Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language

Georg Neis
MPI-SWS, Germany
neis@mpi-sws.org

Chung-Kil Hur [*]
Seoul National University,
South Korea
gil.hur@sf.snu.ac.kr

Jan-Oliver Kaiser
MPI-SWS, Germany
janno@mpi-sws.org

Craig McLaughlin
University of Glasgow, UK
mr_mcl@live.co.uk

Derek Dreyer
MPI-SWS, Germany
dreyer@mpi-sws.org

Viktor Vafeiadis
MPI-SWS, Germany
viktor@mpi-sws.org

Abstract

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

In this paper, building on prior work of Hur *et al.*, we develop a novel approach to compositional compiler verification based on *parametric inter-language simulations (PILS)*. PILS are *modular*: they enable compiler verification in a manner that supports separate compilation. PILS are *transitive*: we use them to verify **Pilsner**, a simple (but non-trivial) *multi-pass* optimizing compiler (programmed in Coq) from an ML-like source language $\mathbf{S}$ to an assembly-like target language $\mathbf{T}$, going through a CPS-based intermediate language. Pilsner is **the first multi-pass compiler for a higher-order imperative language to be *compositionally* verified**. Lastly, PILS are *flexible*: we use them to additionally verify (1) Zwickel, a direct non-optimizing compiler for $\mathbf{S}$, and (2) a hand-coded self-modifying $\mathbf{T}$ module, proven correct w.r.t. an $\mathbf{S}$-level specification. The output of Zwickel and the self-modifying $\mathbf{T}$ module can then be safely linked together with the output of Pilsner. All together, this has been a significant undertaking, involving several person-years of work and over 55,000 lines of Coq.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

Keywords Compositional compiler verification, parametric simulations, higher-order state, recursive types, abstract types, transitivity

1. Introduction

Most verification tools operate on programs written in high-level languages, which must be compiled down to machine-level languages prior to execution. The compiler is simply trusted to "preserve the semantics" of its source language (and hence preserve confidence in the high-level verification). Unfortunately, this trust is not well founded. For instance, recent work of Le *et al.* [14] identified 147 confirmed bugs in the industrial-strength GCC and LLVM compilers, of which 95 were violations of semantics preservation.

The goal of compiler verification is to eliminate the need for trust in compilation by providing a formal, machine-checked guarantee that a compiler is semantics-preserving. Toward this end, the most successful project so far has been CompCert [15], a verified optimizing compiler for a significant subset of C that was developed by Leroy and collaborators using the Coq proof assistant. Indeed, Le *et al.* [14] report that, despite extensive testing, they were unable to uncover a single bug in CompCert.

Now one may rightly ask: what does it mean for a compiler to "preserve the semantics" of the source program it is compiling? The standard answer, adopted by CompCert, is as follows. Suppose we are working with a distinguished *source* language $\mathbf{S}$ and a distinguished *target* language $\mathbf{T}$. Given a *whole* $\mathbf{S}$ program p_S, the compiler's output $tr(p_S)$ should be a whole $\mathbf{T}$ program p_T that *refines* p_S. Refinement means that any observable behavior of p_T should also be a valid observable behavior of p_S, for some common notion of "observable behavior" that both the $\mathbf{S}$ and $\mathbf{T}$ languages share (*e.g.,* termination, I/O events).

Although the above definition of semantics preservation is perfectly suitable for whole-program compilers, it says nothing about separate compilation. In practice, many programs are linked together from multiple separately-compiled modules, some of which may be compiled using different compilers or even hand-optimized in assembly. Is it possible to define a *compositional* notion of semantics preservation that says what it means for a single *module* in a program to be compiled correctly, while assuming as little as possible about how the other modules in the program are compiled?

We are not the first to broach this question—it has been an active research topic in recent years. The natural starting point is the notion of *contextual refinement*: target module m_T contextually refines source module m_S if $C[m_T]$ refines $C[m_S]$ for all closing program contexts C. However, contextual refinement fundamentally assumes that m_T and m_S are modules written in the same language, since they are linked with the same program context C. Thus, when defining semantics preservation between very different source and target languages, one must either find a way of embedding both languages

[*] Corresponding author.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...$15.00
http://dx.doi.org/10.1145/2784731.2784764

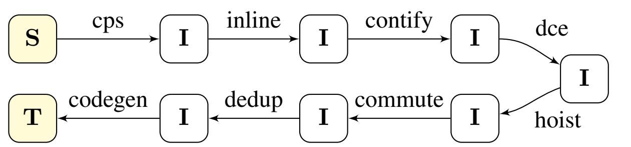


Figure 1. Structure of the Pilsner compiler

in a multi-language semantics [19] (so that contextual refinement remains on the table) or else pursue more complex alternatives to contextual refinement. We leave a more detailed discussion of prior work until §8, but we argue that all previously proposed solutions are lacking in some dimension of compositionality. In particular, we articulate the following three desiderata for a compositional notion of semantics preservation:

- **Modularity:** To enable verified separate compilation, semantics preservation (aka refinement) should be defined at the level of modules, not just whole programs, and it should be preserved under linking. Specifically, suppose that source ($\mathbf{S}$) module m_S is refined by target ($\mathbf{T}$) module m_T, and that $\mathbf{S}$ module m'_S is refined by $\mathbf{T}$ module m'_T. We should then be able to conclude that the $\mathbf{S}$-level linking of m_S and m'_S is refined by the $\mathbf{T}$-level linking of m_T and m'_T. (This is sometimes referred to as "horizontal compositionality".)
- **Transitivity:** Proofs of semantic preservation should be transitively composable. That is, one should be able to prove a compiler correct by verifying refinement for its constituent passes independently and then linking the results together by transitivity. (This is sometimes referred to as "vertical compositionality".)
- **Flexibility:** It should be possible to prove semantics preservation for a range of different compilers and program transformations, so that the results of different verified compilers (which might employ different intermediate languages) can be safely linked together, and so that hand-optimized and hand-verified machine code can be safely linked with compiler-generated code.[1]

In this paper, we present a new technique, particularly suited to compositional compiler verification for higher-order imperative (ML-like) languages, which we call **parametric inter-language simulations (PILS)**. PILS synthesize and improve on two pieces of prior work: (1) Hur *et al.*'s work on *parametric bisimulations* [9, 10] (originally called "relation transition systems"), and (2) Hur and Dreyer's work on a *Kripke logical relation* (KLR) between ML and assembly [8].

Parametric bisimulations are a simulation method for higher-order imperative languages, designed to support proofs that are **modular**, **transitive**, and capable in principle of generalizing to inter-language reasoning, *i.e.,* reasoning about relations between programs in different languages. However, Hur *et al.* only actually used them to prove contextual equivalences between programs in a *single* (high-level) language, leaving open the question of whether the generalization to inter-language reasoning would pan out.

Hur and Dreyer's work, on the other hand, was precisely targeted at supporting inter-language reasoning. Their Kripke logical relations introduced a rich and **flexible** notion of Kripke structures (possible worlds), with which they modeled the protocols governing calling conventions and the invariants connecting the different representations of data in the source and target languages of a compiler. As a proof of concept, they demonstrated the extreme flexibility of these Kripke structures by using them to verify the correctness of a deliberately obfuscated piece of self-modifying assembly code with respect to an ML-level function. However, due to limitations of their logical-relations method—in particular the lack of transitivity—their Kripke structures were only applicable to *single-pass* compilers.

PILS marry the benefits of these approaches together:

- PILS are *modular*: they enable compiler verification in a way that supports separate compilation and is preserved under linking.
- PILS are *transitive*: we use them to verify **Pilsner**, a simple (but non-trivial) *multi-pass* optimizing compiler from an ML-like source language $\mathbf{S}$ (supporting recursive types, abstract types, and general references) to an idealized assembly-like target language $\mathbf{T}$, going through a CPS-based intermediate language $\mathbf{I}$. After CPS conversion, Pilsner performs several optimizations at the $\mathbf{I}$ level prior to code generation. These optimizations include function inlining, contification, dead code elimination, and hoisting (Figure 1). Although Pilsner is relatively simple—it is not nearly as realistic as the (whole-program) verified CakeML compiler, for instance [13]—it is **the first multi-pass compiler for a higher-order imperative language to be *compositionally* verified.**
- PILS are *flexible*: we use them to additionally verify (1) **Zwickel**, a direct (one-pass) non-optimizing compiler from $\mathbf{S}$ to $\mathbf{T}$, and (2) Hur and Dreyer's aforementioned self-modifying code example, programmed as a $\mathbf{T}$ module and proven correct w.r.t. an $\mathbf{S}$-level specification. Thanks to PILS' modularity, the output of Zwickel and the self-modifying $\mathbf{T}$ module can then be safely linked together with the output of Pilsner.

All these results, together with the metatheory of PILS, have been mechanically verified in Coq—a significant undertaking, involving several person-years of work and over 55,000 lines of Coq.

In the rest of the paper, we give a high-level overview of our main results (§2), we review the basic idea behind parametric bisimulations which is also the core of PILS (§3), we describe the structure and some details of the PILS used in Pilsner and Zwickel (§4–§6), we highlight interesting aspects of the Pilsner verification (§7), and we conclude with discussion and related work (§8).

2. Results

2.1 Modularity

The goal of compiler correctness is to obtain a formal guarantee that the program that comes out of the compiler behaves the same as (or refines) the program that went in, according to a mathematical model of the source and target language in question. Traditionally, research on compiler correctness has focused on *whole* program compilation and does not support separate compilation. In separate compilation, the source program consists of several source modules, which are independently compiled to target modules. These target modules are then linked together, creating the final program. Note that different source modules may very well be compiled by different compilers.

We now illustrate how our approach, PILS, supports such separate compilation (and in fact even more heterogeneous scenarios). We consider the setting of a high-level ML-like source language $\mathbf{S}$ and a low-level machine target language $\mathbf{T}$ (for details, see §4).

[1] **Note:** In our model of the semantics preservation problem, every module in a program is represented by both an $\mathbf{S}$ and a $\mathbf{T}$ version, and we aim to prove that the $\mathbf{T}$ version refines its $\mathbf{S}$ counterpart. For modules that are compiled by a verified compiler, the $\mathbf{T}$ version is generated automatically by the compiler. But for any module that is hand-coded in $\mathbf{T}$, one must also manually supply its $\mathbf{S}$ counterpart, which serves as a "specification" that the hand-coded $\mathbf{T}$ module is then proven to refine. (We will see an example of this in §2.3.) This means that we can only account for hand-coded $\mathbf{T}$ modules that have *some* $\mathbf{S}$-level counterpart. This is somewhat of a restriction at present, since we focus here on the setting where $\mathbf{S}$ is a high-level, ML-like language and $\mathbf{T}$ a low-level, assembly-like language, and certainly not all assembly modules have an ML-level counterpart. However, we do not view this as a fundamental restriction: there is nothing in principle preventing us from generalizing our approach to a setting where $\mathbf{S}$ itself supports interoperation between high- and low-level modules. We discuss this point further in §8.

The main component of our development is a relation between target modules and source modules: $\Gamma \vdash M_{\mathbf{T}} \precsim_{\mathbf{TS}} M_{\mathbf{S}} : \Gamma'$ intuitively states that target module $M_{\mathbf{T}}$ refines source module $M_{\mathbf{S}}$ and that they import the functions listed in Γ and export those in Γ'. The first key result, Theorem 1, applies to whole programs, *i.e.,* well-typed modules that import nothing and export at least a main function (F_{main}) of appropriate type. It states that our relation implies the standard behavioral refinement.

Theorem 1 (Adequacy for whole programs)**.**

$$\frac{\cdot \vdash M_{\mathbf{T}} \precsim_{\mathbf{TS}} M_{\mathbf{S}} : \Gamma \qquad (F_{\mathrm{main}} : \mathsf{unit} \to \tau) \in \Gamma}{\mathrm{Behav}(M_{\mathbf{T}}) \subseteq \mathrm{Behav}(M_{\mathbf{S}})}$$

$\mathrm{Behav}(-)$ denotes the set of I/O and termination behaviors that a program can have. The theorem implies for instance that, if $M_{\mathbf{S}}$ always successfully terminates, then so does $M_{\mathbf{T}}$ and moreover they produce the same outputs.

