# Bringing Extensibility to Verified Compilers *

Zachary Tatlock Sorin Lerner
University of California, San Diego
{ztatlock,lerner}@cs.ucsd.edu

## Abstract

Verified compilers, such as Leroy's CompCert, are accompanied by a fully checked correctness proof. Both the compiler and proof are often constructed with an interactive proof assistant. This technique provides a strong, end-to-end correctness guarantee on top of a small trusted computing base. Unfortunately, these compilers are also challenging to extend since each additional transformation must be proven correct in full formal detail.

At the other end of the spectrum, techniques for compiler correctness based on a domain-specific language for writing optimizations, such as Lerner's Rhodium and Cobalt, make the compiler easy to extend: the correctness of additional transformations can be checked completely automatically. Unfortunately, these systems provide a weaker guarantee since their end-to-end correctness has not been proven fully formally.

We present an approach for compiler correctness that provides the best of both worlds by bridging the gap between compiler verification and compiler extensibility. In particular, we have extended Leroy's CompCert compiler with an execution engine for optimizations written in a domain specific language and proved that this execution engine preserves program semantics, using the Coq proof assistant. We present our CompCert extension, XCert, including the details of its execution engine and proof of correctness in Coq. Furthermore, we report on the important lessons learned for making the proof development manageable.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification – Correctness proofs; D.3.4 [*Programming Languages*]: Processors – Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs – Mechanical verification

***General Terms*** Languages, Verification, Reliability

***Keywords*** Compiler Optimization, Correctness, Extensibility

## 1. Introduction

Optimizing compilers are a foundational part of the infrastructure developers rely on every day. Not only are compilers expected to produce high-quality optimized code, but they are also expected to be correct, in that they preserve the behavior of the compiled programs. Even though developers hit bugs only occasionally when using mature optimizing compilers, getting compilers to a level of reliability that is good enough for mainstream use is challenging and extremely time consuming. Furthermore, in the context of safety-critical applications, e.g. in medicine or avionics, compiler correctness can literally become a matter of life and death. Developers in these domains are aware of the risk presented by compiler bugs; imagine the care you would take in writing a compiler if a human life depended on its correctness. To guard against disaster they often disable compiler optimizations, perform manual reviews of generated assembly, and conduct exhaustive testing, all of which are expensive precautions.

One approach to ensure compiler reliability is to implement the compiler within a proof assistant like Coq and formally prove its correctness, as done in the CompCert verified compiler [9]. Using this technique provides a strong end-to-end guarantee: each step of the compilation process is fully verified, from the first AST transformation down to register allocation. Unfortunately, because the proofs are not fully automated, this technique requires a large amount of manual labor by developers who are both compiler experts and comfortable using an interactive theorem prover. Furthermore, extending such a compiler with new optimizations requires proving each new transformation correct in full formal detail, which is difficult and requires substantial expertise [15–17].

Another approach to compiler reliability is based on using a domain-specific language (DSL) for expressing optimizations; examples include Rhodium [8] and PEC [7]. These systems are able to automatically check the correctness of optimizations expressed in their DSL. This technique provides superior extensibility: not only are correctness proofs produced without manual effort, but the DSL provides an excellent abstraction for implementing new optimizations. In fact, these systems are designed to make compilers extensible even for non-compiler experts. Unfortunately, the DSL based approach provides a weaker guarantee than verified compilers, since the execution engine that runs the DSL optimizations is not proved correct.

In this paper we present a hybrid approach to compiler correctness that achieves the best of both techniques by bridging the gap between verified compilers and compiler extensibility. Our approach is based on a DSL for expressing optimizations coupled with both a fully automated correctness checker and a verified execution engine that runs optimizations expressed in the DSL. We demonstrate the feasibility of this approach by extending CompCert with a new module XCert ("Extensible CompCert"). XCert combines the DSL and automated correctness checker from PEC [7] with an execution engine implemented as a pass within CompCert and verified in Coq.

XCert achieves a strong correctness guarantee by proving the correctness of the execution engine fully formally, but it also provides excellent extensibility because new optimizations can be easily expressed in the DSL and then checked for correctness fully

* Supported in part by NSF grants CCF-0644306 and CCF-0811512.



*PLDI'10,* June 5–10, 2010, Toronto, Ontario, Canada.

automatically. In particular, while adding only a relatively small amount to CompCert's trusted computing base (TCB), our technique provides the following benefit: additional optimizations that are added using PEC *do not require any new manual proof effort, and do not add anything to the TCB*.

The main challenge in adding a PEC execution engine to CompCert lies in verifying its correctness in Coq. The verification is difficult for several reasons. First, it introduces new constructs into the CompCert framework including parameterized programs, substitutions, pattern matching, and subtle CFG-manipulation operations. These constructs require careful design to make reasoning about the execution engine manageable. Second, the execution engine imports correctness guarantees provided by PEC into CompCert, which requires properly aligning the semantics of PEC and CompCert. Third, applying the PEC guarantee within the correctness proof of the engine is challenging and tedious because it requires knowing information outside the engine about tests performed deep within the engine.

We discuss three general techniques that we found extremely useful in mitigating these difficulties: (1) *Verified Validation*, a technique inspired by Tristan et al, where, for certain algorithms in the PEC engine, we reduce proof effort by implementing a verified result checker rather than directly verifying the algorithm; (2) *Semantics Alignment*, where we factor out into a separate module the issues related to aligning the semantics between PEC and CompCert, so that these difficulties do not pervade the rest of the proof; and (3) *Witness Propagation*, where we return extra information with the result of a transformation which allows us to simplify applying the PEC guarantee and reduce case analyses.

Our contributions therefore include:

- XCert, an extension to CompCert based on PEC that provides both extensibility and a strong end-to-end guarantee. We first review PEC and CompCert in Section 2, and then present our system and its correctness proof in Sections 3 and 4.
- Techniques to mitigate the complexity of such proofs and lessons learned while developing our proof (Sections 3, 4 and 5). These techniques and lessons are more broadly applicable than our current system.
- A quantitative and qualitative assessment of XCert in terms of trusted computing base, lines of code, engine complexity and proof complexity, and a comparison using these metrics with CompCert and PEC (Section 6).

## 2. Background

In this section, we review background material on the PEC system [7] and the CompCert verified compiler [9].

### 2.1 Parameterized Equivalence Checking (PEC)

PEC is a system for implementing optimizations and checking their correctness automatically. PEC provides the programmer with a domain-specific language for implementing optimizations. Once optimizations are written in this language, PEC takes advantage of the stylized forms of the optimizations to check their correctness automatically.

***Loop peeling*** We show how PEC works through a simple example, loop peeling. Loop peeling is a transformation that takes one iteration of a loop, and moves it either before or after the loop. An instance of this transformation is shown in Figure 1. Loop peeling can be used for a variety of purposes, including modifying loop bounds to enable loop unrolling or loop merging.

Optimizations in PEC are expressed as guarded rewrite rules of the following form:

$$G_\ell \longmapsto G_r \textbf{ where } S$$

```
k := 0                          k := 0
while (i < 100) {               while (k < 99) {
   a[k] += k;                      a[k] += k;
   k++;                            k++;
}                               }
                                a[k] += k;
                                k++;
```

(a) (b)

**Figure 1.** Loop peeling: (a) shows the original code, and (b) shows the transformed code.

$$\left[\begin{array}{l}\mathbf{I}\texttt{ := 0}\\ \texttt{while (}\mathbf{I}\texttt{ < }\mathbf{E}\texttt{) \{}\\ \quad\mathbf{S}\\ \quad\mathbf{I}\texttt{++}\\ \texttt{\}}\end{array}\right] \longmapsto \left[\begin{array}{l}\mathbf{I}\texttt{ := 0}\\ \texttt{while (}\mathbf{I}\texttt{ < }\mathbf{E}\texttt{-1) \{}\\ \quad\mathbf{S}\\ \quad\mathbf{I}\texttt{++}\\ \texttt{\}}\\ \mathbf{S}\\ \mathbf{I}\texttt{++}\end{array}\right]$$

**where** $\mathit{NotMod}(\mathbf{S},\mathbf{I}) \wedge \mathit{NotMod}(\mathbf{S},\mathbf{E}) \wedge \mathit{StrictlyPos}(\mathbf{E})$

**Figure 2.** Loop peeling expressed in PEC

where $G_\ell$ is a code pattern to match, $G_r$ is the code to replace any matches with, and the side condition $S$ is a boolean formula stating the condition under which the rewrite may safely be performed. Throughout the paper we use subscript $\ell$ (which stands for "left") for the original program and subscript "r" (which stands for "right") for the transformed program. Figure 2 shows a simple form of loop peeling, expressed in PEC's domain-specific language. The variables $\mathbf{S}$, $\mathbf{I}$ and $\mathbf{E}$ are PEC *pattern variables* that can match against pieces of concrete syntax: $\mathbf{S}$ matches statements, $\mathbf{I}$ variables, and $\mathbf{E}$ expressions.

