if the list map $f\ L$ has an element x at position n, then the list L has an element y at position n, and $x = f\ y$.
- Trivial or absurd premises, such as $x = x$ or the list equality $L = a, L$.

To deal with such premises, we need a tactic that cleans up the goal, i.e. gets rid of absurd goals and trivial premises, and does a reasonable amount of inversions. For such a tactic to be practical, it must be fast and extensible in a modular way. We are not aware of a built-in in Coq that satisfies these requirements, so we built a plugin. The `smpl` plugin consists of the following components:

- The Coq command `Smpl Create` *name*`.` creates a list of tactics with priorities which can be referred to by *name*. The tactics in the list are always sorted according to their priorities.
- The Coq command `Smpl Add` p t `:` *name*`.` inserts the tactic t with priority p into the list with name *name*.
- When including two modules via `Require` that add to a list with the same name, the insertions of both modules are taken into account.
- The Coq tactic `smpl` *name* $a_1 \ldots a_n$`.` executes the tactics from the list with name *name* in order until one of them succeeds, thereby passing the arguments a_1 to a_n to each of them.

The sources of the `smpl` plugin and its technical documentation is available on GitHub: https://github.com/sigurdschneider/smpl.

17.7.2 Native Tactics

We developed the following native tactics in OCaml to support the development of LVC.

- A tactic `is_param` c n that succeeds if c is a (possibly curried) application of a type constructor, and its n-th argument is a proper parameter for that type constructor. This tactic is useful to determine which arguments of an inductive type constructor need to be generalized for induction.
- A tactic `is_constructor_app` c that determines whether the term c is a application of a constructor.
- A tactic `is_inductive` c that determines whether the term c is an inductive type.
- A tactic `warntime` n s t that executes tactic t and prints s if execution of t exceeded n milliseconds.

17.7.3 Automating Inductive Generalization

Many proofs in LVC require generalizing inductive hypotheses. This involves the introduction of equations for indices of inductive predicates. However, generating such equations for parameters is never required and introduces unnecessary complexity. We define an Ltac tactic called `general induction` that is similar to `induction`, but introduces equations for indices of inductive types and maximally generalizes the goal before doing induction. We use the tactic

`is_param` c n

described above to avoid introducing equations for proper parameters. The tactic `general induction` also takes care of cleaning up the goals corresponding to the inductive cases, which gets rid of (most of) the equations again.

18 Related Work

This section discusses related work in depth. Many sections include a paragraph that explicitly compares LVC to related work and is marked with **LVC**.

18.1 Control Flow Graphs, Reaching Definitions, Liveness

The importance of control-flow analysis for compilation has been realized early. The origins of such analyses, including reaching definitions and live variables seem to be difficult to attribute. Hecht [Hec77] wrote in 1977 that the definitions appear in a large number of papers. According to Seidl, Wilhelm, and Hack [SWH12], the notion of true liveness originates from Giegerich, Möncke, and Wilhelm [GMW81]. For a discussion about different representations for control-flow, dominance relationships and references to literature from before the 1970s, we refer the reader to Allen [All70]. An important notion is that of a reaching definition [All70] in a control-flow graph (CFG): A definition of a variable reaches a use of the same variable, if there is a path in the CFG from the definition to the use.

LVC The notion of liveness (see §9) is central to LVC, in particular, it is the basis for coherence. Coherence is based on a reaching definitions analysis, and an alternate way to explain coherence is that it ensures that every use has at most one reaching definition.

18.2 Static Single Assignment

A key problem for imperative programs is that they cannot be renamed apart without further ado. The problem are join-points in the CFG that are reached by two different definitions of a the same variable. The value of the variable at the join-point hence depends on the predecessor from which the join-point was reached. If either definition is renamed, this relationship is lost.

Static single assignment (SSA) solves this problem by introducing the notion of a ϕ-function, and placing phony assignments of the form $x = \phi(y_1, \ldots, y_n)$ at every join-point that is reached by $n > 1$ different definitions y_i of x. Upon execution, the ϕ-function chooses one of its arguments depending on the predecessor block

from which the join-point was reached. Once such ϕ-assignments are in place, the imperative program can be renamed apart without losing information.

SSA is the culmination point of a line of research in data-flow analysis conducted during the 1980s. ϕ-functions appear first at POPL 1989 in two different papers. One is due to Rosen, Wegman, and Zadeck [RWZ88], the other is due to Alpern, Wegman, and Zadeck [AWZ88]. Zadeck [Zad09] gave a talk about the history of SSA. The standard construction algorithm for SSA [Cyt+91] also happens to be the standard reference for SSA. The standard construction algorithm uses a dominance ordering of the nodes in the CFG to determine where to introduce ϕ-assignments. An alternative construction algorithm, which works on partially constructed CFGs, is available [Bra+13].

Appel [App98] and Kelsey [Kel95] showed that the ϕ-assignments can be simulated by tail-call function applications. In particular, if a block in a CFG has i-many ϕ-assignments $x_i = \phi(y_{(i,1)}, \ldots, y_{(i,n)})$, then this CFG block can be encoded as a function with formal parameters $x_1, \ldots, x_i$. The k-th predecessor then tail-calls this function with arguments $y_{(1,k)}, \ldots, y_{(i,k)}$. Based on this observation, Kelsey [Kel95] seems to be the first to provide an informal but detailed proof that SSA programs correspond a certain class of CPS programs. The correspondence, however, had been noted before Kelsey by others [App92; ODo93].

LVC LVC follows Appel and Kelsey and uses a functional representation which encodes ϕ-assignments as function application. A key contribution of LVC is the additional imperative interpretation for the functional language, which allows to approach SSA from a different angle. In LVC, we did not try to answer the question how the particular definition of SSA given by Cytron et al. [Cyt+91] could be formalized, but approached the problem from the more general angle how an imperative program can be renamed apart. This approach allowed us to get rid of CFGs, ϕ-nodes, and, most importantly, dominance constraints and replace them by a language with nested mutually recursive functions and two semantic interpretations.