If we have a compiler that respects our relation $\precsim_{\mathbf{TS}}$, then Theorem 1 gives us the same result as traditional whole-program compiler verification would. However, our relation also satisfies the following crucial property.

Theorem 2 (Preservation under linking, a.k.a. modularity)**.**

$$\frac{\Gamma \vdash M^1_{\mathbf{T}} \precsim_{\mathbf{TS}} M^1_{\mathbf{S}} : \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash M^2_{\mathbf{T}} \precsim_{\mathbf{TS}} M^2_{\mathbf{S}} : \Gamma_2}{\Gamma \vdash (M^1_{\mathbf{T}} \bowtie M^2_{\mathbf{T}}) \precsim_{\mathbf{TS}} (M^1_{\mathbf{S}} \bowtie M^2_{\mathbf{S}}) : \Gamma_1, \Gamma_2}$$

(Here, $\bowtie$ is overloaded notation for the linking operation both in the source and target language.) The theorem says that if we link two target modules, each of which is related to a source module, then the resulting target module is related to the linking of those source modules. Notice that, for the linking to make sense, the types of the first module's exported functions (in Γ_1) need to match the second module's assumptions. Of course, if a program consists of more than two modules, this theorem can be iterated as necessary and once linking results in a whole program, we can apply Theorem 1.

Observe that these properties don't mention any particular compiler but are stated in terms of arbitrary related modules. The missing link is a theorem saying that the desired compilers adhere to our relation. We prove this for *Pilsner* and *Zwickel*, our compilers from $\mathbf{S}$ to $\mathbf{T}$. Their correctness theorems apply to any well-typed source module:

Theorem 3 (Correctness of Pilsner)**.**

$$\frac{\Gamma \vdash M_{\mathbf{S}} : \Gamma'}{\Gamma \vdash \mathrm{Pilsner}(M_{\mathbf{S}}) \precsim_{\mathbf{TS}} M_{\mathbf{S}} : \Gamma'}$$

Theorem 4 (Correctness of Zwickel)**.**

$$\frac{\Gamma \vdash M_{\mathbf{S}} : \Gamma'}{\Gamma \vdash \mathrm{Zwickel}(M_{\mathbf{S}}) \precsim_{\mathbf{TS}} M_{\mathbf{S}} : \Gamma'}$$

While Zwickel carries out a straightforward direct translation from $\mathbf{S}$ to $\mathbf{T}$, Pilsner is more sophisticated: as shown in Figure 1, it compiles via an intermediate language $\mathbf{I}$ and performs several optimizations. We will discuss Pilsner (in detail) and Zwickel (briefly) in §7.

These results mean that we can preserve correctness not only by linking, say, Pilsner-produced code with other Pilsner-produced code, but also by linking it with code produced by Zwickel.

Moreover, we would like to stress two important points. PILS were designed with flexibility in mind and make only few assumptions about the translation of source programs, namely details of the calling convention and in-memory representation of values (see §5). Consequently:

1. Nothing stops us from proving a theorem analogous to the previous two for *yet another* compiler from $\mathbf{S}$ to $\mathbf{T}$, perhaps even using several different intermediate languages.
2. Nothing stops us from proving the relatedness of a source and target module *by hand*, *e.g.,* when the target module is not the direct result of a compiler run but was manually optimized (see §2.3 for an extreme example of this, where the target module overwrites its own code at run time).

Hence, we can also preserve correctness when linking with code that was produced by other compilers or even hand-translated. We only have to ensure that these translations are also correct w.r.t. $\precsim_{\mathbf{TS}}$, such that Theorem 2 applies.

2.2 Transitivity

Proving a property like Theorem 3 can require a lot of effort: the more complex the compiler, the more complex its correctness proof. It is thus crucial that a correctness proof can be broken up into several pieces, *e.g.,* one sub-proof per compiler pass. PILS support such a decomposition thanks to a transitivity-like property. In our setting, where Pilsner compiles via one intermediate language **I**, we can show the following:

Theorem 5 (Transitivity)**.**

$$\frac{\begin{array}{c}|\Gamma| \vdash M_{\mathbf{T}} \precsim_{\mathbf{TI}} M_{\mathbf{I}} : |\Gamma'| \quad |\Gamma| \vdash M_{\mathbf{I}} \precsim^{*}_{\mathbf{II}} M'_{\mathbf{I}} : |\Gamma'| \\ \Gamma \vdash M'_{\mathbf{I}} \precsim_{\mathbf{IS}} M_{\mathbf{S}} : \Gamma'\end{array}}{\Gamma \vdash M_{\mathbf{T}} \precsim_{\mathbf{TS}} M_{\mathbf{S}} : \Gamma'}$$

Here, $\precsim_{\mathbf{TI}}$ relates target modules to intermediate modules, $\precsim_{\mathbf{II}}$ relates intermediate modules to intermediate modules, and $\precsim_{\mathbf{IS}}$ relates intermediate modules to source modules. All are very similar to $\precsim_{\mathbf{TS}}$ and support similar reasoning principles. We will say more about them in §5; for now suffice it to say that, since $\precsim_{\mathbf{TI}}$ and $\precsim_{\mathbf{II}}$ involve only untyped languages,[2] the relations themselves are "untyped" and we erase the typing annotations in their environments (*e.g.,* written $|\Gamma|$), leaving just a list of function labels. Notice how using the transitive closure of $\precsim_{\mathbf{II}}$ in the second premise of the rule allows us to verify each IL transformation separately.

2.3 Flexibility

As already mentioned above, PILS make few assumptions about details of a translation and, as such, can be used not only to verify multiple different compilers with the same source and target languages (*e.g.,* Pilsner and Zwickel), but also to account for linking with hand-optimized low-level code. To substantiate this claim, we have proven[3] the challenging refinement from Hur and Dreyer [8] mentioned in §1, which relies on tricky manipulations of local state, far more involved than those of any imaginable compiler.

This example is based on Pitts and Stark's "awkward" example [20], which is easy to explain:

$$\begin{array}{l} e_a := \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda f.\ (x := 1; f\,\langle\rangle; !x) \\ e_b := \lambda f.\ (f\,\langle\rangle; 1) \end{array}$$

Both expressions evaluate to higher-order functions that, when applied, call the argument "callback" function f and then return a number. In e_b this number is simply 1. In e_a it is the result of dereferencing a local (private) reference x, which is initialized to 0. Notice, though, that when e_a is called for the first time, it immediately writes 1 to x. Since there are no other writes, the value of x returned at the end will always be 1 as well. As a result, e_a and e_b are equivalent programs (see the next section for a proof sketch).

Hur and Dreyer [8] adapt this example by substituting for e_b a tricky self-modifying machine program that implements the same behavior, but in a rather baroque way. Figure 2 shows what this program looks like in memory. It is parameterized by the load address n and $\mathbb{E}(-)$ denotes the encoding of an instruction as a

[2] In order to demonstrate that PILS are not inherently tied to typed languages, we consider a type-erasing compiler, not a completely type-directed one.

[3] After slightly modifying the machine code to account for a difference in calling convention.

```
n+0    𝔼(ld arg 1)                    | n+12   666 + 𝔼(ld aux 𝔼(jmp n+15))
       𝔼(alloc arg arg)               |        666 + 𝔼(ld i n+5)
       𝔼(ld aux n+5)                  |        666 + 𝔼(sto [i + 0] aux)
       𝔼(sto [arg + 0] aux)           |        666 + 𝔼(sto ⟨sp + 0⟩ ret)
       𝔼(jmp ret)                     |        666 + 𝔼(bop + sp sp 1)
n+5    𝔼(ld i n+11)                   | n+17   666 + 𝔼(ld ret n+20)
       𝔼(ld aux [i + 0])              |        666 + 𝔼(sto clo arg)
       𝔼(bop − aux aux 666)           |        666 + 𝔼(jmp [clo + 0])
       𝔼(sto [i + 0] aux)             |        666 + 𝔼(bop − sp sp 1)
       𝔼(bop + i i 1)                 |        666 + 𝔼(ld ret ⟨sp + 0⟩)
n+10   𝔼(bop − aux n+24 i)            | n+22   666 + 𝔼(ld arg 1)
       666 + 𝔼(jnz aux n+6)           |        666 + 𝔼(jmp ret)
```

Figure 2. Self-modifying "awkward" example

machine word. Notice that part of the code has been "encrypted" by adding 666 to its encoding. We briefly explain how the code works.

The first few lines allocate a new function closure with empty environment and code pointer $n + 5$, and return it to the context. When this function gets called the first time, it starts out by decrypting the encrypted instructions (offsets 5–11), thus replacing the encrypted code in memory. Subsequently (offsets 12–14), it replaces its first instruction by a direct jump in order to skip over the decryption loop in future executions. The remaining code (offsets 15–23), which is also the target of that jump, simply performs the callback function call and then returns 1.

Hur and Dreyer showed that this contrived implementation refines the high-level program e_a as a demonstration that their KLR approach is flexible enough to reason about semantically involved "transformations", even ones whose correctness relies on low-level internal state changes that clearly have no high-level counterpart. By verifying the same example (with respect to $\precsim_{\mathbf{TS}}$), we aim to demonstrate that PILS are equally flexible. In fact, the high-level structure of our proof closely follows that of Hur and Dreyer's proof because, as we explain in the next section, PILS and KLRs have a lot in common.

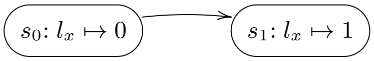
3. Background

PILS are essentially an inter-language generalization of the earlier work on parametric bisimulations (PBs) [9], which in turn emerged from prior work on KLRs [3, 6] in an attempt to overcome its limitations concerning transitivity. Both PBs and KLRs support the same high-level reasoning principles for higher-order imperative programs; they just do so in technically different ways. In this section, we give a bit of background on KLRs and PBs, and the similarities and differences between them.

What PBs and KLRs have in common: Protocols. One of the key features shared by PBs and recent work on KLRs is the ability to impose *protocols* on the local state of some functions, which describe how that local state is permitted to evolve over time. This idea was articulated most clearly by Dreyer *et al.* [3, 6], who proposed the use of various forms of *state transition systems (STSs)* to model these protocols and applied them to the verification of many challenging contextual equivalences.

The basic idea of protocols—and how intuitively one reasons about them—is easiest to explain by appeal to the simple "awkward" example presented in the previous section. We thus begin by walking through a proof of this example, using STS protocols, at a very informal level. At such a high level of abstraction, there is really no difference between a proof of the example using PBs vs. KLRs.

Step 1: Recall that we wish to prove e_a equivalent to e_b. By symbolically executing e_a, we see that it allocates a fresh heap location—call it l_x—which gets bound to x. Since this location is fresh, and since it is kept private by e_a, we can impose a *protocol* on it, dictating how its contents may evolve. We choose the following protocol, initially in state s_0:



This protocol expresses that the value at l_x currently is 0 (which indeed it is), but that it may eventually change to 1, after which point it will stay 1 forever.

Remark. This STS is very simple: besides consisting of only two states and a single transition, it also refers to only one of the two memories, namely that of e_a. In general, an STS may refer to both memories and even relate them to each other.

Now that we have installed this protocol on l_x, the proof reduces to showing that the function values v_a and v_b returned by e_a and e_b "behave equivalently" (and, in so doing, respect the protocol).

$$v_a := \lambda f.\ (l_x := 1; f\,\langle\rangle; !l_x)$$
$$v_b := \lambda f.\ (f\,\langle\rangle; 1)$$

(Note that v_b is just e_b, since e_b was already a value.)

So what does it mean to "behave equivalently"? This is really the big question, for which KLRs and PBs give different answers. Rather than try to answer it directly, we will instead describe two informal proof principles concerning behavioral equivalence that, at the level of abstraction we are working at here, are supported by both proof methods, and we will finish the proof sketch by just appealing to these proof principles. After that, we will explain how the different proof methods implement these principles.