The semantics of a rewrite rule $G_\ell \longmapsto G_r$ **where** $S$ is that, for any substitution $\theta$ mapping pattern variables to concrete syntax, if $\theta(G_\ell)$ is found somewhere in the original program (where $\theta(G_\ell)$ denotes applying the substitution $\theta$ to $G_\ell$ to produce concrete code), then the matched code is replaced with $\theta(G_r)$, as long as $S(\theta(G_\ell), \theta(G_r))$ holds.

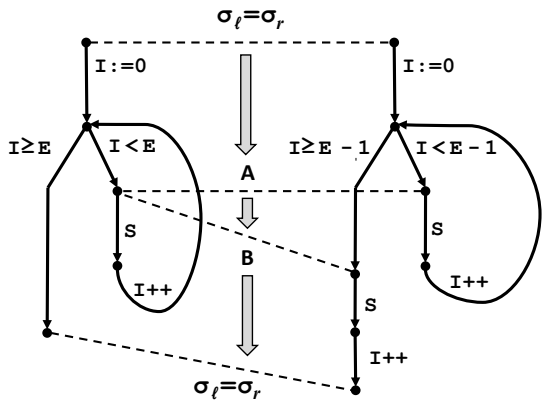
The side condition $S$ is a conjunction over a fixed set of *side condition predicates*, such as *NotMod* and *StrictlyPos*. These side condition predicates have a fixed semantic meaning – for example, the meaning of $\mathit{StrictlyPos}(\mathbf{I})$ is that $\mathbf{I}$ is greater than 0. PEC trusts that the execution engine provides an implementation of these predicates that implies their semantic meaning: if the implementation of the predicate returns true, then its semantic meaning must hold.

***Correctness checking*** PEC tries to show that a rewrite rule $G_\ell \longmapsto G_r$ **where** $S$ is correct by matching up execution states in $G_\ell$ and $G_r$ using a simulation relation. A simulation relation $\sim$ is a relation over program states in the original and transformed programs. Intuitively, $\sim$ relates a given state $\eta_\ell$ of the original program with its corresponding state $\eta_r$ in the transformed program.

The key property to establish is that the simulation relation is preserved throughout execution. Using $\rightarrow$ to denote small-step semantics, this property can be stated as follows:

$$\eta_\ell \sim \eta_r \wedge \eta_\ell \rightarrow \eta'_\ell \Rightarrow \exists \eta'_r,\ \eta'_\ell \sim \eta'_r \wedge \eta_r \rightarrow \eta'_r \qquad (1)$$

Essentially, if the original and transformed programs are in a pair of related states, and the original program steps, then the transformed program will also step, in such a way that the two resulting states will be related. Furthermore, if the original states of the two programs are related by $\sim$, then the above condition guarantees




| | |
|---|---|
| A(σℓ , σr) | σℓ=σr ∧ eval(σℓ , I < E) ∧ eval(σr , I < E-1) |
| B(σℓ , σr) | σℓ=σr ∧ eval(σℓ , I < E) ∧ eval(σr , I ≥ E-1) |

**Figure 3.** Simulation relation for loop peeling

through an inductive argument over program traces that the two program always executes in lock step on related states.

Figure 3 shows $G_\ell$ and $G_r$ for loop peeling, and shows the simulation relation that PEC automatically infers for this example. $G_\ell$ and $G_r$ are shown in CFG form, where a node is a program point, and edges are statements. A dashed edge between $G_\ell$ and $G_r$ indicates that the program points being connected are related in the simulation relation. Furthermore, each dashed edge is labeled with a formula showing how the heaps $\sigma_\ell$ and $\sigma_r$ (of $G_\ell$ and $G_r$) are related at those program points.

The entry and exit points are related with state equality ($\sigma_\ell = \sigma_r$), which means that the simulation relation shows that if $G_\ell$ and $G_r$ start in equal states, then they will end in equal states (if the exit points are reached). Aside from the entry points, there are two other entries in the simulation relation, labeled with formulas $A$ and $B$ in Figure 3 (shown below the CFGs). The notation $\texttt{eval}(\sigma, e)$ represents the result of evaluating expression $e$ in heap $\sigma$.

The PEC checker takes as input the rewrite rule shown in Figure 2, and it automatically generates the relation shown in Figure 3. After generating this relation, PEC checks that the relation satisfies the properties required for it to be a simulation relation, namely property (1). PEC does this by enumerating the paths from each simulation relation entry to other entries that are reachable. In this case, there are five such paths: entry to $A$, entry to $B$, $A$ to $A$, $A$ to $B$, and $B$ to exit. While enumerating paths, PEC prunes infeasible ones. For example, PEC prunes the path "$A$ to exit", because the simulation entry at $A$ tells us that $\mathbf{I} < \mathbf{E} - 1$, which after executing **I++** in the original program gives $\mathbf{I} < \mathbf{E}$, which forces the original program to go back into the loop. For each feasible path that PEC enumerates, PEC shows using an automated theorem prover (more specifically an SMT solver) that if the original and transformed programs start executing at the beginning of the path, in related heap states, then they end up in related heap states at the end of the path. One important property of the simulation relation is that all loops are cut, and so there are no loops between entries in the simulation relation. As a result, the SMT solver only has to reason about short sequences of straight line code, which SMT solvers do very well in a fully automated way.

***Guarantee provided by PEC*** The PEC work [7] initially considered the following as its correctness guarantee: starting with any initial heap $\sigma$, if the original program executes to its exit and yields heap $\sigma'$, then the transformed program will also execute to its exit and produce the same $\sigma'$. However, as we will show in Section 3, this fails to capture the correctness guarantee that PEC in fact provides for non-terminating computations. As a result, to integrate PEC within CompCert and prove the PEC execution engine correct, particularly for non-terminating computations, we will have to update the interface of the PEC checker so that it also returns the simulation relation it discovered.

The techniques we present in this paper work for the "Relate" module from PEC, which accounts for about three quarters of the optimizations presented in [7]. The remaining optimizations, which include some of the more sophisticated loop optimizations like loop reversal, are handled by the PEC "Permute" module, which presents additional challenges that we leave for future work.

### 2.2 CompCert

We now give a brief overview of the CompCert [9] compiler. CompCert takes as input Clight, a large subset of C, and produces PowerPC or ARM assembly. The compiler is implemented inside the Coq proof assistant. CompCert is organized into several stages that work over a sequence of increasingly detailed intermediate representations (IRs): from various C-like AST representations, through CFG based representations like `RTL`, and finally down to abstract syntax for PowerPC assembly.

CompCert is accompanied by a proof of correctness, also implemented in Coq. This proof provides a strong end-to-end correctness guarantee. The guarantee is *strong* because the entire proof is formalized in Coq, not leaving any parts to a paper-and-pencil proof. The guarantee is *end-to-end* because it covers all the steps of compilation, from the source language all the way to assembly code.

The proof is organized around CompCert's compilation stages. For each stage, there is a proof showing that if the input program to the stage has a certain behavior, then the program produced by the stage will have the same behavior. The particular details of how each proof is done depends on the particular stage and the semantics of the input and output IR for the stage. The individual proofs are then composed together to produce an end-to-end correctness argument.