18.3 SSA Optimizations

One original goal of SSA was to lower the complexity of reaching definition analysis: In an imperative program, the reaching definition relation is potentially quadratic in the program size. This can be seen by assuming only one variable and observing that every definition of this variable could potentially reach every use. If, on the other hand, every variable has at most one definition, the reaching definition relation is trivially linear in the program size.

As it turns out, renaming apart the program, which is only possible in SSA, improves the complexity of many value optimizations. This is the case because having

unique variable names allows to track one abstract analysis values per a variable for the whole program, instead of tracking one abstract value per variable and program point. This approach reduces the analysis complexity typically by a factor proportional to the program size. A downside is that the approach precludes context sensitive analyses. For example, such a SSA analysis cannot conclude that inside the consequence of a conditional with condition $x = 2$, that x has the value 2 without losing the asymptotic complexity advantage. Nevertheless, many elaborate optimizations are formulated primarily as SSA optimizations. Examples of SSA optimizations include global value numbering (GVN) [RWZ88], sparse conditional constant propagation (SCCP) [WZ91b], and partial redundancy elimination (PRE) [VH04].

LVC LVC includes a verified implementation of SCCP (see §14.3). SCCP has been reformulated for a functional language by Chakravarty, Keller, and Zadarnowski [CKZ03] in an informal setting. Our verification of SCCP is based on a framework (see §14) that we hope can also be used to verify GVN, and with some extensions, PRE.

18.4 Continuation Passing Style

Continuation Passing Style (CPS) has played a fundamental role for both practical compilation and programming languages research in general. We refer to the survey of Reynolds [Rey93] for an overview over the different contexts in which continuations have been discovered. In our opinion, the key insight behind CPS is that higher-order arguments can be used to encode a wide variety of control-flow behavior. For example, CPS-bases models are known to give semantically precise accounts of the function return mechanism in assembly language (return pointer) and exceptions. Famously, CPS has been used by Appel [App92] to construct compilers for a wide range of languages.

The transformation to CPS has received considerable attention. The transformation to CPS produces a large number of λ-terms. Danvy and Filinski [DF92] single out *administrative* λ-terms and devise a restricted form of β-reduction to eliminate them. Sabry and Felleisen [SF93] prove the resulting reduction system confluent and terminating, a result which gives rise to administrative normal form (ANF). Flanagan et al. [Fla+04] show that ANF form can be constructed directly from a functional source language without applying a CPS transformation first. ANF seems to be similar to linear programs mentioned in Kelsey and Hudak [KH89], which refer to Plotkin [Plo75] as source for the transformation they use to achieve linearity.

Chakravarty, Keller, and Zadarnowski [CKZ03] reformulates SSA-based sparse conditional constant propagation [WZ91b] on a functional language in ANF. Our inter-

mediate language is similar to a subset of the language used by Chakravarty, Keller, and Zadarnowski [CKZ03], but is not in ANF.

LVC The first-order restriction of IL does not permit higher-order arguments, and hence rules out CPS. However, IL also has a tail call restriction, so it is valid to see IL programs as a special case of CPS programs, where the continuation is always statically known.

18.5 Reducibility and Recursive Functions

Allen [All70] classifies flow graphs as either reducible or irreducible. Hecht and Ullman [HU72] provides a second characterization of reducibility and shows that structured control flow always yields a reducible control-flow graph, while unstructured control flow may yield an irreducible control-flow graph. See Hecht and Ullman [HU74] for a list of alternative characterizations of reducibility.

LVC LVC offers mutually recursive functions, which can directly express irreducible control flow.

18.6 Verified Compilers

18.6.1 C-like Languages

Two major verification projects for compilers of C-like languages exist. CompCert [Ler09b] is a verified compiler for a realistic subset of the C language. CompCert follows a conservative design without SSA, which includes constant propagation, local common subexpression elimination, and register allocation as central optimizations. Leroy [Ler09a] notes that SSA was originally not integrated into CompCert because its advantages for correctness proofs were not immediately clear. For CompCert, the SSA-based optimizations GVN is available via the CompCertSSA project [BDP12]. CompCertSSA adds a SSA translation pass to CompCert. Register allocation is not performed on SSA.

The VeLLVM Project [Zha+12; Zha+13; ZZ12] is an ongoing effort to verify the production compiler LLVM [LA04] including its advanced, production-grade optimizations. The VeLLVM project has recently completed all steps [Zha+12; ZZ12; Zha+13] to verify the standard SSA-construction algorithm [Cyt+91]. The intermediate language of LLVM is an imperative intermediate language with ϕ-functions to enable SSA [Zha+12].

LVC Our approach leverages the correspondence between functions with parameters and blocks with ϕ-functions and uses the intermediate language IL with coher-

ence to formalize SSA and be able to flexibly break and re-establish the SSA invariant. In contrast, Both VeLLVM and CompCertSSA use a global invariant to ensure the SSA condition, and have to maintain that invariant during every transformation.

18.6.2 Functional Languages

Chlipala [Chl10] proves correctness for a compiler from Mini-ML to assembly including mutable references, but without system calls. Register assignment uses an interference graph constructed from liveness information. Chlipala restricts functions to take exactly one argument and requires the program to be closure converted prior to register assignment. This means liveness coincides with free variables and values shared or passed between functions reside in an (argument) tuple in the heap: Effectively, register assignment is function local. Chlipala does not prove bounds on the number of different variables used after register assignment and does not investigate the relationship to α-equivalence.

CakeML [Owe+16] is a verified compiler for a substantial subset of Standard ML. The compiler for CakeML is verified in HOL4. The compiler represents loops as recursive functions and forces all variables a function uses to be parameters through closure conversion. This breaks all live ranges at loop headers. The CakeML compiler assumes all function parameters are live, hence register pressure may increase if closure conversion introduces dead parameters. CakeML does not use SSA with ϕ-functions and delegates register allocation to a non-SSA-based, verified IRC algorithm [GA96] that performs spilling and register assignment in one phase in an intertwined fashion. In contrast to the CakeML approach, our approach is SSA-based, separates spilling from register assignment, and allows fine-grained control over live range splitting. Our approach does not require closure conversion, but allows functions to refer to variables that are not parameters.