Principle 1 (Showing Behavioral Equivalence): To show that v_a and v_b **behave** equivalently, it suffices to show that they **behave** equivalently when applied to any arguments f_a and f_b passed in from the environment, which we may **assume** are equivalent.

Principle 2 (Using Assumed Equivalence): If f_a and f_b are **assumed** equivalent, then $f_a\,\langle\rangle$ and $f_b\,\langle\rangle$ **behave** equivalently.

Note that these proof principles make a distinction between when two functions *behave* equivalently and when they are *assumed* equivalent. We explain the difference between these notions below. Note also that we restricted Principle 2 here to functions with unit argument; this is merely to simplify our informal discussion.

Step 2: By Principle 1, suppose that we are given f_a and f_b, passed in from the environment, that are *assumed* equivalent; it remains to show that $v_a\ f_a$ and $v_b\ f_b$ *behave* equivalently. When the execution of these functions begins, the state of l_x's protocol could be either in s_0 or s_1, since the functions could be called at some point in the future. In either case, $v_a\ f_a$ first sets l_x to 1, thereby either updating the state of the protocol to s_1 (if it was in s_0 to start) or leaving it as is—either way, a legal transition.

Step 3: We are now trying to show that $(f_a\,\langle\rangle; !l_x)$ and $(f_b\,\langle\rangle; 1)$ behave equivalently, given that the protocol is now in state s_1. Since f_a and f_b were *assumed* equivalent, we know by Principle 2 that $f_a\,\langle\rangle$ and $f_b\,\langle\rangle$ behave equivalently. The result therefore reduces to showing that $!l_x$ and 1 behave equivalently. We may assume that these expressions are executed starting in state s_1, because that is the only state accessible from the state s_1 we were in previously (*i.e.,* we assume the protocol has been respected by f_a and f_b).

Step 4: Since we know we are still in state s_1, we know that $!l_x$ evaluates to 1. Thus, $!l_x$ and 1 behave equivalently, so we are done.

Logical relations. Both KLRs and PBs allow one to turn the above proof sketch into a proper proof. The key difference between them is how they formalize behavioral vs. assumed equivalence.

KLRs formalize this by defining a relation, which says—once and for all—what it means for two expressions to be indistinguishable at a certain type. One then uses this *same* "logical" relation as the definition of *both* behavioral and assumed equivalence. For expressions, the logical relation says that they are equivalent if they either both run forever or they both evaluate to equivalent values.

For function values, which both the v's and the f's in our example are, the logical relation says that they are equivalent if they map logically-related arguments to logically-related results. Principles 1 and 2 both fall out of this definition as immediate consequences.

The main difficulty with logical relations is that, by conflating behavioral and assumed equivalence, they introduce an inherent circularity in the construction of the logical relation. In particular, the definition of equivalence of function values refers recursively to itself in a negative position (when quantifying over equivalent arguments). Traditionally, for simpler languages (*e.g.,* without recursive types or higher-order state), this circularity is handled by defining the logical relation by induction on the type structure. For richer languages, such as our source language $\mathbf{S}$, induction on types is no longer sufficient, but step-indexing can be used instead to stratify the construction by the number of steps of computation in the programs being related [3]. This is the approach taken by Hur and Dreyer [9] in their earlier work on compositional compiler correctness. However, it is not known how to prove *transitivity* of logical relations for step-indexed models (at least in a way that is capable of scaling to handle inter-language reasoning, which we need for compiler verification).[4]

Parametric bisimulations. This problem with transitivity was one of the key motivations for *parametric bisimulations (PBs).* Unlike logical relations, PBs treat behavioral and assumed equivalence as distinct concepts. In particular, rather than trying to *define* assumed equivalence, PBs take assumed equivalence as a *parameter* of the model (hence the name "parametric bisimulations"). That is, a PB proof that two expressions are behaviorally equivalent is parameterized by an arbitrary *unknown relation U* representing assumed equivalence,[5] and U could relate *any* functions f_a and f_b!

To make use of this unknown U parameter, PBs update the definition of behavioral equivalence accordingly. For function values, one can show them behaviorally equivalent precisely as suggested by Principle 1, *i.e.,* if they map U-related (assumed equivalent) arguments to behaviorally equivalent results. For expressions, behavioral equivalence extends the definition from logical relations with a new possibility, namely that the expressions may call functions f_a and f_b related by U. This amounts to baking Principle 2 directly into the definition of behavioral equivalence. The reason this is necessary—*i.e.,* the reason Principle 2 does not just fall out of the definition otherwise—is that U is a *parameter* of behavioral equivalence. Knowing that f_a and f_b are assumed equivalent according to U tells us absolutely nothing about them! Consequently, Principle 2 must be explicitly added to the definition of behavioral equivalence as an extra case (called the "external call" case, as it concerns calls to "external" functions passed in from the environment).

One can understand PBs as defining a "local" notion of behavioral equivalence: two expressions are behaviorally equivalent if they behave the same in their local computations, *ignoring* what happens during calls to (U-related) external functions passed in from the environment. Intuitively, this is perfectly sound: it just means each module in a program is responsible for its own local computations, not the local computations of other modules. Moreover, as we have observed already, it is largely a technical detail: PBs can support the same high-level protocol-based reasoning as KLRs do.[6]

The major benefit of PBs over KLRs is that they avoid the problems with the circularity of KLRs which necessitated step-indexing. In particular, note that by taking Principle 1 as the *definition* of behavioral equivalence for function values, we avoid the negative self-reference that plagues logical relations: the arguments f_a and f_b are simply drawn from the unknown relation U. This avoidance of step-indexing was essential in making it possible to establish that PBs do in fact support transitivity, but it does not mean that the proof of transitivity was easy. The interested reader can find further details in an earlier technical report [10].

4. PILS, Part 1: Languages

PILS generalize PBs to the inter-language setting of compiler verification. Recall from §2 that we are interested in compiling from an ML-like source language $\mathbf{S}$ to a machine language $\mathbf{T}$, and that the more complex of our two compilers, Pilsner, employs an intermediate language $\mathbf{I}$. As argued before, besides the main similarity relation $\precsim_{\mathbf{TS}}$ between $\mathbf{T}$ and $\mathbf{S}$ modules, we also need PILS to provide a similarity relation for each pair of languages adjacent in Pilsner's compilation chain (Figure 1), *i.e.,* $\precsim_{\mathbf{TI}}$, $\precsim_{\mathbf{II}}$, and $\precsim_{\mathbf{IS}}$.

4.1 Language-Generic Approach

In order to avoid duplicate work, we define PILS in a language-generic way, *i.e.,* we define similarity $\precsim_{AB}$ for two abstract languages A and B (for some notion of abstract language to be described), and then instantiate it with different language pairs in order to obtain the desired relations. This has two important benefits:

1. Most of the metatheory, which is quite involved, can be established once and for all. This is particularly crucial because PILS were developed from the start in Coq, and over time the definitions—and thus proofs—had to undergo countless changes. This might simply have been infeasible if it weren't for the language-generic setup.
2. One can instantiate PILS with a different intermediate language (or several of them) in order to verify a different compiler. Using modularity (Theorem 2), one can then of course safely link $\mathbf{T}$ code produced by this compiler with Pilsner-produced and/or Zwickel-produced code.

In (1), we say "most of the metatheory" because transitivity and the parts of modularity and adequacy that deal with details of module loading are actually not proven generically. Ultimately, it would be nice to do so but it would require some effort to properly axiomatize various properties of the abstract language that these proofs rely on. Moreover, it might require a distinction between *intermediate language* and *non-intermediate language*, with slightly different sets of requirements. For now, it is much easier to simply prove the theorems for the concrete instances (of course this involves many generically proven lemmas). The downside of this is that, in order to verify a compiler using different intermediate languages, one needs to reprove the corresponding transitivity property. Adequacy and modularity, on the other hand, do not need to be reproven as they do not involve the intermediate languages.

To instantiate the generic PILS model and obtain one of the desired similarity relations requires us to provide: (i) the pair of concrete languages, and (ii) the *global world* for this pair. The latter can be seen as a predefined protocol (in the sense of §3) responsible for fixing calling conventions and data representations. We will discuss global worlds further in §5.

One point that we glossed over so far is that our generic definition is also not entirely generic—as we will see in the next section, it still essentially bakes in our source language's type structure. Consequently, instantiating PILS as-is with a different source language

[4] Transitivity for logical relations *can* be achieved via methods such as $\top\top$-closure [6], or by restricting the relations to well-typed terms [2], but such approaches depend on the relations relating terms in the same language.

[5] Hur *et al.* [9] called this the "global knowledge", in contrast to the "local knowledge". We feel our terminology is more intuitive, and does not conflict with the global/local distinction as it pertains to worlds (§5.1).

[6] There is one exception: PBs do not admit the eta law for function values. This is a known problem [9], with a known solution [11], but we leave it as future work to incorporate this solution into PILS.

Domains: $\mathbf{Val}, \mathbf{Cont}, \mathbf{Conf}, \mathbf{Mach}, \mathbf{Mod}, \mathbf{Anch}$

Operators and relations:

- $\mathrm{cload} \in \mathbf{Mod} \to \mathbf{Anch} \to (\mathrm{Lbl} \times \mathbf{Val})^* \to \mathcal{P}(\mathbf{Conf})$
- $\mathrm{vload} \in \mathbf{Mod} \to \mathbf{Anch} \to (\mathrm{Lbl} \times \mathbf{Val})^* \to \mathrm{Lbl} \to \mathcal{P}(\mathbf{Val})$
- $\cdot \in \mathbf{Conf} \to \mathbf{Conf} \to \mathbf{Conf}$ (commutative and associative)
- $\emptyset \in \mathbf{Conf}$ (neutral for $\cdot$)
- $\hookrightarrow \in \mathcal{P}(\mathrm{Evt} \times \mathbf{Mach} \times \mathbf{Mach})$
- $\mathrm{real} \in \mathbf{Conf} \to \mathcal{P}(\mathbf{Mach})$
- $\mathrm{error} := \{m \in \mathbf{Mach} \mid \forall c.\, m \notin \mathrm{real}(c)\}$

Figure 3. Language specification

most likely won't make much sense. This is fine, because we focus on a single source language in this work. Extending this to multiple source languages, perhaps even allowing interoperability between them, is clearly important but left to future work.

Another point we glossed over is that we actually define *two* generic models: a typed one and an untyped one. The former is used when the input language is $\mathbf{S}$, the latter is used in all other cases (where no involved language has static types). However, we will continue to refer to them as just "the generic model", because the untyped one is obtained simply by erasing all the type arguments from the typed one (highlighted in brown in the figures in §5).

4.2 Language Specification

We now describe the language abstraction in terms of which PILS are defined. In the subsequent sections, we then briefly present the concrete languages under consideration ($\mathbf{S}$, $\mathbf{I}$, $\mathbf{T}$). Common to all languages are a set of events and a countably infinite set of labels:

$$t \in \mathrm{Evt} ::= \epsilon \mid ?n \mid !n \qquad F_1, F_2, \ldots \in \mathrm{Lbl}$$

Events are produced by an executing program; they consist of internal computation (ϵ) and I/O operations (reading or writing a number n, respectively). Labels are used to identify module components; in this work, we consider a simplistic notion of module as the unit of compilation.

Figure 3 presents the abstract language in terms of a signature that any concrete language must implement. Keep in mind that we need to account for a very high-level language ($\mathbf{S}$) on the one extreme and a very low-level language ($\mathbf{T}$) on the other extreme.