A common strategy used in CompCert for proving optimizations correct is to use a simulation relation. For each optimization that the programmer wants to add, the programmer must carefully craft a simulation relation for the optimization, and prove that it satisfies property (1) in Coq. Once this is done, CompCert has several useful theorems about small-step semantics that allows the programmer to conclude that the semantics is preserved by the optimization.

In general, proving property (1) requires a substantial amount of manual effort, and more importantly, it requires in depth knowledge of Coq, CompCert's data-structures, and proof infrastructure provided by CompCert. In contrast, in the PEC system, once the checker has been implemented, new optimizations can be checked for correctness fully automatically, with no manual proof effort.

## 3. XCert: CompCert + PEC

We have seen in Section 2.1 how PEC provides extensibility, and in Section 2.2 how CompCert provides strong guarantees. We now give an overview of how XCert extends CompCert with PEC to get both extensibility and a strong correctness guarantee. This section gives a high-level informal description of the approach, whereas Section 4 will describe the formalism as implemented in Coq.

Our general approach is to implement an execution engine for PEC optimizations in CompCert, and prove that this execution en-

gine preserves semantics, given that the optimizations being executed have successfully been checked using PEC.

### 3.1 Execution engine

To add a PEC engine to CompCert, we must decide where in CompCert's compilation pipeline the PEC engine should be added. Although there are many different points in the pipeline, each using a different IR, the decision really comes down to picking between a CFG-based IR and an AST-based IR.

We decided to apply PEC optimizations to the `RTL` intermediate representation, which is CompCert's highest level CFG-based IR. This is also the IR over which CompCert's primary optimizations work: the `RTL` stage in the compilation pipeline is perfectly suited for implementing general optimizations because all of the source language constructs have been compiled away, but none of the target specific details have yet been introduced. Although running PEC optimizations on a CFG has many benefits, it also presents several challenges.

***Pattern matching*** First, pattern matching is more difficult on a CFG than an AST. At a high-level, given a rewrite rule $G_\ell \longmapsto G_r$ **where** $S$, the PEC execution engine must find occurrences of $G_\ell$ in the program being optimized. An AST pattern-matcher is quite simple to implement recursively using a simultaneous traversal over the pattern and the expression being matched. A CFG pattern matcher, on the other hand, is more complex, primarily because CFGs can have cycles, whereas ASTs are acyclic. Not only does this make the pattern matcher itself more complex, but reasoning about it formally also becomes more difficult.

***Verified Validation*** To address the challenge of reasoning about a CFG-based pattern matcher, we make use of *Verified Validation*, a technique inspired by the work of Tristan et al. on verified translation validation [15–17]. The insight is that the result checker for an algorithm is often much simpler than the algorithm itself, and so proving the result checker correct is often much simpler than proving the algorithm correct. In our context, *Verified Validation* allows us to produce matches that are guaranteed to be correct, while only reasoning about a pattern-match result checker, rather than the pattern matcher itself.

***Transforming the CFG*** The second challenge in executing PEC optimizations on a CFG is that a CFG is more difficult to transform than an AST, and this difficulty is reflected in the Coq proof of correctness. Because ASTs are trees with no cycles or sharing, one can easily perform transformations locally, replacing a whole subtree with another subtree. In a CFG, however, replacing one subgraph with another requires appropriately connecting incoming and outgoing edges for the region that has been replaced.

To make this task as easy as possible, we take advantage of the way that CFGs are represented in CompCert. A CFG in CompCert is a map from program points to instructions, and each instruction contains successor program points. For example, a branch instruction would contain two successor program points, whereas a simple assignment would only contain one successor program point. Consider for example the original CFG shown in Figure 4(a), with a matched region of the CFG that we want to transform. We graphically display each entry in a CompCert CFG as a box that is subdivided into two parts: the left part of the box contains a program point $p$ and the right part the instruction $i$ that the program point is mapped to. We use arrows from an instruction directly to its successor program points.

***Side Conditions*** As noted in Section 2.1, PEC relies on the execution engine to provide correct implementations for a fixed set of side conditions predicates, which are used to create the side conditions of the PEC rewrite rules. For achieving a strong correctness guarantee, it is crucial that the implementation of these side condition predicates be verified. To this end, we have implemented and verified a handful of side condition predicates, e.g. *NotMod* and *StrictlyPos* from Figure 2.

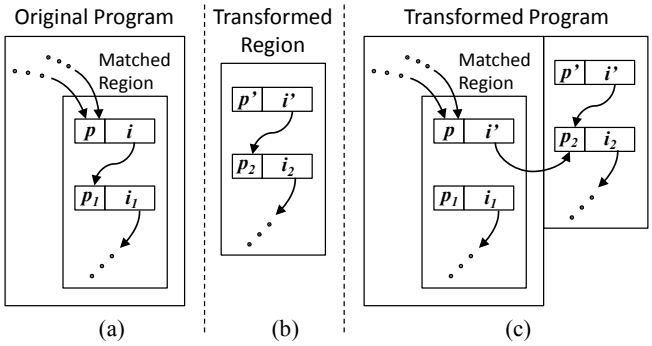



**Figure 4.** Example of CFG splicing

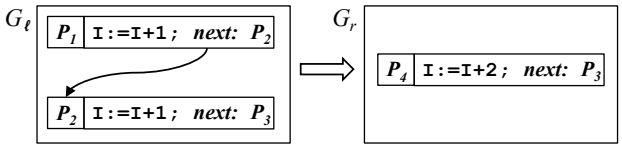



**Figure 5.** PEC rewrite rule using Parameterized CFGs

***Parametrized CFGs*** Given a PEC rewrite rule $G_\ell \longmapsto G_r$ **where** $S$, we represent $G_\ell$ and $G_r$ as *parametrized* CFGs. A parametrized CFG (PCFG) is a CompCert CFG that can contain pattern variables like $\mathbf{S}$, $\mathbf{E}$, and $\mathbf{I}$, which must be instantiated to get a concrete CFG. Furthermore, these PCFGs also use pattern variables wherever a program point would be expected. Thus, when the PEC engine finds a match for the loop-peeling rewrite from Figure 2, the resulting substitution not only states what $\mathbf{S}$, $\mathbf{E}$, and $\mathbf{I}$ map to, but also how the program points of $G_\ell$ map to program points of the CFG being transformed.

For example, Figure 5 shows how the rewrite rule $\mathbf{I}$`++;`$\mathbf{I}$`++` $\longmapsto$ $\mathbf{I}$`+=2` would be represented using PCFGs. Note that the transformed PCFG, namely $G_r$, contains a program point pattern variable $\mathbf{P}_4$ that is not bound in the original PCFG, namely $G_\ell$. Such unbound pattern variables (of which there can be many in the transformed PCFG) represent fresh program points that the engine will need to generate when it applies the transformation. Although in general it's perfectly legal for two pattern variables to map to the same piece of concrete syntax, these unbound program points have a special semantics, in that the engine generates a fresh (and thus distinct) program point for each unbound program point pattern variable.

For simplicity of presentation, we will assume that all parametrized program points in the domain of $G_r$ (i.e. program points in the left parts of the boxes in the diagrams) must be free, in that they do not appear in $G_\ell$. This makes the example easier to understand intuitively and slightly simplifies the formalization in Section 4. Our actual implementation in Coq does not make this assumption.

***Connecting outgoing edges*** To see how we connect edges leaving the transformed region, let's take a look at Figure 5 again. Note that the transformed PCFG uses the pattern variable $\mathbf{P}_3$, which *is* bound in the original PCFG. Thus, when the PEC engine finds a match for $G_\ell$ in Figure 5, the resulting substitution will have an

entry for $\mathbf{P}_3$, which essentially captures the fall-through of the matched region of code. When the engine applies this substitution to $G_r$, to produce the transformed region of code, $\mathbf{P}_3$ will be replaced with the fall-through of the original region. In this way, the regular match-and-transform process in the PEC engine naturally connects outgoing edges in the transformed region, without requiring a special case.