18.7 Translation Validation

Translation validation [PSS98] is widely used in verified compilers. Translation validation means that the translation itself is not verified, but a (usually simpler) validator decides afterwards whether the translation was correct. An explanation of the significance of translation validation for compiler verification has been given by Leroy [Ler09b].

LVC While LVC supports translation validation to attain flexibility, we verified all translation steps in LVC.

18.8 Bisimulations

Alternative characterizations of contextual equivalence for the purpose of proof are in the literature: Logical relations [Pit04] and bisimulations [Pit12] are the standard examples. Recently, Hur et al. [Hur+12] have presented a hybrid approach of bisimulations and logical relations. Bisimulations emerged in the analysis of concurrent systems [BK08]. Our first-order setting simplifies proving the congruence property of our simulation.

Simulations have been used for proving semantic preservation in CompCert and derived projects [Ler09a]. CakeML uses an evaluation function with a step limit that counts function applications to specify the semantics. This approach supports inductive proofs on the step limit.

18.8.1 Inductive Proofs for Bisimulations

The idea to use compatibility lemmas to simplify correctness proofs is outlined in §2.8 of the master thesis of one of the authors [Sch13]. The master thesis uses a version of IL without mutual recursion and system calls. A basic version of the extension lemma, which enables the inductive method and which we prove in §8.3.2, appears in the master thesis as Lemma 3. The masters thesis uses the extension lemma to show that contextual equivalence is characterized by a simulation-based definition. In §8.2 of this thesis we show that a bisimulation-based definition is sound for contextual equivalence.

Neis et al. [Nei+15] recently used an inductive method to deal with stuttering steps when proving their elaborate parametric inter-language simulations (PILS). In the PILS framework, they verify a compiler for an imperative higher-order language with non-mutually recursive functions that take a fixed number of arguments.

Neis et al. use an inductive method to deal with stuttering steps in, among other things, the correctness proof of a form of DCE with respect to PILS that only eliminates unused let-bindings. Neis et al. mention that their framework provides a series of compatiblity lemmas simplifying the proof, but do not state the precise form of the lemmas in the paper. Our DCE removes dead function parameters, unused function definitions, and unreachable branches of conditionals.

18.8.2 Correctness Arguments in Verified Compilers

The correctness arguments in VeLLVM [Zha+13], the verified LLVM project, CompCertSSA [BDP12], and CompCertTSO [Sev+13] are based on showing simulation diagrams, which, by a general lemma, provide for the existence of a suitable simulation relation.

Both CompCert and VeLLVM operate on a graph-based program representation, on which structural induction is not as useful as in our term-based setting.

Our determinism requirements are similar, but subtly weaker than the requirements found in the work of Sevcík et al. [Sev+13]. In contrast to their work, our IDRS are not necessarily receptive, and are only determinate for τ transitions (c.f. condition 6). This is because we view the system as being a part of the IDRS, while Sevcík et al. [Sev+13] view the system as an external component. In our setting, we use non-τ transitions to *inform* the semantic framework about non-determinism, very much in the sense of an instrumentation, while other work uses non-τ transitions to synchronize with an external process in the style of CCS [] or π-calculus. The advantage our approach is that we can verify implementations of system calls. One IDRS can give an external call a non-deterministic semantics, and another can replace it with an implementation of a system call that has less behavior (but at least *one* behavior). This means that forward and backward simulation on IDRS do not coincide, as the following example shows.

Example 18.1 Consider any state σ in an IDRS which can reduce with an external event α that non-deterministically returns one of its arguments to a stuck state. Now consider the a similar state σ' in a second IDRS, which can reduce with an external event α that returns its first argument to a stuck state. σ can simulate σ', but not vice versa.

18.9 Languages with Dual Interpretation

Kelsey and Hudak [KH89] constructed a simple compiler with an intermediate language that has both functional and imperative features. To produce an assembly program, the intermediate language is transformed to a subset of the language that corresponds to assembly, and this subset has a dual semantics. We extend this idea, integrate it with SSA construction, and formally prove correctness of our transformations.

Beringer, MacKenzie, and Stark [BMS03] used a language with a functional and imperative interpretation for proof carrying code. They give a sufficient condition for the two semantics to coincide which they call Grail normal form (GNF). GNF requires functions to not use the closure at all, i.e. a function must only depend on its parameters. This makes the notion significantly weaker than ours: The requirement that a function must only depend on its parameters corresponds to maximal insertion of ϕ-functions, hence GNF is not suitable for SSA construction. Register allocation is not considered.

Correspondences between imperative and functional programming languages have been investigated very early on [Lan65].

18.10 Research Compilers with Functional Intermediate Languages

Research compilers have employed (essentially) functional intermediate representations (IR) for at least a decade now. Compilers strive to normalize the source program aggressively; contextual equivalence facilitates normalization. Johnson and Mycroft [JM03] minimize code for embedded systems on a functional IR. Tate et al. [Tat+09] introduce the program expression graph (PEG), a graph-based IR with a lazy semantics. libFirm [Ger12] is a research compiler that uses a graph-based, functional IR for the compilation of C.

An important idea in compiler construction are sea-of-nodes intermediate representations put forward by Click and Cooper [CC95] and Click and Paleczny [CP95]. The key idea of the sea-of-nodes approach is to represent the program, including expressions such as additions, as a graph on which certain rewriting steps are sound with respect to program equivalence. Many optimizations can be characterized as rewriting operations and their correctness can be established by arguing that the rewrites steps are semantics preserving. The research compilers mentioned before [JM03; Tat+09; Ger12], and the production-grade Java HotSpot VM [PVC01] build to different degrees on the sea-of-nodes idea.

18.11 Register Allocation

Every register allocation approach must lower the register pressure sufficiently, and then find an assignment of variables to registers. Register pressure is lowered by temporarily moving values to memory, and reloading values when necessary. Spilling is the problem of determining where and what to spill and reload. Register assignment is the problem of finding an assignment from variables to registers while satisfying register constraints. In an imperative approach, spilling and register assignment are interdependent, and must hence be performed in one phase in an intertwined fashion.

