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 S to an assembly-like target language 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 S, and (2) a hand-coded self-modifying T module, proven correct w.r.t. an S-level specification. The output of Zwickel and the self-modifying 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 S and a distinguished target language T. Given a whole S program ps, the compiler’s output tr(ps) should be a whole T program pt that refines ps. Refinement means that any observable behavior of pt should also be a valid observable behavior of ps, for some common notion of “observable behavior” that both the S and 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 mT contextually refines source module mS if C[mT] refines C[mS] for all closing program contexts C. However, contextual refinement fundamentally assumes that mT and mS 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...
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 (S) module $m_S$ is refined by target (T) module $m_T$, and that S module $m'_S$ is refined by T module $m'_T$. We should then be able to conclude that the S-level linking of $m_S$ and $m'_S$ is refined by the T-level linking of $m_T$ and $m'_T$. (This is sometimes referred to as “horizontal compositionality”.)

• **Transitivity:** Proofs of semantic preservation should be transitive-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 PILS, a simple (but non-trivial) multi-pass optimizing compiler from an ML-like source language S (supporting recursive types, abstract types, and general references) to an idealized assembly-like target language T, going through a CPS-based intermediate language I. After CPS conversion, PILS performs several optimizations at the I level prior to code generation. These optimizations include function inlining, constification, dead code elimination, and hoisting (Figure 1). Although PILS 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 S to T, and (2) Hur and Dreyer’s aforementioned self-modifying code example, programmed as a T module and proven correct w.r.t. an S-level specification. Thanks to PILS’ modularity, the output of Zwickel and the self-modifying T module can then be safely linked together with the output of PILS. 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 PILS and Zwickel (§4–§6), we highlight interesting aspects of the PILS 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 S and a low-level machine target language 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 S and a T version, and we aim to prove that the T version refines its S counterpart. For modules that are compiled by a verified compiler, the T version is generated automatically by the compiler. But for any module that is hand-coded in T, one must also manually supply its S counterpart, which serves as a “specification” that the hand-coded 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 T modules that have some S-level counterpart. This is somewhat of a restriction at present, since we focus here on the setting where S is a high-level, ML-like language and 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 S itself supports interoperability 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: $\Gamma \vdash M_T \sim_{TS} M_S : \Gamma'$
intuitively states that target module $M_T$ refines source module $M_S$
and that they import the functions listed in $\Gamma$ and export those in
$\Gamma'$. The first key result, Theorem 1, applies to whole programs,
{	extit{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.

\begin{theorem}[Adequacy for whole programs]
\label{th:adequacy}
\begin{align*}
\Gamma \vdash M_T \sim_{TS} M_S : \Gamma \\
\text{Behav}(M_T) \subseteq \text{Behav}(M_S)
\end{align*}
\end{theorem}

Behav$(-)$ denotes the set of I/O and termination behaviors that
a program can have. The theorem implies for instance that, if $M_S$
always successfully terminates, then so does $M_T$ and moreover they
produce the same outputs.

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

\begin{theorem}[Preservation under linking, a.k.a. modularity]
\label{th:preservation}
\begin{align*}
\Gamma \vdash M_1 \sim_{TS} M'_1 : \Gamma_1, \Gamma_1 \vdash M_2 \sim_{TS} M'_2 : \Gamma_2 \\
\Gamma \vdash (M_1 \bowtie M_2) \sim_{TS} (M'_1 \bowtie M'_2) : \Gamma_1 \cup \Gamma_2
\end{align*}
\end{theorem}

(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 $\Gamma_1$) 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 $S$ to $T$. Their correctness theorems apply to any well-typed
source module:

\begin{theorem}[Correctness of PILSner]
\label{th:correctness_pilsner}
\begin{align*}
\Gamma \vdash M_S : \Gamma' \\
\Gamma \vdash \text{Pilsner}(M_S) \sim_{TS} M_S : \Gamma'
\end{align*}
\end{theorem}

\begin{theorem}[Correctness of Zwickel]
\label{th:correctness_zwickel}
\begin{align*}
\Gamma \vdash M_S : \Gamma' \\
\Gamma \vdash \text{Zwickel}(M_S) \sim_{TS} M_S : \Gamma'
\end{align*}
\end{theorem}

While Zwickel carries out a straightforward direct translation from
$S$ to $T$, PILSner is more sophisticated: as shown in Figure 1, it
compiles via an intermediate language $I$ 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 $S$ to $T$, 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. $\sim_{TS}$,
such that Theorem 2 applies.