***Connecting incoming edges*** For connecting edges entering the transformed region, let's go back to Figure 4(a), and suppose the pattern matcher has found a sub-CFG $g_\ell$ in the original CFG that matches $G_\ell$, and let's assume that the resulting substitution is $\theta$. Furthermore, suppose that applying $\theta$ to $G_r$ produces the replacement CFG shown in Figure 4(b). As mentioned previously, the engine generates new fresh program points in the transformed CFG, which means that we can simply union the CFG from Figures 4(a) and (b) without any name clashes in the program points. Furthermore, after this union is performed, outgoing edges of the replacement CFG are already connected, as mentioned previously. As a result, we are only left with connecting the incoming edges.

Our approach to doing this is simple yet effective. In particular, we take the entry program point in the matched region from Figure 4(a) and *update* the instruction at that point with the first instruction of the replacement region from Figure 4(b). Figure 4(c) shows the result of this process. In essence, instruction $i'$ has been copied to the entry of the matched region, and since $i'$ contains inside of it all its successor program points, the instruction at $p$ now has successor links pointing directly into the transformed region. The remainder of the original matched region is left unchanged, although disconnected (except if there are other entry points into the matched region). Any unreachable code will be removed by a subsequent dead code elimination phase. Note that in our example, the program point $p'$ is also left disconnected, but this does not have to be the case in general, since instructions from the transformed region may point to it (for example, in the case of a loop).

***Witness Propagation*** In general, applying the PEC guarantee within the Coq correctness proof of the execution engine is challenging and tedious because it requires knowing information outside the engine about tests performed deep within the engine. To facilitate the task of applying the PEC guarantee, we use *Witness Propagation*, a technique in which functions are made to return additional information that is used only for reasoning purposes. For example, we make the PEC execution engine in CompCert return not only the final transformed CFG, but also the substitution that was used to generate this transformed CFG. When executing the compiler, the substitution is not used outside the engine; however, in the proof it makes applying the PEC guarantee much easier, and it simplifies case analysis for code that calls the execution engine.

### 3.2 Correctness

Recall that optimizations at the `RTL` level are proved correct in CompCert using a simulation relation, and this amounts to showing property (1) in Coq, where $\eta_\ell$ and $\eta'_\ell$ are states in the original program, and $\eta_r$ and $\eta'_r$ are states in the program produced by the PEC execution engine. When performing this proof in Coq, we assume that all the rules executed by the engine have been checked successfully by PEC, and therefore, we know that the correctness condition provided by PEC holds for those rules (outlined in Section 2.1).

One of the challenges that comes up in performing this proof is that the original program and the transformed program don't execute in perfect synchrony anymore with respect to the small-step semantics $\rightarrow$: given a piece of code that has been transformed, it may take, say, 5 steps to go through it in the original program, and only 2 steps in the transformed one. This misalignment in the semantics means that, strictly speaking, property (1) does not hold. Although CompCert has stuttering variations of (1) that can be used in this case, using these variations makes the proof more complex, but more importantly it also conflates issues: the proof would have to deal at the same time with the misalignment of semantics, and with the complexities of reasoning about PEC rewrites.

***Semantics alignment*** To separate these concerns, and to modularize the proof, we introduce two new semantics for the purposes of *Semantics Alignment*, $\rightarrow_\ell$ and $\rightarrow_r$, which are meant to align exactly: each step taken by $\rightarrow_\ell$ should correspond to precisely one step of $\rightarrow_r$, making it easier to show the equivalence of $\rightarrow_\ell$ and $\rightarrow_r$. In a separate *Semantics Alignment* module, we can then show the equivalence between $\rightarrow$ and $\rightarrow_\ell$ for the original program, and between $\rightarrow_r$ and $\rightarrow$ for the transformed program.

Our first attempt at defining $\rightarrow_\ell$ and $\rightarrow_r$ unfortunately was not strong enough. In particular, we stated that $\rightarrow_\ell$ and $\rightarrow_r$ act like $\rightarrow$, but step "over" any regions of code transformed by PEC in the original or optimized programs, respectively. Although this approach works well for terminating computations, non-terminating computations introduce additional challenges. When CompCert proves that an optimization preserves behavior, the definition of behavior includes the possibility of running forever (with a infinite trace of externally visible events, such as calls to `printf`). Thus, we need to prove that the PEC engine preserves non-terminating behaviors (including the details of the infinite trace). In general formally reasoning about the preservation of non-termination has proven challenging in the context of formally verified compilers. Indeed, many verified compilers, for example the recent work of Chlipala [3], still don't have a proof that non-termination is preserved.

***The big-step problem*** The problem with our original definition of $\rightarrow_\ell$ and $\rightarrow_r$ in regards to non-termination is that they take a big step over regions that PEC has transformed, and such a big step does not provide a guarantee when the program gets into an infinite loop inside these "stepped over" regions. The checks that PEC performs does however guarantee that non-termination is preserved inside of the regions it transforms. Thus, one way to address this problem is to strengthen the original guarantee provided by the PEC work (stated in Section 2.1), using a similar approach to what CompCert does at the optimization level: define the behavior of a region of code as either "terminates" or "runs forever". The guarantee that PEC provides would then state that the behavior of a region transformed by PEC is preserved, which would include the "runs forever" case.

While pursuing this approach, we realized that the proof was getting unwieldy. Applying the new PEC correctness guarantee was difficult because in the non-terminating case, CompCert requires the proof to produce the infinite trace in the transformed program, which in turn requires a lot of accounting to properly "glue" traces together. The complexity is in part due to the fact that different kinds of traces must be glued together: finite (inductively defined) traces with infinite (co-inductively defined) traces.

By carefully observing the challenges in the proof, we realized that, in the end, all the problems stemmed from a single mismatch in the semantics: big-step vs. small step. The CompCert `RTL` theory works using a small-step semantics, and our "step-over" approach essentially introduces a big step over potentially non-terminating computations.

***Changing the PEC interface*** Our solution to this problem is another instance of the *Semantic Alignment* technique, where we essentially change the PEC interface so that it aligns with CompCert's small-step proofs. The key to achieving this alignment stems from the realization that PEC actually performs its checking using small steps. In particular, the simulation relation that PEC generates has the property that there are no loops between entries. If there is a loop, PEC will generate an entry in the simulation relation that cuts

the loop into acyclic paths, in much the same way that a loop invariant cuts loops in program verification. Entry $A$ in Figure 3 is such a loop-cutting entry in the simulation relation. Therefore, there is no possibility that a program will not terminate *between* simulation relation entries. Furthermore, PEC uses a simulation relation in its checking, which is precisely the technique used in CompCert too. It would therefore make sense to change the interface between the two systems to take advantage of their similarities.

To this end, we modify the interface between PEC and CompCert so that the PEC checker *returns* the simulation relation that it used to prove a particular optimization correct, and we import this simulation relation into CompCert. When we prove that running this optimization in CompCert using the PEC execution engine preserves behavior, we can make use of CompCert's simulation relation approach, by creating a simulation relation for the entire program as follows: if we're *not* in a region that has been transformed, use state equality; if we *are* in a region that has been transformed, use the simulation relation returned by the PEC checker for that optimization.

Furthermore, along with the PEC simulation relation, we assume that the PEC checker returns a Coq proof that the simulation relation satisfies the simulation property, namely property (1). This proof is nothing more than a Coq reification of the proofs that PEC's SMT solver performed. If PEC used an SMT solver that returned proofs, it could perform a translation from the SMT proofs into Coq's proof language. The proof returned by PEC is used in our proof to show that the simulation relation we created for the entire program is preserved while inside transformed code.

Function calls are handled in CompCert using small steps, so that a call instruction transfers execution to the CFG of the callee. If a call instruction occurs inside the transformed region, we consider the call to essentially leave the transformed region. As a result, inside the callee, the simulation relation we construct will simply use state equality, not the PEC simulation relation. Once the call returns, execution comes back into the transformed region, and the simulation relation we construct goes back to using the PEC simulation relation.