A language must come with a set $\mathbf{Val}$ of *values*, a set $\mathbf{Cont}$ of *continuations*, a set $\mathbf{Conf}$ of *configurations*, a set $\mathbf{Mach}$ of *machines*, a set $\mathbf{Mod}$ of *modules*, and a set $\mathbf{Anch}$ of *anchors* (think: load addresses). The core of the operational semantics is given in the form of a transition system ($\hookrightarrow$) of machines, whose transitions are labelled with events t. Configurations can be thought of as partial machines—they play different roles in different contexts (*e.g.,* they might represent just a heap or just an expression, or even a full machine). If a configuration c is complete, it is *realized* by a set of machines $\mathrm{real}(c)$. (In all our instantiations, this is either empty, meaning the configuration is invalid or incomplete, or it contains exactly one machine.) Configurations must form a partial commutative monoid with composition $\cdot$ and neutral element $\emptyset$, except that the partiality is implicit via real. (Having $\cdot$ be total is more convenient for mechanization [18]). We say a machine denotes an *error*, $m \in \mathrm{error}$, iff it does not realize any configuration.

Intuitively, modules are sets of labelled function values (the *exports*), which may refer to external functions (the *imports*) by their unique labels. In the language specification, the module interface consists of two operations, cload and vload. The former, cload, takes an anchor saying "where" the module is to be loaded and values for each of its imports. It then returns a set of initial configurations in which the module is considered loaded. Given the same inputs and additionally the label of one of the exported functions, vload returns the module's corresponding function value.

$$\begin{aligned}
\tau &::= \alpha \mid \mathsf{unit} \mid \mathsf{nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\,\tau \mid \\
&\quad\ \forall\alpha.\,\tau \mid \exists\alpha.\,\tau \mid \mathsf{ref}\,\tau \\
e &::= x \mid F \mid \langle\rangle \mid n \mid e_1 \circ e_2 \mid \mathsf{ifnz}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \langle e_1, e_2\rangle \mid \\
&\quad\ e.1 \mid e.2 \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\,e\,(x.\,e_1)\,(x.\,e_2) \mid \mathsf{roll}\ e \mid \mathsf{unroll}\ e \mid \\
&\quad\ \mathsf{fix}\,f(x).\,e \mid e_1\ e_2 \mid \Lambda.\,e \mid e[] \mid \mathsf{pack}\ e \mid \mathsf{unpack}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 \mid \\
&\quad\ l \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid e_1 == e_2 \mid \mathsf{input} \mid \mathsf{output}\ e \\
v &::= \langle\rangle \mid n \mid \langle v_1, v_2\rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \mathsf{roll}\ v \mid \mathsf{fix}\,f(x).e \mid \Lambda.e \mid \mathsf{pack}\ v \mid l
\end{aligned}$$

$$\begin{aligned}
&\mathbf{Val} && := \{v \mid \mathrm{FV}(v) = \emptyset\} \\
&\mathbf{Cont} \ni K && ::= \bullet \mid K\ e \mid v\ K \mid \ \ldots \\
&\mathbf{Mod} \ni M && ::= [F_1{=}e_1, ..., F_n{=}e_n] && \mathbf{Anch} := 1 \\
&\mathrm{Env} && := \mathrm{Lbl} \rightharpoonup \mathbf{Val} && \mathrm{Heap} := (\mathrm{Loc} \rightharpoonup \mathbf{Val})_\bot \\
&\mathbf{Mach} && := \mathrm{Heap} \times \mathrm{Env} \times \mathrm{Exp} \\
&\mathbf{Conf} && := \mathrm{Heap} \times \mathrm{Env}_{\bot,\emptyset} \times \mathrm{Exp}_{\bot,\emptyset}
\end{aligned}$$

$$\begin{aligned}
(h, \sigma, K[\mathsf{input}]) &\overset{?n}{\hookrightarrow} (h, \sigma, K[n]) \\
(h, \sigma, K[F]) &\hookrightarrow (h, \sigma, K[v]) && (\text{if } \sigma(F) = v) \\
(h, \sigma, K[\mathsf{ref}\ v]) &\hookrightarrow (h \cdot \{l \mapsto v\}, \sigma, K[l]) && (\text{if } h \cdot \{l \mapsto v\} \neq \bot) \\
&\ \ldots \\
(h, \sigma, e) &\hookrightarrow (\bot, \sigma, e) \quad (\text{if } e \neq v \text{ and no other rule applicable})
\end{aligned}$$

Figure 4. Source language $\mathbf{S}$

4.3 Source Language S

The source language $\mathbf{S}$ is a standard PCF-like language extended with products, sums, universals, existentials, general recursive types, general reference types, and numeric I/O. Its type and term syntax is given in Figure 4.

Continuations are the evaluation contexts K in terms of which the language semantics is defined. Machines consist of a heap h, a read-only environment σ for resolving labels to values, and an expression e. Heaps are either undefined ($\bot$) or partial maps from locations to values. We assume the obvious composition operation $\cdot$ for heaps (overloading notation) that returns the union of two heaps iff both are defined and they don't overlap (otherwise it returns $\bot$). Note that the empty heap $\emptyset$ is its neutral element. The step relation between machines is a pretty standard substitution-based left-to-right call-by-value reduction and we state only a few rules. If a machine cannot take a successful step (according to the usual rules such as beta reduction), then it steps to an error state by invalidating its heap component.

Configurations are machines where one or more components may be missing or invalid. For a machine m to realize a configuration c, it must match the configuration ($m = c$ with the obvious embedding of $\mathbf{Mach}$ in $\mathbf{Conf}$). Moreover, it must carry a valid and finite heap (finiteness guarantees that allocation will succeed). We define $\mathbf{Conf}$ and their composition operation ($\cdot$) such that heaps can successfully be split across several configurations, but environments and expressions cannot—they must be defined in exactly one component in order for the composition to be realizable.

We assume a standard typing judgment $\Gamma \vdash e : \tau$, where Γ assigns types to both labels and variables. It implies that τ and the types in Γ are closed and that all labels and free variables in e are in the domain of Γ.

A module M is simply an ordered list of uniquely labelled function definitions. The above typing judgment is lifted to modules as $\Gamma \vdash M : \Gamma'$, but requires that both Γ and Γ' only contain labels and, for simplicity, that the module components are all functions ($\mathsf{fix}\,f(x).\,e$ or $\Lambda.\,e$). Moreover it imposes a strict left-to-right dependency order on the module components.[7] As there is no need for anchors in the source language, we define them as a singleton set.

Linking two modules ($\bowtie$) simply concatenates them (assuming their labels are disjoint). Note that this is an asymmetric operation

[7] This is merely to keep the module semantics simple. PILS themselves, being a coinductive method, are perfectly compatible with mutual recursion.

$$
\begin{array}{l}
a ::= \langle\rangle \mid n \mid \langle x_1, x_2\rangle \mid x.1 \mid x.2 \mid \mathsf{inl}\ x \mid \mathsf{inr}\ x \mid \\
\qquad \mathsf{fix}\, f(y,k).\, e \mid \Lambda k.\, e \mid x_1 == x_2 \mid x_1 \circ x_2 \\
e ::= \mathsf{let}\ y = a\ \mathsf{in}\ e \mid \mathsf{let}\ k\ y = e_1\ \mathsf{in}\ e_2 \mid y \leftarrow \mathsf{input};\ e \mid \mathsf{output}\ x;\ e \mid \\
\qquad y \leftarrow \mathsf{ref}\ x;\ e \mid x_1 := x_2;\ e \mid y \leftarrow !x;\ e \mid \mathsf{ifnz}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \\
\qquad \mathsf{case}\, x\ (y.\, e_1)\ (y.\, e_2) \mid x_1\ x_2\ k \mid x[]\ k \mid k\ x
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{Val} \ni v & ::= & \langle\rangle \mid n \mid l \mid \langle v_1, v_2\rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \\
 & & \langle\sigma, \mathsf{fix}\, f(y,k).e\rangle \mid \langle\sigma, \lambda y.e\rangle \\
\mathbf{Cont} & := & \mathbf{Val} \\
\mathrm{Env} & := & \mathrm{Lbl} \uplus \mathrm{TVar} \uplus \mathrm{KVar} \rightharpoonup \mathbf{Val} \\
\mathbf{Mach} & := & \mathrm{Heap} \times (\mathrm{Env} \times \mathrm{Exp})
\end{array}
$$

$$
\begin{array}{rl}
(h, (\sigma, \mathsf{let}\ k\ y = e_1\ \mathsf{in}\ e_2)) \hookrightarrow (h, (\sigma[k \mapsto \langle\sigma, \lambda y.e_1\rangle], e_2)) & \\
(h, (\sigma, k\ x)) \hookrightarrow (h, (\sigma'[y \mapsto \sigma(x)], e)) & \\
 & (\text{if } \sigma(k) = \langle\sigma', \lambda y.e\rangle) \\
\dots & \\
(h, (\sigma, e)) \hookrightarrow (\bot, (\sigma, e)) & (\text{if no other rule applicable})
\end{array}
$$

Figure 5. Intermediate language **I**

as it may resolve imports of the right module, but not of the left. The semantics of a program, *i.e.,* a complete module containing a designated main function (F_{main}) of type $\mathsf{unit} \to \tau$, is the semantics of the machine consisting of an empty heap, the module itself as environment, and the call of the main function as the expression component.

4.4 Intermediate Language I

I is an untyped, or rather, dynamically typed CPS-variant of **S**, inspired by Kennedy's intermediate language [12]. Parts of it are shown in Figure 5.

In contrast to the source language, **I** is defined using an environment-based semantics where continuations and functions evaluate to closures of code and environment. This avoids the need to reason about substitutions when verifying optimizations, which is often a hassle.

Being in continuation-passing style, every subexpression is explicitly named and functions never "return". Concretely, we distinguish between (i) *pure expressions* a, which are evaluated in let-bindings and always yield a value without any side-effects, and (ii) *control expressions* e. Ignoring conditionals, every control expression is essentially a sequence of bindings ending in a function or continuation call. For instance, $\mathsf{let}\ k\ y = e_1\ \mathsf{in}\ e_2$ defines a new continuation k with argument y and body e_1, and then executes e_2 (which may use k). Here $y \in \mathrm{TVar}$ is term variable, while $k \in \mathrm{KVar}$ is a continuation variable. Any x in the language syntax stands for either a term variable or a label.

Modules, anchors, configurations, etc. are similar to those in the source language. We define **Cont** simply as **Val** because continuations are already values in the language.

4.5 Target Language T

As shown in Figure 6, our target language **T** is an idealized assembly language featuring instructions for arithmetic, control flow, memory access, and I/O. Some of them support multiple addressing modes. For instance, if $o = \langle r_1 \pm n\rangle$, then $\mathsf{sto}\ o\ r_2$ stores the contents of register r_2 on the stack at the address contained in register r_2, offset by $\pm n$. If $o = [r_1 \pm n]$, then it stores it on the heap instead. The lpc instruction loads the current program counter into the given register.

T is idealized for instance in the sense that machine words are unbounded natural numbers and stack and heap are unbounded as well. The set of registers, though, is fixed (their names, by the way, are merely suggestive and do not matter for the language semantics). Moreover, code is encoded as data and can be modified. We assume a deterministic (but otherwise arbitrary) memory allocator.