\subsection{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
\begin{align*}
\Gamma \vdash M_T & \sim_{TI} M_I : [\Gamma'] \\
\Gamma \vdash M_I & \sim_{IH} M'_I : [\Gamma'] \\
\Gamma \vdash M'_I & \sim_{IS} M_S : [\Gamma']
\end{align*}
\begin{align*}
\Gamma \vdash M_T \sim_{TS} M_S : [\Gamma']
\end{align*}

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

\subsection{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\footnote{After slightly modifying the machine code to account for a difference in calling convention.} the challenging refinement from Hur and Dreyer\cite{HurDreyer07} 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\cite{PittsStark98}, which is easy to explain:

\begin{align*}
e_0 & := \text{let } x := \text{ref } 0 \text{ in } \lambda f. (x := 1; f();!x) \\
e_1 & := \lambda f. (f();!1)
\end{align*}

Both expressions evaluate to higher-order functions that, when
applied, call the argument “callback” function $f$ and then return
a number. In $e_0$ this number is simply 1. In $e_1$ it is the result of
dereferencing a local (private) reference $x$, which is initialized
to 0. Notice, though, that when $e_0$ 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, $e_0$ and
$e_1$ are equivalent programs (see the next section for a proof sketch).

Hur and Dreyer\cite{HurDreyer07} adapt this example by substituting for $e_0$ 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 $E(-)$ denotes the encoding of an instruction as a
machine word. Notice that part of the code has been "encrypted" by adding 666 to its encoding. We briefly explain how the code works.

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

Hur and Dreyer showed that this contrived implementation refines the high-level program \( e_a \) 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 \( \equiv_{TS} \)), 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 PBs and KLRs, 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 \( l_2 \) — 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 \( s_0 \):

\[
\begin{align*}
  s_0: & \quad l_2 \rightarrow 0 \\
  s_1: & \quad l_2 \rightarrow 1
\end{align*}
\]

This protocol expresses that the value at \( l_2 \) 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 \( l_2 \), 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).

\[
\begin{align*}
  v_a := \lambda f. \langle l_2 := 1; f(); !l_1 \rangle \\
  v_b := \lambda f. \langle f(); 1 \rangle
\end{align*}
\]

(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 \( f_a \) and \( f_b \) 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 \( v_a \) and \( v_b \) behave equivalently. When the execution of these functions begins, the state of \( l_2 \)'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, \( v_a \) and \( v_b \) will first set \( l_2 \) 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 \( (f_a(); !l_2) \) and \( (f_b(); 1) \) 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 \( f_a() \) and \( f_b() \) behave equivalently. The result therefore reduces to showing that \( !l_2 \) 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 \( !l_2 \) evaluates to 1. Thus, \( !l_2 \) 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 $S$, 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).

### 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. $U$ and $\langle f, U \rangle$ could relate any functions $f$, and $f'$. 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 from $f$, and $f'$ related by $U$. This amounts to baking Principle 2 directly into the definition of behavioral equivalence. The reason this is necessary—is, the reason Principle 2 does not just fall out of the definition otherwise—is that $U$ is a parameter of behavioral equivalence. Knowing that $f$, and $f'$ 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.

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 $f$, and $f'$ 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 $S$ to a machine language $T$, and that the more complex of our two compilers, Pilsner, employs an intermediate language $I$. As argued before, besides the main similarity relation $\sim_{IIS}$ between $T$ and $S$ 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., $\sim_{IT}$, $\sim_{II}$, and $\sim_{IS}$.