***Left and right semantics, revisited*** Now that PEC returns a simulation relation, we can give the definitions of $\rightarrow_\ell$ and $\rightarrow_r$ that we use in our proof: if we're *not* in a region that has been transformed, $\rightarrow_\ell$ and $\rightarrow_r$ work the same as $\rightarrow$; if we *are* in a region that has been transformed, $\rightarrow_\ell$ and $\rightarrow_r$ simply step from one entry to another in the simulation relation returned by PEC.

To illustrate how $\rightarrow$, $\rightarrow_\ell$ and $\rightarrow_r$ work, Figure 6 shows part of an execution trace $trace_\ell$ for the original program (with round circles for program states), and part of a trace $trace_r$ for the transformed program (with crosses for program states), along with the simulation relation as it unfolds throughout execution (shown as dotted edges between the original and transformed traces). The simulation relation inside the transformed region is the one that PEC returns. Figure 6 also shows how the three step semantics operate on the original and transformed programs: $\rightarrow$ and $\rightarrow_\ell$ on the original program and $\rightarrow_r$ and $\rightarrow$ on the transformed program.

### 3.3 Proof architecture

To summarize, our proof is therefore organized into three steps, which we show separately: (1) if a program $\pi$ has behavior $b$ under $\rightarrow$, then $\pi$ has behavior $b$ under $\rightarrow_\ell$; (2) if a program $\pi$ has behavior $b$ under $\rightarrow_\ell$, then the program produced by our execution engine on $\pi$ has behavior $b$ under $\rightarrow_r$; (3) if a program $\pi$ has behavior $b$ under $\rightarrow_r$, then $\pi$ has behavior $b$ under $\rightarrow$. Steps (1) and (3) are where semantics alignment issues are resolved, and step (2) is where we build a simulation relation for the original and transformed programs using the simulation relation returned by the PEC checker.

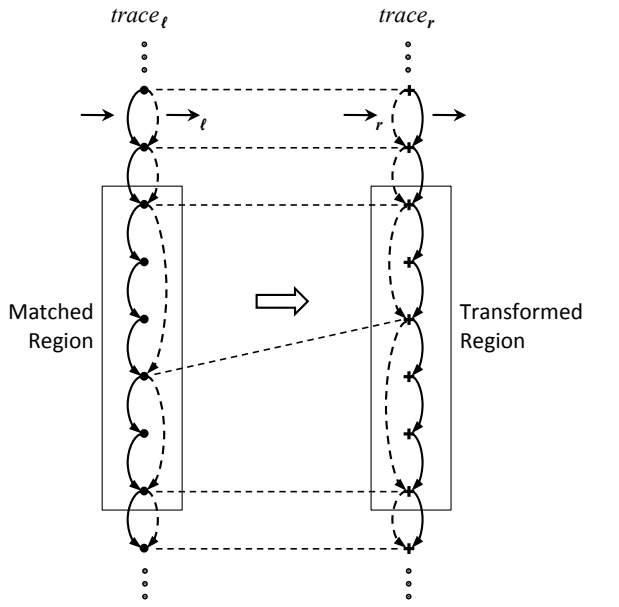



**Figure 6.** Traces showing how $\rightarrow$, $\rightarrow_\ell$ and $\rightarrow_r$ work

| | | | | | |
|---|---|---|---|---|---|
| Instruction | $i$ | $\in$ | $Instr$ | | |
| Program point | $p$ | $\in$ | $PP$ | | |
| CFG | $g$ | $\in$ | $CFG$ | $=$ | $PP \rightharpoonup Instr$ |
| Program | $\pi$ | $\in$ | $Prog$ | $=$ | $String \rightharpoonup CFG$ |
| Program heap | $\sigma$ | $\in$ | $Heap$ | | |
| Program state | $\eta$ | $\in$ | $State$ | $=$ | $CFG \times PP \times Heap$ |
| PEC Sim Rel | $\psi$ | $\in$ | $Sim$ | $=$ | $\mathcal{P}(State \times State)$ |
| Substitution | $\theta$ | $\in$ | $Subst$ | | |
| Param. $Sim$ | $\Psi$ | $\in$ | $PSim$ | | |
| Param. $CFG$ | $G$ | $\in$ | $PCFG$ | | |
| Side condition | $S$ | $\in$ | $SC$ | $=$ | $CFG \times CFG$ |
| Rewrite rule | $r$ | $\in$ | $Rule$ | $=$ | $PCFG \times PCFG \times SC$ |

**Figure 7.** Common types used in our formalism

## 4. Formalization

In this section, we make the ideas from Section 3 more precise, by presenting a formalization of the PEC engine and its proof. The development presented here closely mirrors our implementation in Coq. Later, in Section 5, we describe some of the additional challenges that arose when translating these high level ideas into Coq code.

### 4.1 Basic definitions

We start with some basic definitions, shown in Figure 7. An instruction $i$ may be any one of a number of basic `RTL` instructions already defined in CompCert. A CFG $g$ is a map from program points to instructions, and a program is map from function names (strings) to CFGs. A program heap $\sigma$ contains the state of dynamically allocated memory blocks. For simplicity of presentation, we assume the heap also contains the state of the registers and stack, even though in the implementation they are kept separate. A program state $\eta$ consists of a CFG (representing the current code being executed), a program point in that CFG (representing where execution has reached), and the heap (which includes the stack). We project

these fields of a program state $\eta$ as follows: $\mathsf{g}(\eta)$ denotes the CFG, $\mathsf{p}(\eta)$ denotes the program point, and $\mathsf{s}(\eta)$ denotes the heap.

A PEC simulation relation $\psi$ is a relation over program states that is returned by the PEC checker. Because they are generated by PEC, these simulation relations have entries for related program points, and each entry is a predicate over program heaps (recall Figure 3). Therefore, such relations have the form:

$$\psi((g_\ell, p_\ell, \sigma_\ell), (g_r, p_r, \sigma_r)) \triangleq \psi_P(p_\ell, p_r)(\sigma_\ell, \sigma_r)$$

where $\psi_P \in (PP \times PP) \rightharpoonup \mathcal{P}(Heap \times Heap)$. We use the notation $p \in \psi$ to denote that $p$ is in the domain of $\psi_P$ (either as a first parameter or second parameter).

A substitution $\theta$ is a map from pattern variables to concrete pieces of syntax. A parametrized simulation relation $\Psi$ is a version of a simulation relation that contains pattern variables which must be instantiated to yield a concrete simulation relation. For example, the simulation relation shown in Figure 3 is parametrized because syntactic values for $\mathbf{S}$, $\mathbf{E}$, and $\mathbf{I}$ must be provided before the simulation relation can apply to concrete program states. Given a parametrized simulation relation $\Psi$, and a substitution $\theta$ that maps every free pattern variable in $\Psi$ to concrete syntax, the result of applying $\theta$ to $\Psi$, denoted $\theta(\Psi)$, is a concrete simulation relation $\psi$. Similarly, a parameterized CFG $G$ is a parametrized version of a CFG. A side condition is a boolean function from two concrete CFGs (here expressed as a relation). A PEC rewrite rule $r$ contains two parametrized CFGs (representing the pattern to match, and the replacement to perform), and a side condition.

### 4.2 PEC checker and guarantee

PEC takes a rewrite rule and attempts to construct a parameterized simulation relation. If PEC is able to check that the rewrite rule is correct, it also returns a proof that the simulation relation satisfies the simulation property. Specifically, PEC has the type:

$$\mathsf{PEC}(r : Rule) : (\Psi : PSim \times \mathsf{Proof}[\mathsf{IsSimRel}(r, \Psi)]) \cup \{\mathsf{Fail}\}$$

The proof returned by PEC plays a central role in our Coq proof of the correctness of the execution engine. To describe the proof returned by PEC we'll make use of a modified step relation, $\eta \xrightarrow{t}_{\psi} \eta'$, which essentially steps over any program points not in the PEC simulation relation $\psi$. That is, $\xrightarrow{t}_{\psi}$ combines the sequence of regular $\xrightarrow{t}$ steps from one entry in $\psi$ to the next into a single "medium" step.