Machines consist of heap, stack, register file, and current program counter (pointing to the heap). We omit the transition rules for brevity, as they are straightforward. The configuration monoid is defined analogously to the previous languages (heap and stack can be split, the rest cannot).

$$
\begin{array}{lcl}
\mathrm{Reg} \ni r & ::= & \mathsf{sp} \mid \mathsf{clo} \mid \mathsf{arg} \mid \mathsf{env} \mid \mathsf{ret} \mid \mathsf{aux} \mid \mathsf{i} \\
\mathrm{Oper} \ni o & ::= & n \mid r \mid \langle r \pm n\rangle \mid [r \pm n] \\
\mathrm{Instr} \ni z & ::= & \mathsf{jmp}\ o \mid \mathsf{jnz}\ r\ o \mid \mathsf{ld}\ r\ o \mid \mathsf{sto}\ o\ r \mid \mathsf{lpc}\ r \mid \\
 & & \mathsf{bop}\ \circ\ r\ o_1\ o_2 \mid \mathsf{input}\ r \mid \mathsf{output}\ r \mid \mathsf{alloc}\ r_1\ r_2 \\
 & & \\
\mathbf{Val} & := & \mathrm{Word} \\
\mathbf{Anch} & := & \mathrm{Word} \\
\mathbf{Cont} & := & \mathrm{Word} \\
\mathrm{RegFile} & := & \mathrm{Reg} \to \mathrm{Word} \\
\mathrm{Stack} & := & (\mathrm{Word} \rightharpoonup \mathrm{Word})_\bot \\
\mathrm{Heap} & := & (\mathrm{Word} \rightharpoonup \mathrm{Word})_\bot \\
\mathbf{Mach} & := & \mathrm{Heap} \times \mathrm{Stack} \times \mathrm{RegFile} \times \mathrm{Word}
\end{array}
$$

$$
\mathrm{real}(c) := \{m \mid m = c \wedge c.\mathrm{hp} \neq \bot \wedge c.\mathrm{hp}\ \text{finite} \wedge c.\mathrm{st} \neq \bot\}
$$

Figure 6. Target language **T**

In order to abstract away the distracting details of relocation, we model target modules as (meta-level) functions. A module thus takes a load address (the role of anchors in **T**) and imports (a value for each imported label), and returns a data segment from which one can obtain the exported values as well as an initial heap (containing all the code) in which they make sense. Linking two modules essentially just concatenates their data segments.

Finally, we dictate the following contract ("calling convention") for calls to module-level functions, both intra- and inter-module: To call a function, (1) write its value into register clo, (2) write its argument into register arg, (3) write its return address into register ret, and (4) jump to $[\mathsf{clo}{+}0]$, *i.e.,* to wherever the value in the heap at address clo points to. If and when control eventually reaches the return address, (5) the function's result must reside in register arg. Moreover, (6) registers env and sp must have been preserved (*i.e.,* these are callee-save registers while the rest are caller-save), and the stack must have been preserved as well.

5. PILS, Part 2: Worlds and Similarity

5.1 Worlds

Worlds are the formalism used by KLRs and PILS to incorporate the idea of protocols described in §3. Their shape is shown in Figure 7. The account of worlds and PILS we present here is somewhat simplified for the sake of presentation. For example, we omit an important distinction between *public* and *private* transitions in protocols, because it is inherited directly from the prior work on KLRs [8] and PBs [9]. At various points, we will discuss several ways in which our actual model (verified in Coq) is more sophisticated. Full details are given in the technical appendix [1].

We distinguish between *global worlds* and *local worlds*. Ultimately we relate programs under a *full world*, which is the composition of a global and a local one. The global world is a parameter of the definitions and needs to be instantiated together with the language implementations—we define exactly one global world for each language pair of interest ($W_{\mathbf{TI}}$, $W_{\mathbf{II}}$, $W_{\mathbf{IS}}$, and $W_{\mathbf{TS}}$). Its task is to describe the calling convention and data representation that all modules have to follow. It also governs the global references, *i.e.,* values being passed around at reference type, and the memory that those point to. A local world, on the other hand, is what one gets to pick in the proof of module similarity. It can assert properties about the modules' *local* state, *e.g.,* as illustrated in §3.

Global and full worlds have the same structure. They consist of a *transition system* T, a *configuration relation* crel, and several methods for "querying" the global state (to be discussed in §5.2).

A transition system defines a protocol state space S with (inverted) transition relation $\sqsupseteq$, and the configuration relation defines the interpretation of the states as relational constraints on the two programs' memories. For instance, in the example from §3, S would

$$
\begin{aligned}
\mathrm{VRelF}_{A,B} &:= \mathrm{TypeF} \to \mathcal{P}(A.\mathbf{Val} \times B.\mathbf{Val})\\
\mathrm{VRel}_{A,B} &:= \mathrm{Type} \to \mathcal{P}(A.\mathbf{Val} \times B.\mathbf{Val})\\[4pt]
T \in \mathrm{TrSys} &:= \{(\mathsf{S}, \sqsupseteq) \in \mathrm{Set} \times \mathcal{P}(\mathsf{S} \times \mathsf{S}) \mid \sqsupseteq \text{ is preorder}\}\\[4pt]
\mathrm{QH}^{T}_{A,B} &:= \{(\mathsf{vqha} \in T.\mathsf{S} \xrightarrow{\mathrm{mon}} \mathsf{VQry}_A \to \mathcal{P}(A.\mathbf{Val})\\
&\qquad ,\mathsf{vqhb} \in T.\mathsf{S} \xrightarrow{\mathrm{mon}} \mathsf{VQry}_B \to \mathcal{P}(B.\mathbf{Val})\\
&\qquad ,\mathsf{cqha} \in T.\mathsf{S} \to \mathsf{CQry}_A \to \mathcal{P}(A.\mathbf{Conf})\\
&\qquad ,\mathsf{cqhb} \in T.\mathsf{S} \to \mathsf{CQry}_B \to \mathcal{P}(B.\mathbf{Conf})\\
&\qquad ,\mathsf{rqh} \in T.\mathsf{S} \xrightarrow{\mathrm{mon}} \mathrm{VRel}_{A,B})\}\\
\mathrm{CR}^{T}_{A,B} &:= \{\mathsf{crel} \in (T.\mathsf{S} \to \mathrm{VRelF}_{A,B}) \xrightarrow{\mathrm{mon}}\\
&\qquad\qquad T.\mathsf{S} \to \mathcal{P}(A.\mathbf{Conf} \times B.\mathbf{Conf})\}\\[4pt]
\mathrm{World}_{A,B} &:= \{(T \in \mathrm{TrSys}, \_ \in \mathrm{CR}^{T}_{A,B}, \_ \in \mathrm{QH}^{T}_{A,B})\}\\
\mathrm{WorldL}_{A,B} &:= \{(T \in \mathrm{TrSys}, \_ \in \mathrm{CR}^{T}_{A,B})\}
\end{aligned}
$$

Figure 7. Worlds (simplified)

be a two-element set $\{s_0, s_1\}$, $\sqsupseteq$ would be $\{(s_1, s_0)\}^*$, and crel would map state s_n to a singleton heap that stores n at location l_x. The fact that crel takes the unknown relation U as argument matters when the protocol involves higher-order state (see [9]).

Local worlds are like full worlds except that they don't contain the global query handlers. Combining a local world $w \in \mathrm{WorldL}_{A,B}$ with a global world W works straightforwardly by taking the product of their transition systems, the separating conjunction of their configuration relations, and passing all state queries on to W.

5.2 Similarity

At the top-level, PILS define module similarity, which we instantiate to obtain $\precsim_{\mathbf{TS}}$ and so on. The main ingredient of this is $\mathbf{E}$, the coinductively defined *behavioral* similarity for "expressions", which is parameterized over the unknown relation U representing *assumed* similarity (as discussed in §3). In our generic setting there is no notion of expressions, but we find it helpful to refer to the configurations related by $\mathbf{E}$ as such. Indeed, for the source language $\mathbf{S}$ these configurations will usually just be expressions (*i.e.,* heap and environment are missing). For $\mathbf{T}$, on the other hand, they will usually just be program counters (*i.e.,* heap, stack, and register file are missing). The missing parts will be provided by the world. In particular, in $\mathbf{T}$, the global part of the world always owns the register file, because it is a global resource accessible by any module. Concretely this means that the state of the two global worlds involving $\mathbf{T}$ (for $\precsim_{\mathbf{TS}}$ and for $\precsim_{\mathbf{TI}}$) contains the current register file R and their crel only allows machine configurations whose register file is precisely R.

Let us now analyze the simplified presentation of $\mathbf{E}$ in Figure 8, first at a high level and then in more detail. Very roughly, it relates two programs e_a (the "target" of a transformation) and e_b (the "source" of a transformation) iff one of three cases holds:

(ERR) e_b can silently (*i.e.,* without I/O) produce an error.

(RET) e_a is finished, and e_b can silently finish returning a related value.

(STEP) e_a can take a step and e_b can match it (perhaps after some internal computation). "Match" means that both steps produce the same event and that the remaining computations either (REC) are again related by $\mathbf{E}$, or (CALL) are about to call related "external" functions, *i.e.,* functions related by the unknown relation U. This "external call" case is the characteristic feature of PBs, as discussed in §3. ($\mathbf{K}$, not shown here, relates continuations and is defined in terms of $\mathbf{E}$.)

In the explanation above, we glossed over many details. For a start, $\mathbf{E}$ is indexed implicitly by a full world W and explicitly by several parameters including the unknown relation U. We now take a closer look at the definition and discuss the key parameters.

Asymmetric small-step formulation. In contrast to the symmetric big-step formulation of $\mathbf{E}$ in Hur *et al.* [9], PILS employ an asymmetric small-step formulation. Notice how $\mathbf{E}$'s STEP case asks us to consider each possible step of the "target" program e_a in turn (using REC repeatedly), each time demanding us to match it with several steps of the "source" program e_b. Besides being seemingly necessary to properly deal with events (here: I/O), such an asymmetric small-step formulation is also important in the context of compiler verification because it gives the compiler the flexibility to remove erroneous behaviors of the source program and resolve some of its nondeterminism.

In this simplified presentation of $\mathbf{E}$, e_b is forced to take at least as many steps as e_a. Of course, this is overly restrictive and the actual definition relaxes this—more or less in the usual way (allowing *stuttering*), but with perhaps unusual implications. We discuss this in §6.1.

Protocol conformance. In order to talk about the execution of e_a and e_b, we first need to "complete" these configurations and convert them into physical machines. These completions should not be completely arbitrary; they should adhere to the world's constraints at the current state s. Hence we quantify over c_a and c_b, representing the portion of the machine state constrained by the world W, and require—in the helper definition configure—that they are indeed related by $W.\mathsf{crel}(U)(s)$. We then attach these to e_a and e_b, together with arbitrary frame configurations η_a and η_b representing the rest of the running program state. Finally, we only consider machines m_a and m_b that realize these composed configurations.

The two other occurrences of configure (in STEP and RET) are proof obligations. For instance, STEP requires us to show that, after the step, each resulting machine can again be decomposed into a (possibly new) expression e'_a, a (possibly new) configuration c'_a, and the *original* frame configuration η_a (similarly for the b-side), since we should not have touched the frame's private state. Moreover, c'_a and c'_b must again satisfy W's constraints, but we may advance s to a future state s' in order to achieve that.

Configuration queries. The RET case has to assert (in a language-generic way) that the two computations have finished and returned similar values. In our source language, termination is a syntactic property and is trivial to check. But what does it mean for a $\mathbf{T}$ computation to be "finished"? We take it to mean that control has reached the original return address. In order to determine whether this is the case, we need to know the return address in the first place. This is why $\mathbf{E}$ takes as arguments initial continuations k^0_a and k^0_b.

The actual check is then performed with the help of the world. Its global part provides *configuration query handlers* cqha and cqhb that answer questions such as "in state s', does e'_a constitute a return of value v_a to continuation k^0_a?", written $e'_a \in W.\mathsf{cqha}(s')(\mathsf{ret}\, v_a\, k^0_a)$. For $\mathbf{T}$, this amounts to checking that the program counter of e'_a (typically its only component) equals k^0_a *and* that the return register contains v_a. The latter requires inspecting the given state s', because it contains the register file.

A different query, app, is used by the STEP/CALL case to test if both configurations represent function calls. In effect, this means that the global world's implementation of cqha and cqhb determines a major part of the calling conventions.