#### 4.1 Language-Generic Approach

In order to avoid duplicate work, we define PILS in a language-generic way, i.e., we define similarity $\sim_{AB}$ 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 $T$ 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...
We now describe the language abstraction in terms of which PILS

\[ \text{core of the operational semantics is} \]

functions, \( \text{cload} \) \& \( \text{vload} \) (think: load addresses). The \( \text{cload} \) \& \( \text{vload} \) their unique labels. In the language specification, the module \( \text{exports} \) \( \text{error} \) \& \( \text{real} \) \( \text{contains exactly one machine.} \) Configurations must form a partial

empty, meaning the configuration is invalid or incomplete, or it

real \( \text{given in the form of a transition system} \)

\[ \text{machines, whose} \]

\( \text{transitions are labelled with events} \)

\( \text{t} \)

\[ \text{machines, whose} \]

\( \text{Fig. 3. Language specification} \)

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 \( \text{two} \) generic models: a typed one and an untyped one. The former is used when the input language is \( \text{S} \), 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 \( \text{S, I, T} \). Common to all

languages are a set of events and a countably infinite set of labels:

\( t \in \text{Evt} \leftrightarrow \epsilon \ | \ 7n | \ n \ | \ F_1, F_2, \ldots \in \text{Lbl} \)

Events are produced by an executing program; they consist of

internal computation \( \epsilon \) \& 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 \( \text{S} \) on the one

extreme and a very low-level language \( \text{T} \) on the other extreme.

A language must come with a set \( \text{Val} \) of values, a set \( \text{Cont} \) of continuations, a set \( \text{Conf} \) of configurations, a set \( \text{Mach} \) of

machines, a set \( \text{Mod} \) of modules, and a set \( \text{Anch} \) of anchors

(think: load addresses). The core of the operational semantics is given in the form of a transition system \( \langle \rightarrow \rangle \) 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 \textit{realized} by

a set of machines \( \text{real}(c) \). (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 \( \cdot \) \& neutral element \( \emptyset \), except that the partiality is implicit via real. (Having \( \cdot \) total is more convenient for mechanization [18].) We say a machine denotes an \textit{error}, \( m \in \text{error} \), iff it does not realize any configuration.

Intuitively, modules are sets of labelled function values \( \text{the exports} \) \( \text{which may refer to external functions} \) \( \text{the imports} \)

by their unique labels. In the language specification, the module

interface consists of two operations, \( \text{cload} \) \& \( \text{vload} \). The former,

\( \text{cload} \), 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, \( \text{vload} \) returns the module’s corresponding function value.
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 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, let \(k \ y \ e_1 \ e_2\) defines a new continuation \(k\) with argument \(y\) and body \(e_1\), and then executes \(e_2\) which may capture \(y\) in \(e_1\). Here \(y \in \text{TVar}\) is term variable, while \(k \in \text{KVar}\) 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 \(\text{Cont}\) simply as \(\text{Val}\) 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 \(\text{sto} \ r_2\) 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).

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 \(\text{clo} + 0\), 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 \((W_{T1}, W_{R1}, W_{S1}, \text{and } W_{R2})\). 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 \(\text{crel}\), 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 \(\sqsubseteq\), 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
At the top-level, PILS define module similarity, which we instantiate when the protocol involves higher-order state (see [9]).

In the explanation above, we glossed over many details. For a start, we should not have touched the frame’s private state. Moreover, we find it helpful to refer to the global part of the machine state constrained by the world $W$, and require—in the helper definition configure—that they are indeed related by $W$. Hence we quantify over $s_a$ and $s_b$, 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 $W.crel(U)(s)$. We then attach these to $s_a$ and $s_b$, together with arbitrary frame configurations $n_a$ and $n_b$ 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 $n_a$, and the original frame configuration $n_b$ (similarly for the b-side), since we should not have touched the frame’s private state. Moreover, $e_a$ and $e_b$ must again satisfy $W$’s constraints, but we may advance $s$ to a future state $s’$ in order to achieve that.