Using this definition, we now define $\mathsf{IsSimRel}(r, \Psi)$, the guarantee provided by the proof term returned by PEC:

DEFINITION 1. *We say* $\mathsf{IsSimRel}((G_\ell, G_r, S), \Psi)$ *holds iff:*

$$S(\theta(G_\ell), \theta(G_r)) \Rightarrow \mathsf{IsConSimRel}(\theta(\Psi), \theta(G_\ell), \theta(G_r))$$

*where* $\mathsf{IsConSimRel}(\psi, g_\ell, g_r)$ *holds iff:*

$$\psi_P(\mathsf{Entry}(g_\ell), \mathsf{Entry}(g_r)) = \mathsf{HeapEq} \wedge$$

$$\psi_P(\mathsf{Exit}(g_\ell), \mathsf{Exit}(g_r)) = \mathsf{HeapEq} \wedge$$

$$\left[\begin{array}{l} g_\ell = \mathsf{g}(\eta_\ell) \wedge g_r = \mathsf{g}(\eta_r) \wedge \\ \psi(\eta_\ell, \eta_r) \wedge \eta_\ell \xrightarrow{t}_{\psi} \eta'_\ell \end{array}\right] \Rightarrow \left[\exists \eta'_r \,.\, \psi(\eta'_\ell, \eta'_r) \wedge \eta_r \xrightarrow{t}_{\psi} \eta'_r\right]$$

Intuitively, the above definition guarantees that the simulation relation returned by PEC: (a) relates states on entry and exit to $G_\ell$ and $G_r$ by heap equality – HeapEq is defined by $\forall\sigma.\mathsf{HeapEq}(\sigma, \sigma)$; and (b) satisfies the simulation property (1).

### 4.3 Execution engine

Figure 8 shows pseudo code for the PEC execution engine in XCert. Given a program $\pi$ and a PEC rewrite $r$, TrProg applies $r$ to each CFG in $\pi$ using TrCFG. It projects the first element of the result of TrCFG because it contains both the transformed CFG and the substitution used to produce this CFG. TrCFG iterates over all the program points in the given CFG $g$, and for each program point it attempts to apply the rewrite starting at that point by calling TrPoint. It gathers the resulting CFGs and chooses one as the transformed version of $g$.

```
TrProg(π, r) :
  return λs.
    fst(TrCFG(π(s), r))

TrCFG(g, r) :
  C ← ∅
  for p ∈ ProgPoints(g) do
    x ← TrPoint(g, r, p)
    C ← C ∪ {x}
  return Pick(C)

TrPoint(g_ℓ, (G_ℓ, G_r, S), p) :
  θ ← Match(G_ℓ, g_ℓ, p)
  if ¬ θ(G_ℓ) ≙^p g_ℓ
    return (g_ℓ, ⊥)
  θ ← Fresh(θ, G_r)
  if ¬ S(θ(G_ℓ), θ(G_r))
    return (g_ℓ, ⊥)
  g_r ← g_ℓ ∪ θ(G_r)
  i ← g_r(θ(G_r.entry))
  g_r ← g_r[p ↦ i]
  return (g_r, θ)
```

**Figure 8.** PEC execution engine

TrPoint first tries to match the left parameterized CFG $G_\ell$ of the rewrite rule to the given concrete CFG $g_\ell$. It then checks that any generated substitution $\theta$ applied to $G_\ell$ is identical to the CFG fragment of $g_\ell$ rooted at $p$; we denote this as $\theta(G_\ell) \stackrel{p}{=} g_\ell$. If this check or the pattern match fails, TrPoint simply returns the original CFG and $\perp$ which indicates an invalid substitution.This instance of *Verified Validation* allows us to avoid reasoning about Match directly and instead simply show that our comparison $\stackrel{p}{=}$ is correct, which is a much smaller proof burden. Next TrPoint creates fresh program points for any parameterized program points that are free in $G_r$. Now, TrPoint checks that the rewrite rule's side condition holds on the CFGs generated by applying $\theta$ to the left and right parameterized programs, $G_\ell$ and $G_r$. Once again, if the check fails, TrPoint simply returns the original CFG and $\perp$. Next TrPoint generates the transformed version of the code $g_r$ by applying the substitution $\theta$ to the right parameterized CFG $G_r$. TrPoint then changes $g_r$ so that program location $p$ points to the first instruction of the transformed part of the CFG. Finally, TrPoint returns the transformed CFG $g_r$ and the substitution $\theta$.

### 4.4 Correctness condition

We define the set of behaviors of a program as follows:

$$Beh = \{\mathsf{term}(t) \mid t \in Trace\} \cup \{\mathsf{forever}(t) \mid t \in Trace\}$$

where $t$ represents a potentially infinite trace of observable events, and $\mathsf{term}(t)$ and $\mathsf{forever}(t)$ respectively denote executions terminating or diverging with a trace $t$. We use $\pi \Downarrow b$ to indicate that $\pi$ has behavior $b$, as defined below.

DEFINITION 2. *The relation* $\pi \Downarrow b$ *is defined as follows:*

- if $\eta_i(\pi) \xrightarrow{t}{}^{*} \eta_f$ and $\eta_f \in Final$ then $\pi \Downarrow \mathsf{term}(t)$
- if $\eta_i(\pi) \xrightarrow{t}{}^{\infty}$ then $\pi \Downarrow \mathsf{forever}(t)$

*where:* $\eta_i(\pi)$ *is the initial state of program* $\pi$*;* $\xrightarrow{t}{}^{*}$ *is the reflexive transitive closure of* $\xrightarrow{t}$*;* $Final$ *is the set of final program states (indicating program termination); and* $\eta \xrightarrow{t}{}^{\infty}$ *indicates that execution runs forever producing trace* $t$ *under* $\rightarrow$ *when started at* $\eta$*.*

To show the correctness of our execution engine, we prove the following theorem in Coq:

THEOREM 1. *If* $\mathsf{PEC}(r) \neq \mathsf{Fail}$ *and* $\pi \Downarrow b$ *then* $\mathsf{TrProg}(\pi, r) \Downarrow b$.

In the following, we describe a Coq proof of Theorem 1. To do this, we fix a particular rule $r$ and assume $\mathsf{PEC}(r) = (\Psi, \rho)$, where $\Psi$ is the parametrized simulation relation found by PEC for $r$ and $\rho$ is a proof of $\mathsf{IsSimRel}(r, \Psi)$ (which essentially guarantees that $\mathsf{IsSimRel}(r, \Psi)$ holds).

***Step left and step right*** To simplify applying the proof $\rho$ of $\mathsf{IsSimRel}(r, \Psi)$, we construct two new, closely related semantics that are specialized to a concrete simulation relation:

DEFINITION 3. *We define* $\eta_\ell \xrightarrow{t}_\ell \eta'_\ell$ *as the smallest relation satisfying:*

$$\left[\begin{array}{rcl} \mathsf{TrCFG}(\mathsf{g}(\eta_\ell), r) & = & (\mathsf{g}(\eta_r), \theta) \\ \psi & = & \theta(\Psi) \end{array}\right] \Longrightarrow$$

$$\left[\begin{array}{ll} \mathsf{p}(\eta_\ell) \notin \theta \wedge \mathsf{p}(\eta'_\ell) \notin \theta \wedge \eta_\ell \xrightarrow{t} \eta'_\ell & \Rightarrow\ \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \\ \mathsf{p}(\eta_\ell) \notin \theta \wedge \mathsf{p}(\eta'_\ell) \in \psi \wedge \eta_\ell \xrightarrow{t} \eta'_\ell & \Rightarrow\ \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \\ \mathsf{p}(\eta_\ell) \in \psi \wedge \mathsf{p}(\eta'_\ell) \in \psi \wedge \eta_\ell \xrightarrow{t}_\psi \eta'_\ell & \Rightarrow\ \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \\ \mathsf{p}(\eta_\ell) \in \psi \wedge \mathsf{p}(\eta'_\ell) \notin \theta \wedge \eta_\ell \xrightarrow{t} \eta'_\ell & \Rightarrow\ \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \end{array}\right]$$

Note that formulas in square brackets are implicit conjunctions of formulas, one formula per line. The relation $\eta_r \xrightarrow{t}_r \eta'_r$ is defined analogously to $\xrightarrow{t}_\ell$ by substituting $\eta_r$ for $\eta_\ell$ in the right-hand side of the main implication above.