In SSA, the spilling problem can be decoupled from the register assignment problem, because, the number of simultaneously live variables equals the register pressure [HGG06]. This means that SSA-based spilling algorithms can effectively determine how many variables must be spilled at each program point without a concrete register assignment. Braun and Hack [BH09] provide such a SSA-based spilling algorithm that is very sensitive to the underlying program structure.

SSA-based register allocation [HGG06] consists of three phases. First spills and loads are inserted into the program to lower the register pressure sufficiently. Second, a register assignment that assigns each pseudo-register a machine register is computed. Such an assignment can always be found in polynomial time. Note that at

this point, there are still ϕ-assignments at each join point that must be implemented by a parallel move. Finding an assignment that provides optimal coalescing, i.e. one that minimizes the number of copies required to implement the parallel moves at join-points is NP-complete. All practical SSA-based coalescing is hence based on heuristics. Coalescing is complicated by the fact that many architectures require a temporary register to implement the parallel move.

LVC LVC implements SSA-based register assignment and spilling, but does not attempt to optimize coalescing. In joint work with Julian Rosemann [RSH17], we developed an novel approach that we call "translation-validation with repair" and which we do not describe in this thesis.

18.11.1 Global register allocation

Chaitin [Cha82] pioneered register allocation. Since Chaitin's initial work, there have been several improvements to graph coloring that mostly concentrated on coalescing, i.e. the removal of copy instructions. Most graph coloring approaches decide for every variable globally whether it resides in a register (and if so, in which) or a spill slot. Especially, graph coloring allocators do not attempt to split live ranges sophistically but rather transfer spilled variables from/to memory upon each access. This gives a simple spilling scheme that is also amenable to formal verification (see below). However, in practice the spilling quality of these algorithms is not sufficient to achieve acceptable performance [BH09].

18.11.2 Register Allocation via Linear Scan

Poletto and Sarkar [PS99] devised linear scan, a register allocation approach that is the basis for many practically popular approaches to register allocation. Linear scan splits live ranges, i.e. it allows a variable to be in a register at one program point and in memory at another. For performance reasons, linear scan over-approximates the live ranges of variables by linearizing control flow, hence the name. Linear scan intertwines spilling and register assignment.

18.11.3 Computational Complexity of Register Allocation

Chaitin proves NP-completeness of global register allocation [Cha82]. Bouchez et al. show that minimizing spills and loads is NP-complete in SSA [BDR07]. Bouchez also shows NP-completeness of different coalescing problems, i.e. minimizing the number of copies/swaps required to implement SSA's ϕ-functions after the register allocation phase.

18.11.4 Register Allocation in Verified Compilers

There are several verified compilers which perform register allocation.

CompCert

Register allocation in the first version of CompCert used a translation validated graph coloring algorithm implemented in OCaml [Ler09a]. Spilling is verified and very simple: Variables not in a register are loaded before use and spilled after redefinition. Later Blazy et al. [BRA10] fully verified Appel's [GA96] iterated register coalescing (IRC) approach, which includes spilling. Being a graph coloring technique, this algorithm suffers from the same drawbacks concerning spilling that we discussed above. Hence, especially for machines with few registers (such as IA32), the code quality is hardly acceptable. Instead of changing the fully verified spiller, which would have been a tremendous effort, Rideau et al. [RL10] developed a new translation validated algorithm for register allocation and spilling. The new spilling algorithm tracks recently spilled and loaded variables and thus avoids loading if the variable is still in a temporary register.

In contrast to the verified register allocation by Blazy et al., the second spilling algorithm we verify as case study splits live ranges. The algorithm follows a strategy similar to the translation validated algorithm of Rideau et al, is verified, but does not support overlapping registers yet. There is a project that aims to bring SSA to CompCert [BDP12], but SSA-based register allocation for CompCert has not been explored yet.

LambdaTamer

Chlipala [Chl10] proves correctness for a compiler from Mini-ML to assembly in the context of the LambdaTamer project. The language includes mutable references but not system calls. Register assignment uses an interference graph constructed from liveness information. Chlipala does not use SSA form. Instead, Chlipala restricts functions to take exactly one argument and requires the program to be closure converted prior to register assignment, which is very similar to what CakeML does. This means liveness coincides with free variables and values shared or passed between functions reside in an (argument) tuple in the heap: Effectively, register assignment is function local. Chlipala considers spilling only indirectly: Some n variables are assigned to registers, the rest resides permanently in spill slots. This approach makes it unnecessary to prove bounds on the number of different variables used after register assignment. Chlipala does not investigate the relationship to α-equivalence.

CakeML

CakeML [Owe+16] is a verified compiler for a substantial subset of Standard ML. The compiler for CakeML [Tan+16] is verified in HOL4. CakeML uses 12 different intermediate languages. The compiler represents loops as recursive functions. Improving over Chlipala's LambdaTamer, CakeML allows functions to take more than one argument, but still uses a heap-allocated closure for variables occurring free in the function. CakeML also does not use SSA with ϕ-functions for register allocation, but translates to a sufficiently low-level language to use a non-SSA-based, verified IRC algorithm [GA96] that performs spilling and register assignment in one phase in an intertwined fashion.

LVC In contrast to the CakeML approach, our approach is first-order, SSA-based, and separates spilling from register assignment. Our approach does not require closure conversion, allows functions to refer to variables that are not parameters, and lets the spilling phase decide whether a variable it accesses (be it a parameter or a value from the closure) shall resides in a register or in a memory location upon invocation.

18.12 IL and SSA-properties

As previously discussed, since SSA and functional programming correspond [Kel95; App98], the translation from IL/I to IL can be seen as SSA construction [Cyt+91], and the translation from IL to IL/I as SSA destruction. In this section we discuss variants of SSA, and how they map to IL with coherence.

18.12.1 Pruned SSA

Pruned SSA adds the additional constraint that all ϕ-functions in SSA must be live. Recall that ϕ-functions in a CFG correspond to parameters in IL. IL supports pruned SSA because Lemma 12.4 shows that to obtain a coherent programs, only live parameters must be inserted.