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 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 PILS 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 cqb that answer questions such as “in state $s’$, does $e_a$ constitute a return of value $v_a$ to continuation $k_0^a$?”.

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 cqb 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: $(v_a, v_b) \in (U(s’))^{x(\tau)}$. 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.
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 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 vqb.

As an example, consider pairs. In T, we choose to represent a pair (v1, v2) such that address a holds the representation of v1 and address a + 1 holds the representation of v2. Since pairs are immutable in the source language, we must ensure that each value from the database actu-

We achieve this by having the global worlds involving T maintain as part of their state a database of allocated values. Their query handler for 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 \(\langle U(s)\rangle^a\), 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 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 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):

\[ \text{fix } f(x). \text{ fix } x \text{ then } e \text{ else } e \leadsto \text{ fix } f(x). e \]

In the process of showing that fix f(x), e is similar to the original function, we will be given arguments related at some unknown relation U and state s, \(\langle v_a, v_b, v_k \rangle \in \langle U(s) \rangle^s\). Now, by inverting the definition of \(\langle U(s) \rangle^s\), 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, \(\langle v_a, v_b \rangle \in \langle U(s) \rangle^s\), 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 Wvlt’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 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 \(e_a\) to take at least as many steps as the “target” program \(e_b\). This can be seen easily when looking at the reasoning principle inherent in E’s step/rec case at a very high level: in order to show \(e_a \leadsto e_b\), it suffices to show

\[ \forall e_a, e_a \leadsto e'_a \implies \exists e_b, e_b \leadsto^+ e'_b \land e'_a \leadsto e'_b. \]

Note the use of \(\leadsto^+\), which requires \(e_b\) to take at least one step for each step of \(e_a\) (here, \(e_a \leadsto^+ e'_a\) corresponds to \(m_b \leadsto m_b' \leadsto m_b''\) in the formal definition of E in Figure 8).
Naturally, we want to allow the source to "stutter" by replacing \( \leadsto \) with \( \leftarrow \), 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 \( E \) 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:

\[
\langle a, b \rangle \sim_n \langle c, d \rangle \quad \text{if and only if} \quad \langle a', b' \rangle \sim_{n+1} \langle c', d' \rangle
\]

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 \( \tilde{X} \) with an embedding of \( X \) into \( \tilde{X} \) (i.e., an order preserving and reflecting map from \( X \) to \( \tilde{X} \)). 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 \( E \) 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 \( E \) 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 worrying about the additional world’s constraints are again satisfied.

6.3 Exploiting Local Determinism

Recall that \( E \) 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 \( T \) 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

\[
e_a \sim_e e'_a \land e_b \sim^* e'_b \land e'_a \sim e'_b.
\]

Fortunately, based on the previous two changes to \( E \), we can prove a lemma that allows us to do such reasoning when \( it \) is \( sound \), 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 \( S \) to \( T \): 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 \( S \) code into \( T \) 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 \( I \) via a CPS transformation. It also takes care to alpha-rename all bound variables such that in the resulting \( I \) 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 \( I \) 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 \( F = \text{fix } f(y, k) \), \( e \), then a call to \( F \) inside a subsequently defined function will be rewritten as follows:

\[
F \ x \ k' \ \sim \ e[F/f][x/y][k'/k]
\]

(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 confluence, which we discuss in §7.2. Subsequently, Pilsner performs a simple dead code (and variable) elimination, rewriting let \( x = a \) 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:

\[
\begin{align*}
\text{let } f &= \text{fix } f(y, k) \text{, let } z &= x.1 \text{ in } e \text{ in } e' \sim \\
\text{let } z &= x.1 \text{ in let } f &= \text{fix } f(y, k), e \text{ in } e' \end{align*}
\]

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:
\[
\begin{align*}
\text{let } x &= a \text{ in let } y = b \text{ in let } z = a \text{ in } e \\
\leadsto \text{let } x &= a \text{ in let } y = b \text{ in } e
\end{align*}
\]

The last IL transformation, duplication, gets rid of such consecutive duplicate bindings by rewriting the above expression as follows:
\[
\text{let } x = a \text{ in let } y = b \text{ in } e[x/z]
\]

This can be seen as a common subexpression elimination.