The notation $\mathsf{p}(\eta_\ell) \notin \theta$ indicates that the program point of state $\eta$ is not in a region of CFG transformed by TrCFG. This is implemented by searching $\theta$ to determine if a parameterized program point maps to $\mathsf{p}(\eta_\ell)$ such that the parameterized program point is not one of the *exit* points from the transformed code back to unmodified code. For briefness, we may speak of a state $\eta$ not being in the transformed region; this simply means $\mathsf{p}(\eta) \notin \theta$.

Intuitively, $\xrightarrow{t}_\ell$ captures distinct ways in which the original code can step from state $\eta_\ell$ to $\eta'_\ell$. In Definition 3, the first line of the main implication's right-hand side handles situations where neither $\eta_\ell$ nor $\eta'_\ell$ are in the transformed region. In this case $\eta_\ell \xrightarrow{t}_\ell \eta'_\ell$ holds whenever $\eta_\ell \xrightarrow{t} \eta'_\ell$ holds, that is whenever $\eta_\ell$ could take a normal RTL step to $\eta'_\ell$. The second and fourth lines capture entering and exiting the transformed region, which again requires $\eta_\ell \xrightarrow{t} \eta'_\ell$. Note that we only allow entering and exiting transformed code through program locations that are in $\psi$. The third line captures the situation where the original program executes from entry to entry of $\psi$ using $\rightarrow_\psi$.

Similar to the definition of $\Downarrow$ (Definition 2), we also define $\Downarrow_\ell$ and $\Downarrow_r$, which respectively use $\rightarrow_\ell$ and $\rightarrow_r$ rather than $\rightarrow$.

### 4.5 Proof architecture

To establish Theorem 1 for program $\pi$ and rewrite rule $r = (G_\ell, G_r, S)$ where $\mathsf{PEC}(r) \neq \mathsf{Fail}$, our Coq proof shows following three lemmas, which we describe in more detail below:

LEMMA 1. *If* $\pi \Downarrow b$ *then* $\pi \Downarrow_\ell b$.

LEMMA 2. *If* $\pi \Downarrow_\ell b$ *then* $\mathsf{TrProg}(\pi, r) \Downarrow_r b$.

LEMMA 3. *If* $\pi \Downarrow_r b$ *then* $\pi \Downarrow b$.

***Lemma 1*** CompCert's library for small-step semantics allows us to demonstrate Lemma 1 if we can exhibit a simulation relation $\sim_1$ and a well-founded order $<$ on program states such that:

$$\eta \sim_1 \eta_\ell \wedge \eta \xrightarrow{t} \eta' \Rightarrow \exists \eta'_\ell, \eta' \sim_1 \eta'_\ell \wedge (\eta_\ell \xrightarrow{t}_\ell \eta'_\ell \vee \eta' < \eta) \quad (2)$$

Intuitively, this is the same as the standard simulation property (1), except that we allow for the possibility that $\eta_\ell$ does not step as long as the order is decreasing from $\eta$ to $\eta'$.

We define $\eta \sim_1 \eta_\ell$ to hold when either: (a) $\eta = \eta_\ell$ and either $\eta$ and $\eta_\ell$ are outside transformed code or both are at an entry in $\psi$ or (b) $\eta$ is in a transformed region, but not at an entry in $\psi$, $\eta_\ell$ is at an entry in $\psi$, and $\eta \xrightarrow{t}_\psi \eta_\ell$. Furthermore, we define the $<$ order as follows: $\eta' < \eta$ iff $m(\eta') < m(\eta)$ where $m(\eta)$ and $m(\eta')$ are the number of steps that $\eta$ and $\eta'$ have, respectively, until reaching the next entry in $\psi$.

We now have to show condition (2). The first and simpler case corresponds to (a) in the definition of $\eta \sim_1 \eta_\ell$. Here we show that the executions are in lockstep and that the successor states $\eta'$ and $\eta'_l$ are equal. The second and more difficult case, corresponding to (b) in the definition of $\eta \sim_1 \eta_\ell$, involves accounting for the steps of $\pi$'s execution *between* entries in $\psi$. In this case: $\eta_\ell$ is at an entry in $\psi$ (because we are in case (b) of the definition of $\eta \sim_1 \eta_\ell$) and it does not step; $\eta$ is not at an entry in $\psi$ and steps to $\eta'$; and from the definition of $\sim_1$ (the second case) we know $\eta \xrightarrow{t}_\psi \eta_\ell$. Thus $\eta'$ is closer than $\eta$ to the next entry in $\psi$ (namely the program point of $\eta_\ell$), which allows us to show that $\eta' < \eta$.

***Lemma 2*** Lemma 2 is the most difficult aspect of our Coq proof. CompCert's library for small-step semantics provides a theorem which allows us to demonstrate Lemma 2 if we can exhibit a simulation relation $\sim_2$ between the states of $\pi$ and $\mathsf{TrProg}(\pi, r)$ that satisfies the following (which is essentially property (1)):

$$\eta_\ell \sim_2 \eta_r \wedge \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \Rightarrow \exists \eta'_r, \eta'_\ell \sim_2 \eta'_r \wedge \eta_r \xrightarrow{t}_r \eta'_r \quad (3)$$

DEFINITION 4. *We define* $\eta_\ell \sim_2 \eta_r$ *as the smallest relation satisfying:*

$$\left[\begin{array}{rcl} \mathsf{TrCFG}(\mathsf{g}(\eta_\ell), r) & = & (\mathsf{g}(\eta_r), \theta) \\ \psi & = & \theta(\Psi) \end{array}\right] \Longrightarrow$$

$$\left[\begin{array}{ll} \mathsf{p}(\eta_\ell) \notin \theta \wedge \mathsf{p}(\eta_\ell) = \mathsf{p}(\eta_r) \wedge \mathsf{s}(\eta_\ell) = \mathsf{s}(\eta_r) & \Rightarrow\ \eta_\ell \sim_2 \eta_r \\ \psi(\eta_\ell, \eta_r) & \Rightarrow\ \eta_\ell \sim_2 \eta_r \end{array}\right]$$

Intuitively $\sim_2$ relates states using heap equality when the program points are outside of a transformed region, and using the simulation relation returned by PEC when the program points are inside of a transformed region.

Proving condition (3) has four main cases, which correspond to the four conjuncts in the definitions of $\rightarrow_\ell$ and $\rightarrow_r$.

*Case 1:* $\eta_\ell$ and $\eta_r$ are both outside of transformed regions and so are their successor states. This case is straightforward. Because $\eta_\ell \sim_2 \eta_r$ we know their heaps and program points are equal and because they are outside of transformed code, we know they are executing the same instruction. Thus $\eta_r$ will step to $\eta'_r$ where $\mathsf{p}(\eta'_r) = \mathsf{p}(\eta'_\ell)$ and $\mathsf{s}(\eta'_r) = \mathsf{s}(\eta'_\ell)$, which implies $\eta'_\ell \sim_2 \eta'_r$ (using the first case of $\sim_2$).

*Case 2:* $\eta_\ell$ and $\eta_r$ are both stepping from outside the transformed region into the transformed region. Because both states start outside the transformed region, we know their heaps are equal and that they're executing the same instruction. Thus $\eta_r$ will step to $\eta'_r$ such that $\mathsf{s}(\eta'_\ell) = \mathsf{s}(\eta'_r)$. Furthermore, because PEC guarantees that the entries of matched code will be related in $\psi$ with heap equality (see the part of definition 1 that uses HeapEq), $\mathsf{s}(\eta'_\ell) = \mathsf{s}(\eta'_r)$ implies $\psi(\eta'_\ell, \eta'_r)$. Thus $\eta'_\ell \sim_2 \eta'_r$ (using the second case of $\sim_2$).