Value closure and value queries. Both the RET and STEP/CALL cases also require that certain values are related, *e.g.,* the returned results: $(v_a, v_b) \in \langle\!\langle U(s')\rangle\!\rangle^{s'}(\tau)$. But what is this relation exactly? As in PBs, U is intuitively only needed to relate unknown "external" functions passed in from the environment. The *value closure operation* $\langle\!\langle - \rangle\!\rangle$ lifts it to other value forms in a straightforward way,

$$
\begin{array}{l}
\mathbf{E} \in A.\mathbf{Cont} \times B.\mathbf{Cont} \to (W.\mathsf{S} \to \mathrm{VRelF}_{A,B}) \to W.\mathsf{S} \to \mathrm{Type} \to \mathcal{P}(A.\mathbf{Conf} \times B.\mathbf{Conf}) \\
\mathbf{E}(k_a^0, k_b^0)(U)(s)(\tau) = \{(e_a, e_b) \mid U \in \mathbf{U} \implies \forall c_a, c_b, \eta_a, \eta_b.\ \forall (m_a, m_b) \in \mathrm{configure}(U)(s)(c_a, c_b)(e_a \cdot \eta_a, e_b \cdot \eta_b). \\
\qquad\quad (\textsc{err})\ \ \exists m_b'.\ m_b \overset{\epsilon}{\hookrightarrow}{}^{*}\ m_b' \wedge m_b' \in B.\mathrm{error} \\
\qquad \vee\ (\textsc{ret})\ \ \exists m_b', s', v_a, v_b, e_a', e_b', c_a', c_b'.\ m_b \overset{\epsilon}{\hookrightarrow}{}^{*}\ m_b' \wedge s' \sqsupseteq s \wedge (m_a, m_b') \in \mathrm{configure}(U)(s')(c_a', c_b')(e_a' \cdot \eta_a, e_b' \cdot \eta_b) \wedge \\
\qquad\qquad\qquad (v_a, v_b) \in \langle\!\langle U(s') \rangle\!\rangle^{s'}(\tau) \wedge (e_a', e_b') \in W.\mathsf{cqha}(s')(\mathsf{ret}\, v_a\, k_a^0) \times W.\mathsf{cqhb}(s')(\mathsf{ret}\, v_b\, k_b^0) \\
\qquad \vee\ (\textsc{step})\ (\exists t, m_a'.\ m_a \overset{t}{\hookrightarrow} m_a') \wedge \forall t, m_a'.\ m_a \overset{t}{\hookrightarrow} m_a' \implies \exists e_a', e_b', c_a', c_b', m_b', m_b'', s'. \\
\qquad\qquad\qquad m_b \overset{\epsilon}{\hookrightarrow}{}^{*}\ m_b' \overset{t}{\hookrightarrow} m_b'' \wedge s' \sqsupseteq s \wedge (m_a', m_b'') \in \mathrm{configure}(U)(s')(c_a', c_b')(e_a' \cdot \eta_a, e_b' \cdot \eta_b) \wedge \\
\qquad\qquad\qquad\quad (\textsc{rec})\ \ \ (e_a', e_b') \in \mathbf{E}(k_a^0, k_b^0)(U)(s')(\tau) \\
\qquad\qquad\qquad \vee\ (\textsc{call})\ \exists \tau_v, \tau', f_a, f_b, v_a, v_b, k_a, k_b.\ (e_a', e_b') \in W.\mathsf{cqha}(s')(\mathsf{app}\, f_a\, v_a\, k_a) \times W.\mathsf{cqhb}(s')(\mathsf{app}\, f_b\, v_b\, k_b) \wedge \\
\qquad\qquad\qquad\qquad\qquad (f_a, f_b) \in \langle\!\langle U(s') \rangle\!\rangle^{s'}(\tau_v \to \tau') \wedge (v_a, v_b) \in \langle\!\langle U(s') \rangle\!\rangle^{s'}(\tau_v) \wedge (k_a, k_b) \in \mathbf{K}(k_a^0, k_b^0)(U)(s')(\tau')(\tau)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{configure} \in (W.\mathsf{S} \to \mathrm{VRelF}_{A,B}) \to W.\mathsf{S} \to (A.\mathbf{Conf} \times B.\mathbf{Conf}) \to (A.\mathbf{Conf} \times B.\mathbf{Conf}) \to \mathcal{P}(A.\mathbf{Mach} \times B.\mathbf{Mach}) \\
\mathrm{configure}(U)(s)(c_a', c_b')(c_a, c_b) = \{(m_a, m_b) \in A.\mathrm{real}(c_a \cdot c_a') \times B.\mathrm{real}(c_b \cdot c_b') \mid (c_a', c_b') \in W.\mathsf{crel}(U)(s)\}
\end{array}
$$

$$
\begin{array}{l}
\langle\!\langle - \rangle\!\rangle^{(-)} \in \mathrm{VRelF}_{A,B} \to W.\mathsf{S} \to \mathrm{VRel}_{A,B} \\
\langle\!\langle R \rangle\!\rangle^{s} = \ldots \cup \{(\tau \to \tau', v_a, v_b) \in R \mid v_a \in W.\mathsf{vqha}(s)(\mathsf{fun}) \wedge v_b \in W.\mathsf{vqhb}(s)(\mathsf{fun})\} \\
\qquad\qquad \cup \{(\mathsf{nat}, v_a, v_b) \mid \exists n.\ v_a \in W.\mathsf{vqha}(s)(\mathsf{nat}\, n) \wedge v_b \in W.\mathsf{vqhb}(s)(\mathsf{nat}\, n)\} \\
\qquad\qquad \cup \{(\tau_1 \times \tau_2, v_a, v_b) \mid \exists v_a^1, v_a^2, v_b^1, v_b^2.\ (v_a^1, v_b^1) \in \langle\!\langle R \rangle\!\rangle^{s}(\tau_1) \wedge (v_a^2, v_b^2) \in \langle\!\langle R \rangle\!\rangle^{s}(\tau_2) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v_a \in W.\mathsf{vqha}(s)(\mathsf{pair}\, v_a^1\, v_a^2) \wedge v_b \in W.\mathsf{vqhb}(s)(\mathsf{pair}\, v_b^1\, v_b^2)\}
\end{array}
$$

Figure 8. Key components of PILS (simplified)

e.g., by saying that two pairs are related iff their first projections are related (recursively) and their second projections as well.

When defining the value closure relation generically, we need to have a way of determining how $\mathbf{S}$'s value forms are represented by languages A and B. Since all modules must agree on the representations of values passed by external functions, it makes sense that the global world governs them. This is exactly the purpose of the global world's *value query handlers* vqha and vqhb.

As an example, consider pairs. In $\mathbf{T}$, we choose to represent a pair $\langle v_1, v_2 \rangle$ such that address a holds the representation of v_1 and address $a + 1$ holds the representation of v_2. Since pairs are immutable in the source language, we must ensure that a will continue to represent $\langle v_1, v_2 \rangle$ in the future.

We achieve this by having the global worlds involving $\mathbf{T}$ maintain as part of their state a database of allocated values. Their query handler for $\mathbf{T}$ then checks for a matching entry in this database. Moreover, their crel requires that each value from the database actually exists in memory, with the expected address and representation. Finally, the associated transition system ensures that the database can only ever grow, thus implying that all registered values stay allocated forever. With this in place, the value closure operation can be defined analogously to the one for PBs (as a least fixed point). Some representative cases are shown in Figure 8.

The reader may wonder how we can show that two *functions* are related by $\langle\!\langle U(s') \rangle\!\rangle^{s'}$ if they were defined by us and not passed in from the outside, *i.e.,* not known to be related by U. To do so, it suffices to show they are "similar". This is because PBs (and PILS) impose a "validity" condition on U, namely that it relates *at least* all functions that behave "similarly". Function similarity is defined essentially via Principle 1 in §3, that is: two functions are similar if they map arguments that are assumed-related (roughly, related by U) to results that are behaviorally-related by $\mathbf{E}$.

5.3 A Note on the Untyped Model

Since our source language is type-safe and therefore its well-typed programs "don't go wrong", neither will correctly produced IL or target programs. One may thus wonder why our model takes faulty programs into account (the ERR case in $\mathbf{E}$). The answer is that this feature is actually crucial for verifying transformations in the untyped version of the model. (Recall that we obtain this version by erasing all the type arguments from the definitions in Figures 7–8.)

To see this, first consider the following optimization at the *source* level (where x is a variable of type nat):

$$\mathsf{fix}\, f(x).\ \mathsf{ifnz}\ x\ \mathsf{then}\ e\ \mathsf{else}\ e \quad \rightsquigarrow \quad \mathsf{fix}\, f(x).\ e$$

In the process of showing that $\mathsf{fix}\, f(x).\ e$ is similar to the original function, we will be given arguments related at some unknown relation U and state s, $(\mathsf{nat}, v_a, v_b) \in \langle\!\langle U(s) \rangle\!\rangle^{s}$. Now, by inverting the definition of $\langle\!\langle U(s) \rangle\!\rangle^{s}$, we learn that $\exists n.\ v_a = v_b = n$.

Let us ignore the remaining proof steps and instead consider this transformation at the IL level, where we would be working in the untyped version of the model. There, we will still be given related arguments, $(v_a, v_b) \in \langle\!\langle U(s) \rangle\!\rangle^{s}$, but this time the type information is missing. Consequently, when inverting the value closure, we don't end up with the single case above (where $v_a = v_b = n$), but must also consider all the other cases, such as v_a and v_b being pairs. Now, note two important points: First, the global world $W_{\mathbf{II}}$'s value query handlers ensure that whenever v_b is a number, then so is v_a. In that case we can proceed as we would above in the typed model. Second, if v_b is *not* a number, then the original program produces an error and, thanks to ERR, there is nothing more to show.

6. Improved Reasoning Principles

We explained in §5.2 that PILS necessarily employ an asymmetric small-step formulation of $\mathbf{E}$. However, the particular formulation that we used results in a somewhat limited and inconvenient reasoning principle. Here we explain how to improve on it.

6.1 Allowing Stuttering in a Compositional Way

As mentioned before, our naive asymmetric small-step formulation forces the "source" program e_b to take at least as many steps as the "target" program e_a. This can be seen easily when looking at the reasoning principle inherent in $\mathbf{E}$'s STEP/REC case at a very high level: in order to show $e_a \sim e_b$, it suffices to show

$$\forall e_a'.\ e_a \hookrightarrow e_a' \implies \exists e_b'.\ e_b \hookrightarrow^{+} e_b' \ \wedge\ e_a' \sim e_b'.$$

Note the use of $\hookrightarrow^{+}$, which requires e_b to take at least one step for each step of e_a (here, $e_b \hookrightarrow^{+} e_b'$ corresponds to $m_b \overset{\epsilon}{\hookrightarrow}{}^{*}\ m_b' \overset{t}{\hookrightarrow} m_b''$ in the formal definition of $\mathbf{E}$ in Figure 8).

Naturally, we want to allow the source to "stutter" by replacing $\hookrightarrow^+$ with $\hookrightarrow^*$, but simply doing so would relate a diverging target program to any source program and thus render the model unsound. To solve this, we follow the standard approach [17] of indexing $\mathbf{E}$ by an element of a (proof-local) well-founded partially-ordered set (poset), and then demand that this element be decreased in the case of stuttering.

However, in order to support certain "compatibility" lemmas that are used in our compiler verification (see §7.1), we require a monoid structure on the order (technically speaking, we use non-trivial well-founded positive strictly ordered monoids, or "NWPS monoids" [5]). For example, the compatibility lemma for pairs is roughly as follows:

$$e_a \sim_n e_b \;\wedge\; e'_a \sim_{n'} e'_b \implies \langle e_a, e'_a \rangle \sim_{n+n'} \langle e_b, e'_b \rangle$$

Here e_b and e'_b can stutter at most n and n' "times" respectively, and thus $\langle e_b, e'_b \rangle$ can do so at most $n + n'$ times. We therefore need not just the well-founded poset structure for n, but also a notion of addition, which NWPS monoids provide.

In order to maximize the user's flexibility, we provide a way to lift an arbitrary well-founded poset to an NWPS monoid. More specifically, given a well-founded poset X, we construct an NWPS monoid $\overline{X}$ with an embedding of X into $\overline{X}$ (*i.e.,* an order preserving and reflecting map from X to $\overline{X}$). Thanks to this, the user can pick an arbitrary well-founded poset without worrying about the additional monoid structure required by our proof framework.

6.2 Unchaining Internal Computation

A second shortcoming of the definition of $\mathbf{E}$ is that it does not recognize *internal* steps of computation, treating each step as if it might result in control being passed to the environment. Concretely, after *each* step of the target program and matching steps of the source program, we are obliged to show that the memory constraints currently imposed by the world are again met. And then, in reasoning about the next step of the target program, we are forced to quantify over completely new configurations yet again.