Lemma 12.4 together with the correctness result in Theorem 12.8 for our SSA construction algorithm form §12.5 produces pruned SSA. Semi-Pruned SSA is a relaxation of pruned SSA, and hence also supported.

18.12.2 Critical Edges

Critical edges are artifacts that arise in CFGs that group straight-line code (i.e. code without conditional branches) in so called basic blocks, and view the CFG as a graph of basic blocks. In such a CFG, an edge is critical if neither the predecessor block of the edge nor the successor block is executed if and only if the edge is taken. This

means that there is no basic block in which code can be placed that should be executed if and only if the edge is taken. This is especially important in the phase where SSA's ϕ-nodes are lowered in basic-block-based compilers, because of the placement of the lowering code. Since IL is statement based, any statement can be pre-pended by any number of assignments (i.e. let-bindings), hence no critical edges exist and there is always a place for the lowering code.

18.12.3 Loop Closed SSA

Loop-closed SSA is variant of SSA which requires additional ϕ-nodes for variables defined in loop bodies: whenever variable defined inside the loop is used outside the loop body, it must go through a ϕ. Pop, Jouvelot and Silber [PJS08] mention loop-closed SSA; it is, however, supported in realistic, practical compilers such as GCC [PJS08; Con17], and LLVM [Goh09].

IL supports loop-closed SSA, but does not enforce it. To convert a program to loop-closed SSA, it suffices to ensure that the loop is left through a the application of a function with a sufficient number of arguments. See Figure 18.1 for an example.

```
1
2
3
4  fun f (x,y) =
5   let z = x + y
6   if (x / 5 = 0) then
7     let a = 13 + z in
8     a
9   else
10    f (x + 1, y*2)
11 in f (a,b)
```

```
1  fun g (z) =
2    let a = 13 + z in a
3  in
4  fun f (x,y) =
5   let z = x + y
6   if (x / 5 = 0) then
7     g (z)
8
9   else
10    f (x + 1, y*2)
11 in f (a,b)
```

Figure 18.1: An example not in loop-closed SSA (left) and a similar program in loop-closed SSA (right). To convert a program to loop-closed SSA, it suffices to ensure all branches that leave the loop consist of exactly one applications of a function with sufficient parameters.

19 Conclusion

We presented the design of the verified compiler LVC that realizes SSA advantages with the term-based language IL and the notion of coherence. We showed that IL and coherence realize the asymptotic complexity advantage for SSA program analyses that originally motivated the development of SSA by verifying SCCP in §14 and the associated analysis using the program analysis framework we described in §15. Furthermore, we implemented a SSA-specific version of register assignment in §13 and showed that IL and coherence allows to separate register allocation into two separate passes, which is known to only work on SSA programs. We integrated our approach with CompCert in §16 and our research prototype is able to produce PowerPC assembly code. We hence think that we have successfully demonstrated that IL and coherence provide a faithful and viable implementation of SSA in a verified setting.

Our approach based on coherence has the key advantage that the SSA invariant can be broken, because its semantic foundation is simply the semantics of IL/I and IL/F. LVC makes use of this feature during register allocation, where the SSA invariant is broken and reestablished as usual in SSA-based register allocation approaches. We think that a coherence-based approach is hence flexible and well suited for a verified setting.

Our formalization relies on a term-based language, in contrast to many other verification projects which are CFG-based. This allowed us to formulate many properties as inductive predicates. We leverage the inductive structure further by providing the custom proof method for bisimulation we developed in §8, which works by induction on the program structure and solves the problem of stutter-steps usually encountered in coinductive proofs.

For the formalization of optimizations, we followed a three-layer approach: An analysis algorithm that produces analysis information, a inductive predicate that specifies soundness, and sometimes completeness, of analysis information, and a transformation that uses the analysis information to transform the program. We think that this is a success: The correctness of the analysis is insulated from the correctness of the transformation by our inductive soundness specification. Furthermore, the inductive predicates allow us to stay away from instrumented semantics. Instrumented semantics are tailored towards the transformations they are used for, and in

CompCert and CakeML, which use this approach, this results in 9, and 12 different semantic specifications, respectively.

For analysis information, we often specify only soundness, for example for liveness information in §9. Correctness proofs only require soundness of analysis information, and our proofs make this explicit. Restricting the requirements to soundness only also provides flexibility: Many transformations must maintain sound liveness information, and had we also required completeness, we would also have had to maintain completeness across transformations. Completeness, however, often makes transformations more effective. For example, for reachability information, which we discussed in §11, we require completeness because we must eliminate all unreachable code. In general, we follow the paradigm that for a transformation, we only show correctness, but not effectiveness. For example, our analysis likely produces information which is as complete as it can be given undecidability of the problem. But we never have to deal with this fact, and never have to formally prove it. Another example is the nesting of function definitions discussed in §12.6: We conjecture that both SSA construction and register allocation are more effective if the program is deeper nested. For this reason, CFG-based SSA register allocation approaches require to process CFG nodes in dominance order; in LVC we show got rid of all dominance requirements and used the program structure instead, which is sufficient to prove correctness. This is due to the fact that program structure in IL is always sound wrt. the dominance order.

19.1 Future Work

There are several directions in which would be interesting to explore.

First, we would like to formally show that our IL-based approach is on par with CFG-based approaches with respect to quality of SSA construction and register allocation. Our term-based approach can be seen as essentially a AST-based approach. However, we think that, at least for the transformations we verified, our approach does not suffer from principal limitations in comparison with a CFG-based approach. We would like to formally show that the quality of SSA construction and register allocation on IL matches the quality that the same transformations on a CFG could obtain if the IL program is maximally nested with respect to the dominance order. This would require specifying the dominance order of function definitions on IL, devising an analysis that computes the optimal order, and another algorithm that reorders definitions to optimize the nesting structure.

Second, we developed a program logic for value optimizations, and would like to use it to verify global value numbering and partial redundancy elimination. For this purpose, we would need to extend the program logic with the ability to introduce new variables, which is currently not possible.