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:

\[
\text{commute } e \rightarrow \text{exp}
\]

\[
\text{commute}(e) ::= \begin{cases} 
\text{let } y = b \text{ in let } x = a \text{ in } e_0 & \text{if } e \text{ is } (\text{let } x = a \text{ in let } y = b \text{ in } e_0) \text{ and } x \notin \text{FV}(b) 
\end{cases}
\]

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:

\[
\begin{align*}
\Gamma; \Sigma &\vdash e \in \text{exp} \quad \text{unique(BV(e), } \Gamma) \\
\Gamma &\vdash e' \preceq_{11} e
\end{align*}
\]

The relation in the conclusion is the reflexive transitive closure of $\preceq_{11}$, a fairly straightforward lifting of $E$ from configurations to open $I$ expressions (not to be confused with module similarity $\preceq_{11}$). 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 $\Gamma$ as well as continuations for the continuation variables in $\Gamma$. 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 $\preceq_{11}$. They state that the relation is preserved by each language construct of $I$, 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

\[
\begin{align*}
\text{let } f &= (\text{fix } f(x,k), e) \text{ in } e'
\end{align*}
\]

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

\[
\begin{align*}
\text{let } f &= (\text{fix } f(x,k), e) \text{ in } e' \\
\text{let } k_f &= e[y/x][k'/k] \text{ in } e'
\end{align*}
\]

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:

\[
\Gamma \vdash k_f z \preceq_{11} f z k'
\]

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 $\Gamma \vdash e \preceq_{11} e$ judgment to the form $\Gamma; \Sigma_0; \Sigma_1 \vdash e \preceq_{11} e$, where $\Sigma_0$ and $\Sigma_1$ each map a subset of $\Gamma$ 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:

\[
\begin{align*}
\Sigma_a(k_f) &= \lambda y. e[y/x][k'/k] \\
\Sigma_a(f) &= \text{fix } f(x,k), e \\
\Gamma; \Sigma_0; \Sigma_1 &\vdash k_f z \preceq_{11} f z k'
\end{align*}
\]

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 \( f \mapsto k' \mapsto k_f z \), 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 \( \text{codegen}(M) \preceq_{\text{TI}} M \) (ignoring contexts) for well-formed \( I \) 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 \( I \) expression form is related to the code generated for it. The hard work lies in defining this relation—let’s call it \( \preceq_{\text{TI}} \) (not to be confused with module similarity \( \preceq_{\text{TI}} \))—and then proving the lemmas.

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

 Naturally, the 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 \( \preceq_{\text{TI}} \) 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 \( \preceq_{\text{TI}} \) to express that the \( \text{env} \) 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 \( I \) 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 \( \preceq_{\text{TI}} \) to express that for each continuation in the \( I \) 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 Zwinkel) 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:

\[ \text{Lemma 1 (Transitivity, decomposed).} \]

\[ \Gamma \vdash M_T \preceq_{\text{TI}} M_1 : \Gamma' \hspace{1cm} \Gamma \vdash M_1 \preceq_{\text{IS}} M_S : \Gamma' \]

\[ \Gamma \vdash M_T \preceq_{\text{PS}} M_S : \Gamma' \hspace{1cm} \Gamma \vdash M_T \preceq_{\text{TI}} M_1 : \Gamma' \]

\[ \Gamma \vdash M_1 \preceq_{\text{IS}} M_S : \Gamma' \]

Note that we do not need to show transitivity of \( \preceq_{\text{PS}} \) 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 \( E \) and the possibility of stuttering, the part of the proof dealing with \( E \) 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 interoperability. 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 producing 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**