This is unnecessarily strict. Intuitively, we should not need to show that the world's conditions are satisfied again until the point where we pass control to the environment; similarly, we should not need to quantify over new configurations except at points where control is passed to *us*, because there is no way that the state could have changed in between the internal steps of our local computation.

To solve this problem, we parameterize $\mathbf{E}$ with a boolean flag, signalling whether we are currently engaged in internal computation or not. The idea is that when we start a computation and are given configurations related by the world, we temporarily *acquire* them by merging them into our own configurations and setting the flag to true. As we continue executing local steps, we are freed from any worldly obligations. However, before we are allowed to use the RET or CALL case—*i.e.,* when we want to pass control back to the environment—we will be forced to *release* our grip on the world by setting the internal flag to false, at which point we must show that all the world's constraints are again satisfied.

6.3 Exploiting Local Determinism

Recall that $\mathbf{E}$ asks us to consider a single step of the target program at a time and that such a formulation is generally necessary because of non-determinism in the target language. However, reasoning in such a way can be extremely tedious since $\mathbf{T}$ programs are typically very long. Moreover, usually a single source step translates into many target steps, so for most of the target steps one would simply stutter on the source side.

Often one actually knows exactly how the given target program will execute. In these cases, one would like to just take a number of steps on the target side and a number of steps on the source side, and then continue reasoning with the resulting programs. In other words, instead of doing asymmetric small-step reasoning, one would like to do *symmetric big-step reasoning*, which says: in order to show $e_a \sim e_b$, it suffices to show

$$e_a \hookrightarrow^* e'_a \;\wedge\; e_b \hookrightarrow^* e'_b \;\wedge\; e'_a \sim e'_b.$$

Fortunately, based on the previous two changes to $\mathbf{E}$, we can prove a lemma that allows us to do such reasoning *when it is sound to do so*, namely when the target execution in question is guaranteed to behave deterministically. That is, the lemma rests on the idea of lowering determinism from being a property of a language to being a property of a machine configuration.

Consequently, we do not need to impose any restrictions on the languages. This is in contrast to the CompCert compiler, which, in order to enable forward reasoning, uses languages that are internally completely deterministic.

7. The Pilsner and Zwickel Compilers

Using PILS, we have proven *in Coq* the correctness of two compilers from $\mathbf{S}$ to $\mathbf{T}$: Pilsner and Zwickel. Pilsner's structure is depicted in Figure 1. It uses a CPS-based intermediate language and performs several optimizations. Zwickel, on the other hand, is more simplistic: it directly translates $\mathbf{S}$ code into $\mathbf{T}$ code in a straightforward way, similar to Hur and Dreyer's one-pass compiler [9]. In particular, Zwickel neither uses an intermediate language nor performs any CPS transformation. In the remainder of this section, we focus solely on Pilsner, which is by far the more interesting compiler.

Given a source module, Pilsner first translates it to $\mathbf{I}$ via a CPS transformation. It also takes care to alpha-rename all bound variables such that in the resulting $\mathbf{I}$ module, every variable is bound at most once. This *uniqueness condition* simplifies the implementation of most of the subsequent optimization passes, as one does not have to worry about accidental variable capturing when rearranging code. Another nice characteristic of the produced intermediate code (not of $\mathbf{I}$ per se) is that continuations are used in an affine fashion [12], *i.e.,* called at most once. This property is preserved by all other transformations at the intermediate level and enables a more efficient treatment of continuation variables compared to ordinary variables in the code generation pass.

At the intermediate level, Pilsner performs six optimizations. It first inlines selected top-level functions. For instance, if a module defines $F = \mathsf{fix}\, f(y,k).\, e$, then a call to F inside a subsequently defined function will be rewritten as follows:

$$F\ x\ k' \quad\rightsquigarrow\quad e[F/f][x/y][k'/k]$$

(If the function is recursive, *i.e.,* if f is used in e, this is essentially just an unrolling.) Since inlining destroys the uniqueness property of bound variables, we immediately follow it with a "freshening" pass that re-establishes uniqueness.

Next comes contification, which we discuss in §7.2. Subsequently, Pilsner performs a simple dead code (and variable) elimination, rewriting $\mathsf{let}\ x = a\ \mathsf{in}\ e$ to e whenever x does not occur in e. This is justified because, in our IL, evaluation of an atomic expression a does not have any observable side effects. In the same manner, it also eliminates unused read operations and unused allocations (but not write operations because that would be unsound). Following DCE, it hoists let-bindings out of function and continuation definitions, subject to some syntactic constraints. For example:

$$\begin{array}{rl} & \mathsf{let}\ f = (\mathsf{fix}\, f(y,k).\, \mathsf{let}\ z = x.1\ \mathsf{in}\ e)\ \mathsf{in}\ e' \\ \rightsquigarrow & \mathsf{let}\ z = x.1\ \mathsf{in}\ \mathsf{let}\ f = (\mathsf{fix}\, f(y,k).\, e)\ \mathsf{in}\ e' \end{array}$$

if x is none of f, y, k. This avoids recomputation of the projection each time f is called.

Next comes a pass that commutes let-bindings (where possible) in order to group together bindings that assign names to the same expression. For instance:

$$\begin{array}{rl} & \mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ \mathsf{let}\ z = a\ \mathsf{in}\ e \\ \leadsto & \mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ z = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ e \end{array}$$

The last IL transformation, deduplication, gets rid of such consecutive duplicate bindings by rewriting the above expression as follows:

$$\leadsto \quad \mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ e[x/z]$$

This can be seen as a common subexpression elimination.

Code generation, the final pass in the chain, translates to the machine language $\mathbf{T}$. Recall that there are three kinds of "variables" in $\mathbf{I}$: term variables x, continuation variables k, and labels F. Labels are translated to absolute addresses according to the import table. Term variable accesses are translated to lookups (based on position) in a linked list on the heap, pointed to by the env register. Functions are converted to closures, *i.e.,* pairs of environment and code pointer (module-level functions simply have an empty environment), which live on the heap. A closure's environment is loaded into the env register when the function is called. Finally, continuations are allocated on the stack. Accordingly, continuation variable accesses are translated to lookups (based on position) on the stack, with the side effect that the continuation in question, as well as all more-recently defined ones (above it on the stack), are popped. This is safe because the affinity property mentioned earlier ensures that they won't be needed anymore.

7.1 Infrastructure for IL Transformations

For the local IL transformations in Pilsner, we developed a simple framework of transformations as expression annotations. The idea is to split module-level transformations into two parts: (1) an analysis that is applied to each top-level function and annotates selected subexpressions with to-be-performed micro-transformations (but does not actually rewrite the code); and (2) the micro-transformations themselves, together with their correctness proofs. Given these, we automatically produce a verified module transformation that analyzes the input module and performs transformations according to the generated annotations in a bottom-up manner.

A micro-transformation is a partial function on expressions—it must fail if the preconditions for its correctness do not hold. For instance, here is the (only) micro-transformation used in the commute-pass of Pilsner:

$$\begin{array}{l} \mathsf{commute} \in \mathsf{exp} \rightharpoonup \mathsf{exp} \\ \mathsf{commute}(e) := \mathsf{let}\ y = b\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ e_0 \\ \qquad \textit{if } e \textit{ is } (\mathsf{let}\ x = a\ \mathsf{in}\ \mathsf{let}\ y = b\ \mathsf{in}\ e_0) \textit{ and } x \notin \mathrm{FV}(b) \end{array}$$

In the case that a micro-transformation fails (for which the analysis is to blame), the subexpression that was being transformed simply stays unchanged, or, alternatively, the whole module transformation (and thus the compiler) fails. In either case, if the module transformation succeeds, the output module is guaranteed to correctly implement the input module. This means that the analysis does not need to be verified—in the worst case, the transformation doesn't optimize the code.

The concrete correctness property demanded by the framework for each micro-transformation f is twofold. *Syntactic correctness* says that f preserves well-formedness (including affinity of continuation variables) and the uniqueness condition. *Semantic correctness* states the following:

$$\frac{f(e) = e' \quad \Gamma \vdash e \quad \mathrm{unique}(\mathrm{BV}(e), \Gamma)}{\Gamma \vdash e' \preccurlyeq^{*}_{\mathbf{II}} e}$$

The relation in the conclusion is the reflexive transitive closure of $\preccurlyeq_{\mathbf{II}}$, a fairly straightforward lifting of $\mathbf{E}$ from configurations to open $\mathbf{I}$ expressions (not to be confused with module similarity $\precsim_{\mathbf{II}}$). Its definition considers the expressions under an arbitrary unknown relation and state, with pointwise-related environments[8] providing values for the term variables and labels in Γ as well as continuations for the continuation variables in Γ. The local world that it uses is empty, which suffices for reasoning about Pilsner's optimizations.

In order to ease the proofs of semantic correctness, we provide typical *compatibility* lemmas about $\preccurlyeq_{\mathbf{II}}$. They state that the relation is preserved by each language construct of $\mathbf{I}$, and are very helpful in verifying transformations that leave parts of the module unchanged (even outside our annotation framework). The lemmas are straightforward but tedious to show. Only the one about recursive functions requires a proof by coinduction, as one would expect.

7.2 Contification

Pilsner includes a (very simplistic) contification pass. Contification [7] is an optimization that turns a function into a continuation when the function is only ever called with the same continuation argument. This makes control flow more explicit, thus potentially enabling subsequent optimizations. In Pilsner, it has the additional benefit that continuations don't need to be heap-allocated.

Contification in Pilsner uses the framework described above, *i.e.,* it first runs an untrusted analysis that annotates places in the module where the expression-level contification should happen. This saves a lot of work because only the expression-level contification needs to be verified. This transformation consists of two steps. Given a contifiable function binding

$$\mathsf{let}\ f = (\mathsf{fix}\, f(x, k).\, e)\ \mathsf{in}\ e'$$

(for simplicity we gloss over the issue of variable uniqueness here), we first extend it with a fresh continuation definition, namely the contification of f:

$$\begin{array}{l} \mathsf{let}\ f = (\mathsf{fix}\, f(x, k).\, e)\ \mathsf{in} \\ \mathsf{let}\ k_f\ y = e[y/x][k'/k]\ \mathsf{in}\ e' \end{array}$$

Here, k' is the continuation being passed in all invocations of f within e'. In the second step, these invocations of f are turned into calls of the newly added continuation k_f (*e.g.,* $f\ z\ k' \leadsto k_f\ z$). Note that contification does not purge the definition of f, but leaves this to the dead code elimination pass. If the analysis is incorrect, then either the expression-level contification will fail or it will succeed but dead code elimination won't be able to remove the original function binding.

Regarding verification of the expression-level transformation, obviously the first step is trivially correct as it only introduces an unused binding, and so the hard work lies in dealing with the second stage. The core property we need to show is that calls of k_f are related to calls of f, *i.e.,* something along the lines of:

$$\Gamma \vdash k_f\ z \preccurlyeq^{*}_{\mathbf{II}} f\ z\ k'$$

Of course this does not hold in such general form, because the connection between k_f and f would be lost. We must restrict attention to environments in which k_f (in the "target" program) actually maps to the contified version of whatever f maps to (in the "source" program). To do so, we generalize the $\Gamma \vdash e' \preccurlyeq_{\mathbf{II}} e$ judgment to the form $\Gamma; \Sigma_a; \Sigma_b \vdash e' \preccurlyeq_{\mathbf{II}} e$, where Σ_a and Σ_b each map a subset of Γ to closure values with concrete code expressions (if these subsets are empty, we obtain the original judgment). The property we then prove roughly looks as follows:

$$\frac{\Sigma_a(k_f) = \lambda y.\, e[y/x][k'/k] \qquad \Sigma_b(f) = \mathsf{fix}\, f(x, k).\, e}{\Gamma; \Sigma_a; \Sigma_b \vdash k_f\ z \preccurlyeq^{*}_{\mathbf{II}} f\ z\ k'}$$

The generalized judgment is crucial because it supports the same compatibility properties as the original one did (modulo some new

[8] We actually allow variables in the "target" expression to be renamings of those in the "source" expression, as needed e.g. in the proof of deduplication.

side conditions). These compatibility properties enable us to lift the above correctness result, which says we can rewrite $f\ z\ k'$ to $k_f\ z$, to a result which says we can rewrite all calls of f to calls of k_f inside the expression e' in which f is bound.