Third, we already worked into the direction of SMT-based translation validation in the context of LVC. We would like to integrate an SMT solver with the program logic to allow a more general form of translation validation. First experiments into this direction have been encouraging.

Finally, we would like to improve the integration with CompCert, and develop LVC to the point where it can replace the middle end in CompCert. Although much work remains to be done in this direction, we think that the results of this thesis show that functional SSA is a viable approach to integrating SSA in CompCert.

Bibliography

[All70] Frances E. Allen. “Control flow analysis”. In: *SIGPLAN Notices* 5.7 (July 1970) (cit. on pp. 189, 192).

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge, England: Cambridge University Press, 1992 (cit. on pp. 190, 191).

[App98] Andrew W. Appel. “SSA is Functional Programming”. In: *SIGPLAN Notices* 33.4 (Apr. 1998) (cit. on pp. 2, 120, 190, 199).

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. “Detecting Equality of Variables in Programs”. In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. San Diego, CA, USA, Jan. 10–13, 1988 (cit. on pp. 3, 190).

[BDP12] Gilles Barthe, Delphine Demange, and David Pichardie. “A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert”. In: *ESOP*. Vol. 7211. LNCS. Tallinn, Estonia, Mar. 24–Apr. 1, 2012 (cit. on pp. 1, 3, 7, 8, 192, 194, 198).

[BDR07] Florent Bouchez, Alain Darte, and Fabrice Rastello. “On the complexity of spill everywhere under SSA form”. In: *LCTES*. San Diego, California, USA, June 13–15, 2007 (cit. on p. 197).

[BH09] Matthias Braun and Sebastian Hack. “Register Spilling and Live-Range Splitting for SSA-Form Programs”. In: *CC*. Vol. 5501. LNCS. York, UK, Mar. 22–29, 2009 (cit. on pp. 140, 141, 196, 197).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Representation and Mind Series. Cambridge, MA, USA: The MIT Press, 2008 (cit. on p. 194).

[BMS03] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. “Grail: a Functional Form for Imperative Mobile Code”. In: *ENTCS* 85.1 (2003) (cit. on p. 195).

[Boe] Hans-J. Boehm. *ISO/IEC JTC1/SC22/WG14/N1528*. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1528.htm (visited on 05/02/2017) (cit. on p. 60).

[Bra+13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. “Simple and Efficient Construction of Static Single Assignment Form”. In: *Compiler Construction - 22nd International Conference. Proceedings*. Vol. 7791. LNCS. Rome, Italy, Mar. 16–24, 2013 (cit. on p. 190).

[BRA10] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. "Formal Verification of Coalescing Graph-Coloring Register Allocation". In: *ESOP*. Vol. 6012. LNCS. Paphos, Cyprus, Mar. 20–28, 2010 (cit. on pp. 4, 8, 198).

[Cac+05] David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. "Extracting a data flow analyser in constructive logic". In: *Theor. Comput. Sci.* 342.1 (2005) (cit. on p. 159).

[CC95] Cliff Click and Keith D. Cooper. "Combining Analyses, Combining Optimizations". In: *TOPLAS* 17.2 (1995) (cit. on p. 196).

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. "Automatic Construction of Sparse Data Flow Evaluation Graphs". In: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. Orlando, FL, Jan. 21–23, 1991 (cit. on p. 3).

[CH88] Thierry Coquand and Gérard P. Huet. "The Calculus of Constructions". In: *Inf. Comput.* 76.2/3 (1988) (cit. on p. 31).

[Cha82] Gregory J. Chaitin. "Register Allocation & Spilling via Graph Coloring". In: *PLDI*. Boston, Massachusetts, USA, June 23–25, 1982 (cit. on p. 197).

[Chl10] Adam Chlipala. "A verified compiler for an impure functional language". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Madrid, Spain, Jan. 17–23, 2010 (cit. on pp. 8, 193, 198).

[CKZ03] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. "A Functional Perspective on SSA Optimisation Algorithms". In: *ENTCS* 82.2 (2003) (cit. on pp. 191, 192).

[Con17] GCC Contributors. *GCC Documentation: Loop-closed SSA form*. 2017. URL: https://gcc.gnu.org/onlinedocs/gccint/LCSSA.html (cit. on p. 200).

[Coq93] Thierry Coquand. "Infinite Objects in Type Theory". In: *TYPES*. Vol. 806. LNCS. Nijmegen, The Netherlands, May 24–28, 1993 (cit. on pp. 60, 61, 183).

[CP95] Cliff Click and Michael Paleczny. "A Simple Graph-Based Intermediate Representation". In: *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. San Francisco, CA, USA, Jan. 22, 1995 (cit. on p. 196).

[Cyt+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "An Efficient Method of Computing Static Single Assignment Form". In: *POPL*. Austin, Texas, USA, Jan. 11–13, 1989 (cit. on p. 2).

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *TOPLAS* 13.4 (1991) (cit. on pp. 3, 8, 10, 120, 130, 190, 192, 199).

[DF92] Olivier Danvy and Andrzej Filinski. "Representing Control: A Study of the CPS Transformation". In: *MSCS* 2.4 (1992) (cit. on p. 191).

[Fla+04] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. "The Essence of Compiling with Continuations (with Retrospective)". In: *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*. 2004 (cit. on p. 191).

[GA96] Lal George and Andrew W. Appel. "Iterated Register Coalescing". In: *ACM Trans. Program. Lang. Syst.* 18.3 (1996) (cit. on pp. 8, 193, 198, 199).

[Ger12] et al Gerhard Goos. *libFirm – an SSA-based intermediate representation*. 2012. URL: http://pp.ipd.kit.edu/firm/ (cit. on p. 196).

[GMW81] Robert Giegerich, Ulrich Möncke, and Reinhard Wilhelm. "Invariance of Approximate Semantics with Respect to Program Transformations". In: *GI - 11. Jahrestagung in Verbindung mit Third Conference of the European Co-operation in Informatics (ECI). Proceedings*. Vol. 50. Informatik-Fachberichte. München, Germany, Oct. 20–23, 1981 (cit. on p. 189).