7.3 Verification of Code Generation

Code generation is the most radical transformation in Pilsner, and so it comes as no surprise that its proof is also the longest. Here we give a brief overview.

The goal is to show codegen$(M) \precsim_{\mathbf{TI}} M$ (ignoring contexts) for well-formed $\mathbf{I}$ module M. With two simple inductions following the recursive definition of code generation (one on module well-formedness, one on expression well-formedness), the goal reduces to showing a collection of compatibility-like lemmas. Each of these states that one particular $\mathbf{I}$ expression form is related to the code generated for it. The hard work lies in defining this relation—let's call it $\preccurlyeq_{\mathbf{TI}}$ (not to be confused with module similarity $\precsim_{\mathbf{TI}}$)—and then proving the lemmas.

In $\preccurlyeq_{\mathbf{TI}}$, we must ultimately say that the two programs are similar according to $\mathbf{E}$. But clearly the generated code makes many assumptions about its environment, *e.g.,* where term variables can be looked up, how continuations are laid out on the stack, where temporary results are placed, etc. In order to restrict the environments in which the code is placed, we must therefore express parts of the compiler-internal protocol that the code follows. Naturally, the world plays a critical role here.

While the proofs of all other passes are oblivious to the local world, for the code generation proof it is critical that we can choose a particular one. We construct it such that its state is always a pair of a heap and a stack, representing the memory used internally by Pilsner. This memory is used for storing (i) code, (ii) variable environment, and (iii) continuations.

Regarding (i), we use our local world in $\preccurlyeq_{\mathbf{TI}}$ to express the initial assumption that the generated code resides in memory and that the program counter points to the first instruction. This alone is not sufficient, though: when reasoning about an external function call we need to know that, when the function returns, our code is still in memory—otherwise, we wouldn't even know which instructions get run next. To achieve this, we define our world's transition relation such that it rules out any mutation of Pilsner-generated code residing on the heap. (Note that this does not prevent other modules that we link with from using self-modifying code themselves.)

Regarding (ii), we use our local world in $\preccurlyeq_{\mathbf{TI}}$ to express that the env register points to a linked list in the heap—the variable environment—and that each of the values stored there is related to its counterpart in the $\mathbf{I}$ program's environment by (the value closure of) U. When reasoning about code involving a variable lookup, we can then be sure to get the correct value. Of course, when reasoning about code that extends the variable environment (*e.g.,* code for a regular let-binding), we must be able to update the state accordingly. For this reason, while disallowing transitions that mutate the environment list, we do allow transitions that extend it.

Regarding (iii), we use our local world in $\preccurlyeq_{\mathbf{TI}}$ to express that for each continuation in the $\mathbf{I}$ program's environment there is a corresponding continuation (code pointer and environment) on the machine stack. This correspondence is not trivial, as it must in turn describe the assumptions that code generated for continuations makes. For instance, such code assumes that before it gets executed the stack is popped as described at the beginning of the section.

7.4 Extraction

Our Coq development contains a script that extracts Pilsner (and Zwickel) as OCaml code and couples it with code for parsing command-line arguments as well as a lexer and parser for the source language. In order to execute target machine code, we have implemented a single-step interpretation function in Coq and proved that it conforms to the operational semantics. This function is extracted to OCaml and wrapped in a loop.

Pilsner provides command-line flags to selectively disable optimizations (more precisely, one flag per IL transformation). Accordingly, the extracted Pilsner function takes not only a source module as argument but also a *selection*, a record of booleans indicating for each transformation whether it is to be performed. We have proven that Pilsner is correct for any such selection and thus for any combination of compiler flags (not shown in Theorem 3).

8. Discussion and Related Work

Proof of transitivity. Recall the statement of Theorem 5. We actually derive this as a corollary of two properties:

Lemma 1 (Transitivity, decomposed)**.**

$$\frac{|\Gamma| \vdash M_{\mathbf{T}} \precsim_{\mathbf{TI}} M_{\mathbf{I}} : |\Gamma'| \qquad \Gamma \vdash M_{\mathbf{I}} \precsim_{\mathbf{IS}} M_{\mathbf{S}} : \Gamma'}{\Gamma \vdash M_{\mathbf{T}} \precsim_{\mathbf{TS}} M_{\mathbf{S}} : \Gamma'} \tag{1}$$

$$\frac{|\Gamma| \vdash M_{\mathbf{I}} \precsim_{\mathbf{II}} M'_{\mathbf{I}} : |\Gamma'| \qquad \Gamma \vdash M'_{\mathbf{I}} \precsim_{\mathbf{IS}} M_{\mathbf{S}} : \Gamma'}{\Gamma \vdash M_{\mathbf{I}} \precsim_{\mathbf{IS}} M_{\mathbf{S}} : \Gamma'} \tag{2}$$

Note that we do not need to show transitivity of $\precsim_{\mathbf{II}}$ itself because we can simply iterate (2).

Thanks to our uniform setup, the proofs of (1) and (2) mirror each other. They also closely follow the transitivity proof of the original PB model [10]. However, since our language "in the middle" is untyped, the complexity having to do with abstract types can be avoided. On the other hand, due to our asymmetric small-step formulation of $\mathbf{E}$ and the possibility of stuttering, the part of the proof dealing with $\mathbf{E}$ becomes significantly more tricky.

As for PBs, one of the main complexities in the PILS transitivity proof lies in dealing with an ambiguity regarding reference allocation: while in one of the two given proofs, an allocation of the middle program may be treated as public (extending the global state), the same allocation may be treated as private (extending the local state) in the other proof. This is a result of transitivity being proven for completely arbitrary local worlds! One might wonder if we could not simplify matters significantly by resorting to an instrumentation of the IL that makes the choice of public vs. private allocation explicit in the program code. It seems to us that this approach only makes sense if one is willing to a priori decide on all subsequent optimizations. The issue is that a later added optimization might, for instance, figure out that some reference is never used and therefore can be removed. Such can only be proven correct if the reference was allocated as private, which it may not have been. For the sake of modularity, we therefore believe it is better to bite the bullet and deal with the ambiguity issue semantically, *e.g.,* in the way we did.

Parametric bisimulations. PBs [9] were inspired by a number of different methods for relational reasoning about higher-order stateful languages: notably, Kripke logical relations [6], normal form bisimulations [23], and environmental bisimulations [24, 21]. From Kripke logical relations, PBs adapted the mechanism of possible worlds as state transition systems, enabling the enforcement of protocols on local or global state. From normal form bisimulations, PBs took the central idea of viewing unknown functions as black boxes—in particular, the CALL case in Figure 8 is highly reminiscent of a similar case in normal form bisimulations. From environmental bisimulations, PBs borrowed the treatment of abstract types and polymorphism via type names (which we have glossed over here).

The key advance of PBs was to show how to combine all these mechanisms in a way that supported transitive composition and did not rely on "syntactic" devices employed by the other higher-order simulation methods (*e.g.,* modeling related unknown functions as a common free variable [23], or using context closure operations [24, 21]), because such syntactic devices would preclude a generalization to inter-language reasoning. But it was far from obvious whether PBs would necessarily fare any better in this regard.

In this paper, we have demonstrated through PILS that the claims of Hur *et al.* [9] were indeed correct, and that PBs do in fact generalize to inter-language reasoning as promised.

Multi-language semantics. Motivated by the goal of supporting compiler verification for programs that interoperate between different languages, Perconti and Ahmed [19] propose an approach based on *multi-language semantics* [16]. In particular, they define a "big-tent" language that comprises the source, target, and intermediate languages of a compiler, and provides "wrapping" operations for embedding terms of each language within the others. They then use logical relations to prove that every source module is contextually equivalent to a suitably wrapped version of the target module to which it is compiled. In this way, their method synthesizes the benefits of logical relations (modularity and different source and target languages) and contextual equivalence (transitivity).

One downside of their approach is that the intermediate languages (ILs) used in a compiler show up explicitly in the statement of compiler correctness. This leads to a loss of flexibility: the semantics of source-level linking is not preserved when linking the results of compilers that have different ILs. Another limitation with respect to flexibility is that their approach seems to be restricted to compilers that use *typed* intermediate and assembly languages, and has only so far been applied to a purely functional source language. On the other hand, Perconti and Ahmed are more flexible than we are with respect to multi-language interoperation. One of their explicit goals is to reason about the linking of ML code with *arbitrary* typed assembly code, whereas we only support verified linking with assembly modules that refine *some* source-level counterpart. As we observed in footnote 1, we do not believe this is a fundamental limitation of our approach: it should in principle be possible to develop PILS for a different source language in which high- and low-level modules may interoperate, in which case the "source"-level specification of a "target"-level module could be the target-level module itself. But we leave that to future work.

Compositional verified compilation for C. Motivated by the goal of compositional compiler verification, Beringer *et al.* [4, 22] propose an adaptation of the CompCert framework based on a novel "interaction" semantics that differentiates between internal (intra-module) and external (inter-module) function calls. They introduce a notion of "structured simulation" that assumes little about the memory transformations performed by external function calls.

Beringer *et al.*'s approach is transitive, and like Perconti and Ahmed's (but unlike ours), it supports verified compilation of multi-language programs—in this case, programs that link C and assembly modules. However, also like Perconti and Ahmed's approach, Beringer *et al.*'s is somewhat lacking in flexibility. It depends on compiler passes only performing a restricted set of memory transformations—additional transformations could potentially break the transitivity property. In addition, their method appears to be geared specifically toward compilers à la CompCert, which employ a uniform memory model across source, intermediate, and target languages. It is not clear how to generalize their technique to support richer (*e.g.,* ML-like) source languages, or compilers whose source and target languages have different memory models.

Wang *et al.* [25] have also recently explored compositional compiler verification for a restricted C-like language called Cito. Their approach embeds the verification statement within a Hoare logic for partial correctness of assembly modules, thus enabling support for verified cross-language linking, but without guaranteeing preservation of termination behavior. Further work is needed to better understand the relationship between this approach and traditional refinement-based compiler verification.

Acknowledgements

This research has been supported in part by a Google European Doctoral Fellowship granted to the first author, by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) / National Research Foundation of Korea(NRF) (Grant NRF-2008-0062609), and by an internship from MPI-SWS.

References

[1] Appendix and Coq development. `http://plv.mpi-sws.org/pils`.

[2] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

[3] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

[4] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *ESOP*, 2014.

[5] D. Dobbs, M. Fontana, and S.-E. Kabbaj, editors. *Advances in Commutative Ring Theory*. CRC Press, 1999.

[6] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *JFP*, 22(4-5), 2012.

[7] M. Fluet and S. Weeks. Contification using dominators. In *ICFP*, 2001.

[8] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *POPL*, 2011.

[9] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, 2012.

[10] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The transitive composability of relation transition systems. Technical Report MPI-SWS-2012-002, MPI-SWS, 2012.

[11] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. A logical step forward in parametric bisimulations. Technical Report MPI-SWS-2014-003, MPI-SWS, 2014.

[12] A. Kennedy. Compiling with continuations, continued. In *ICFP*, 2007.

[13] R. Kumar, M. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *POPL*, 2014.

[14] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[15] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[16] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL*, 2007.

[17] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *FSTTCS*, pages 284–296, 1997.

[18] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, 2010.

[19] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, 2014.

[20] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[21] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *LICS*, 2007.

[22] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *POPL*, 2015.

[23] K. Støvring and S. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *POPL*, 2007.

[24] E. Sumii and B. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54(5):1–43, 2007.

[25] P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *OOPSLA*, 2014.