[Goh09] Dan Gohman. *ScalarEvolution and Loop Optimization*. Talk at the LLVM Developers' Meeting. 2009. URL: http://llvm.org/devmtg/2009-10/ (cit. on p. 200).

[Har13] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge, England: Cambridge University Press, 2013 (cit. on p. 41).

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. New York, NY, USA: Elsevier North-Holland, Inc., 1977 (cit. on p. 189).

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. "Register Allocation for Programs in SSA-Form". In: *CC*. Vol. 3923. LNCS. Vienna, Austria, Mar. 30–31, 2006 (cit. on pp. 1, 4, 8, 24, 131, 132, 142, 143, 196).

[HU72] Matthew S. Hecht and Jeffrey D. Ullman. "Flow Graph Reducibility". In: *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*. Denver, CO, USA, May 1–3, 1972 (cit. on p. 192).

[HU74] Matthew S. Hecht and Jeffrey D. Ullman. "Characterizations of Reducible Flow Graphs". In: *J. ACM* 21.3 (1974) (cit. on p. 192).

[Hur+12] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. "The Marriage of Bisimulations and Kripke Logical Relations". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Philadelphia, PA, USA, Jan. 22–28, 2012 (cit. on p. 194).

[Hur+13] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. "The power of parameterization in coinductive proof". In: *POPL*. Rome, Italy, Jan. 23–25, 2013 (cit. on pp. 5, 8, 55, 63, 64, 69).

[ISO11] ISO/IEC. *ISO 9899:2011 - Programming languages - C*. Geneva, Switzerland, 2011 (cit. on p. 60).

[JM03] Neil Johnson and Alan Mycroft. "Combined Code Motion and Register Allocation Using the Value State Dependence Graph". In: *Compiler Construction, 12th International Conference. Proceedings*. Vol. 2622. LNCS. Warsaw, Poland, Apr. 7–11, 2003 (cit. on p. 196).

[Jou+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. "A Formally-Verified C Static Analyzer". In: *POPL*. Mumbai, India, Jan. 15–17, 2015 (cit. on p. 159).

[Kel95] Richard A. Kelsey. "A correspondence between continuation passing style and static single assignment form". In: *SIGPLAN Not.* 30 (3 Mar. 1995). ISSN: 0362-1340 (cit. on pp. 2, 120, 190, 199).

[KH89] Richard Kelsey and Paul Hudak. "Realistic Compilation by Program Transformation". In: *POPL*. Austin, Texas, USA, Jan. 11–13, 1989 (cit. on pp. 5, 7, 191, 195).

[LA04] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *CGO*. San Jose, CA, USA, 2004 (cit. on p. 192).

[Lan65] Peter J. Landin. "Correspondence between ALGOL 60 and Church's Lambda-notation: part I". In: *CACM* 8.2 (1965) (cit. on p. 195).

[Ler09a] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *JAR* 43.4 (2009) (cit. on pp. 8, 55, 133, 141, 192, 194, 198).

[Ler09b] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *CACM* 52.7 (2009) (cit. on pp. 1, 5–8, 39, 48, 51, 55, 60, 61, 192, 193).

[Les11] Stéphane Lescuyer. "First-Class Containers in Coq". In: *Stud. Inform. Univ.* 9.1 (2011). URL: https://github.com/coq-contribs/containers/ (cit. on p. 33).

[Let08] Pierre Letouzey. "Extraction in Coq: An Overview". In: *CiE*. Vol. 5028. LNCS. Athens, Greece, June 15–20, 2008 (cit. on p. 31).

[May89] Cathy May. "The Parallel Assignment Problem Redefined". In: *IEEE Trans. Software Eng.* 15.6 (1989) (cit. on p. 147).

[McB00] Conor McBride. "Elimination with a Motive". In: *TYPES*. Vol. 2277. LNCS. Durham, UK, Dec. 8–12, 2000. URL: https://doi.org/10.1007/3-540-45842-5 (cit. on p. 182).

[Nei+15] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. "Pilsner: a compositionally verified compiler for a higher-order imperative language". In: *ICFP*. Vancouver, Canada, Sept. 1–3, 2015 (cit. on p. 194).

[ODo93] Ciaran O'Donnell. "High Level Compiling for Low Level Machines". In: *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*. Vol. A-23. IFIP Transactions. Orlando, FL, USA, Jan. 20–22, 1993 (cit. on p. 190).

[Owe+16] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. "Functional Big-Step Semantics". In: *ESOP*. Vol. 9632. Lecture Notes in Computer Science. Eindhoven, The Netherlands, Apr. 2–8, 2016 (cit. on pp. 7, 193, 199).

[Pic08] David Pichardie. "Building Certified Static Analysers by Modular Construction of Well-founded Lattices". In: *Electr. Notes Theor. Comput. Sci.* 212 (2008) (cit. on p. 159).

[Pie+17] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Version 5.0. http://www.cis.upenn.edu/~bcpierce/sf. Electronic textbook, 2017 (cit. on p. 71).

[Pit04] Andrew Pitts. In: Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. Cambridge, MA, USA, 2004. Chap. Typed Operational Reasoning (cit. on p. 194).

[Pit12] Andrew Pitts. In: *Advanced Topics in Bisimulation and Coinduction*. Cambridge, England, 2012. Chap. Howe's Method for Higher-Order Languages (cit. on p. 194).

[PJS08] Sebastian Pop, Pierre Jouvelot, and George André Silber. "In and Out of SSA : a Denotational Specification". In: *Static Single-Assignment Form Seminar*. Autrans, France, June 15–20, 2008. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-00915979 (cit. on p. 200).

[Plo75] Gordon D. Plotkin. "Call-by-Name, Call-by-Value and the lambda-Calculus". In: *Theor. Comput. Sci.* 1.2 (1975) (cit. on p. 191).

[Pou06] Damien Pous. "Weak Bisimulation Up to Elaboration". In: *CONCUR*. Vol. 4137. LNCS. Bonn, Germany, Aug. 27–30, 2006 (cit. on p. 68).

[Pou16] Damien Pous. "Coinduction All the Way Up". In: *LICS*. New York, NY, USA, July 5–8, 2016 (cit. on pp. 66, 67).

[PS99] Massimiliano Poletto and Vivek Sarkar. "Linear scan register allocation". In: *TOPLAS* 21.5 (1999) (cit. on p. 197).

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation Validation". In: *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Vol. 1384. LNCS. 1998 (cit. on pp. 6, 193).

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. "The Java HotSpot™ server compiler". In: *Symposium on Java™ Virtual Machine Research and Technology Symposium*. JVM'01. Monterey, California, 2001 (cit. on p. 196).

[Rey93] John C. Reynolds. "The Discoveries of Continuations". In: *LSC* 6.3-4 (1993) (cit. on p. 191).

[RL10] Silvain Rideau and Xavier Leroy. "Validating Register Allocation and Spilling". In: *CC*. Vol. 6011. LNCS. Paphos, Cyprus, Mar. 20–28, 2010 (cit. on pp. 8, 198).

[RSH17] Julian Rosemann, Sigurd Schneider, and Sebastian Hack. "Verified Spilling and Translation Validation with Repair". In: *ITP*. Vol. 10499. LNCS. Brasília, Brazil, Sept. 26–29, 2017 (cit. on pp. 6, 197).

[RSL08] Laurence Rideau, Bernard Paul Serpette, and Xavier Leroy. "Tilting at Windmills with Coq: Formal Verification of a Compilation Algorithm for Parallel Moves". In: *JAR* 40.4 (2008) (cit. on pp. 24, 144, 145, 147).

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Global Value Numbers and Redundant Computations". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. San Diego, CA, USA, Jan. 10–13, 1988 (cit. on pp. 3, 190, 191).

[Sch13] Sigurd Schneider. "Semantics of an Intermediate Language for Program Transformation". Master's Thesis. Saarland University, 2013. URL: http://www.ps.uni-saarland.de/~sdschn/master (cit. on pp. 179, 194).

[Sev+13] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency". In: *J. ACM* 60.3 (2013) (cit. on pp. 8, 55, 194, 195).

[SF93] Amr Sabry and Matthias Felleisen. “Reasoning about Programs in Continuation-Passing Style”. In: *LSC* 6.3-4 (1993) (cit. on p. 191).

[SM92] Davide Sangiorgi and Robin Milner. “The Problem of "Weak Bisimulation up to"”. In: *CONCUR*. Vol. 630. Lecture Notes in Computer Science. Stony Brook, NY, USA, Aug. 24–27, 1992 (cit. on p. 68).

[Soz07] Matthieu Sozeau. “Subset Coercions in Coq”. In: *TYPES*. Vol. 4502. LNCS. Nottingham, UK, Apr. 18–21, 2007 (cit. on p. 183).

[SS17] Steven Schäfer and Gert Smolka. “Tower Induction and Up-to Techniques for CCS with Fixed Points”. In: *RAMiCS*. Vol. 10226. LNCS. Lyon, France, May 15–18, 2017 (cit. on pp. 55, 66, 67, 182).

[SSH15] Sigurd Schneider, Gert Smolka, and Sebastian Hack. “A Linear First-Order Functional Intermediate Language for Verified Compilers”. In: *ITP*. Vol. 9236. LNCS. Nanjing, China, Aug. 24–27, 2015 (cit. on pp. 36, 140).

[SSH16] Sigurd Schneider, Gert Smolka, and Sebastian Hack. “An Inductive Proof Method for Simulation-based Compiler Correctness”. In: *CoRR* abs/1611.09606 (2016). URL: http://arxiv.org/abs/1611.09606 (cit. on p. 140).

[STS15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. “Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions”. In: *ITP*. Vol. 9236. LNCS. Nanjing, China, Aug. 24–27, 2015 (cit. on p. 181).

[SWH12] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Compiler Design - Analysis and Transformation*. New York, NY, USA: Springer, 2012 (cit. on p. 189).

[Tan+16] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. “A new verified compiler backend for CakeML”. In: *ICFP*. Nara, Japan, Sept. 18–22, 2016 (cit. on pp. 4, 5, 8, 199).

[Tat+09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. “Equality Saturation: a New Approach to Optimization”. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Savannah, GA, USA, Jan. 21–23, 2009 (cit. on p. 196).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cit. on p. 183).

[VH04] Thomas VanDrunen and Antony L. Hosking. “Value-Based Partial Redundancy Elimination”. In: *Compiler Construction, 13th International Conference. Proceedings*. Vol. 2985. LNCS. Barcelona, Spain, Mar. 29–Apr. 2, 2004 (cit. on p. 191).

[Win93] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993 (cit. on p. 71).

[WZ91a] Mark N. Wegman and F. Kenneth Zadeck. “Constant Propagation with Conditional Branches”. In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991). ISSN: 0164-0925 (cit. on pp. 1, 3, 149, 155).

[WZ91b] Mark N. Wegman and F. Kenneth Zadeck. “Constant Propagation with Conditional Branches”. In: *TOPLAS* 13.2 (1991) (cit. on p. 191).

[Zad09] Kenneth Zadeck. *The Development of Static Single Assignment Form*. Talk. 2009. URL: http://www.cdl.uni-saarland.de/ssasem/ (cit. on pp. 2, 190).

[Zha+12] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. "Formalizing LLVM Intermediate Representation for Verified Program Transformations". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Philadelphia, PA, USA, Jan. 22–28, 2012 (cit. on pp. 1, 7, 60, 192).

[Zha+13] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. "Formal Verification of SSA-based Optimizations for LLVM". In: *PLDI*. Seattle, WA, USA, June 16–19, 2013 (cit. on pp. 192, 194).

[ZZ12] Jianzhou Zhao and Steve Zdancewic. "Mechanized Verification of Computing Dominators for Formalizing Compilers". In: *Certified Programs and Proofs - Second International Conference. Proceedings*. Vol. 7679. LNCS. Kyoto, Japan, Dec. 13–15, 2012 (cit. on pp. 3, 192).