*Case 3:* $\eta_\ell$ and $\eta_r$ are both stepping from one entry of $\psi$ to the next. We use the fact that $\mathsf{TrCFG}(\mathsf{g}(\eta_\ell), r) = (\mathsf{g}(\eta_r), \theta)$ to invoke the guarantee provided by PEC's proof

of $\mathsf{IsSimRel}(r, \Psi)$. Specifically, $\mathsf{TrCFG}(\mathsf{g}(\eta_\ell), r) = (\mathsf{g}(\eta_r), \theta)$ implies that $S(\theta(G_\ell), \theta(G_r))$ which ensures $\mathsf{IsConSimRel}(\theta(\Psi), \theta(G_\ell), \theta(G_r))$ (see Definition 1 and TrCFG in Figure 8) . This fact ensures that the $\eta_r$ will execute to $\eta_r'$ and $\psi(\eta_\ell', \eta_r')$. Thus $\eta_\ell' \sim_2 \eta_r'$ (using the second case of $\sim_2$).

*Case 4:* $\eta_\ell$ and $\eta_r$ are both stepping from inside the transformed region to outside the transformed region. Similar to Case 2 above, PEC guarantees that exits of matched code will be related in $\psi$ with heap equality (see the part of definition 1 that uses HeapEq), meaning that $\psi(\eta_\ell, \eta_r)$ at the exit implies $\mathsf{s}(\eta_\ell) = \mathsf{s}(\eta_r)$. Furthermore, the way our pattern matching works ensures that $\mathsf{p}(\eta_\ell) = \mathsf{p}(\eta_r)$ and that the instruction at these program points are equal. Thus $\eta_r$ will step to $\eta_r'$ where $\mathsf{p}(\eta_r') = \mathsf{p}(\eta_\ell')$ and $\mathsf{s}(\eta_r') = \mathsf{s}(\eta_\ell')$. From this it follows that $\eta_\ell' \sim_2 \eta_r'$ (using first case of $\sim_2$).

***Lemma 3*** CompCert's library for small-step semantics provides a theorem which allows us to demonstrate Lemma 3 if we can show:

$$\eta_r \xrightarrow{t}_r \eta_r' \Rightarrow \eta_r \xrightarrow{t}^+ \eta_r'$$

The above follows immediately from the definition of $\rightarrow_r$.

## 5. Coping with challenges

Throughout Sections 3 and 4, we have already shown how three techniques are very useful in managing the complexity of extending CompCert to support PEC rewrite rules: *Verified Validation*, *Semantics Alignment* and *Witness Propagation*. In this section we present several additional important challenges that we faced in our development and their solutions.

### 5.1 Termination of Coq code

Functions expressed in Coq's Calculus of Inductive Constructions must be shown to terminate. In most cases, Coq can prove termination automatically by finding an appropriate measure on a function's arguments that decreases with recursive calls. However, analyses that attempt to reach a fixed point or traverse cyclic structures like CFGs often pose problems for Coq's automated termination-proving strategy. One solution to this problem is to develop a termination proof for such functions in Coq. In general this can be hard, and it also makes the functions more difficult to update, since the termination proof also needs updating.

Another solution to is the introduction of a timeout parameter that is decremented for each recursive call. If it ever reaches zero the function immediately returns with a special $\perp$ value. Using this approach, Coq can now show termination automatically. The downside of this simplistic approach is that the algorithm is now incomplete, since in some cases it can return $\perp$, and the proof of correctness needs to take this into account. However, this is not a problem in domains where there is a safe fallback return value that makes the proof go through. This is indeed the case in the compiler domain: the safe return value is the one that leads to no transformations – for example a pattern matcher can always return Fail. Although a constant timeout may appear to be crude solution at first, we have found that it presents a very good engineering trade-off, since a large timeout often suffices in practice.

### 5.2 Case explosion

Conceptually, our intermediate semantics $\rightarrow_\ell$ and $\rightarrow_r$ have only four cases, as show in Section 4. However, such definitions on paper often lead to formal Coq definitions with many cases. For example, expressing $\rightarrow_\ell$ and $\rightarrow_r$ in terms of CompCert's small-step $\rightarrow$ leads to a total of 9 cases. Most of these 9 cases use $\rightarrow$ which itself has 12 cases, leading to an explosion in the number of cases. In the end, however, only a handful of these case are actually feasible at any one point in the proof, and a paper-and-pencil proof could easily say "the only feasible cases are ...". However, the formal proof needs to handle every case, leading to complex accounting.

One approach that we have found very helpful with eliminating the many infeasible cases is to thread additional information in the return values of functions. This additional information is not used by the computation itself, but rather in the proof, to provide the right context in the callers to know how to prune appropriate cases. One example of this approach is the PEC execution engine from Figure 8, which threads the substitution found in TrPoint all the way back up to TrProg, even though for the purposes of applying PEC rules, this substitution is not needed outside of TrPoint. In other cases, we have also found that implementing specialized tactics in Coq's tactic languages allows us to easily handle many similar cases using few lines of proof.

### 5.3 Law of the excluded middle

The law of excluded middle occurs very naturally when working out high level proof sketches. Unfortunately, the constructive logic underlying Coq does not provide this luxury. As an example, one could be tempted in a proof sketch to split on termination: either execution returns from a given function call or it does not. However, this intuitive fact cannot be shown in Coq, because it would require deciding algorithmically if the function terminates. Instead one must create an inductive construct with two constructors corresponding to the intuitive case split. This is precisely how termination vs. non-termination is handled in CompCert, as shown in the definition of $\Downarrow$ (Definition 2). Alternatively, in situations where it is possible, one can implement a decision procedure that correctly distinguishes between the various cases of interest. Then, within a proof, one can perform case analysis on the result produced by this decision procedure.

## 6. Evaluation

XCert extends the CompCert verified compiler with an execution engine that applies parameterized rewrite rules checked by PEC. Below we characterize our implementation of XCert by comparing it to both an untrusted prototype execution engine and to some of the manual optimizations found within CompCert (Sections 6.1 and 6.2). Next, we evaluate XCert in terms of its trusted computing base (Section 6.3), extensibility (Section 6.4) and correctness guarantee (Section 6.5). We conclude by considering the limitations of our current execution engine (Section 6.6).

### 6.1 Engine Complexity

The PEC execution engine that we added to CompCert comprises approximately 1,000 lines of Coq code. Its main components are the pattern matching and the substitution application which allow us to easily implement the transformations specified by PEC rewrite rules.

The PEC untrusted prototype execution engine mentioned in [7] was roughly 400 lines of OCaml code. Although both execution engines apply PEC rewrite rules to perform optimizations, they work in very different settings. The CompCert execution engine targets the CFG-based RTL representation in CompCert, while the prototype in [7] targets an AST-based representation of a C-like IR.

We also compare the PEC execution engine against CompCert's two main RTL optimizations, common subexpression elimination (CSE) and constant propagation (CP). CSE is 440 lines of Coq code, and CP is 1,000 lines. Both of these optimizations make use of a general purpose dataflow solver, which is about 1,200 lines. Structurally, the PEC execution engine is very different from the optimizations in CompCert. Most of the code in the PEC engine performs pattern matching and tricky CFG splicing to achieve the task of replacing an entire region of the CFG with another. Instead, CSE and CP in CompCert perform simple CFG rewrites (one

statement to another), and instead focus their efforts on computing dataflow information.

### 6.2 Proof Complexity

The proof of correctness for our execution engine is approximately 3,000 lines of Coq proof code. This code defines (1) the intermediate semantics $\rightarrow_\ell$ and $\rightarrow_r$ that facilitate applying the PEC guarantee, (2) Coq proof scripts demonstrating the semantic preservation of transformations performed by the execution engine and (3) tactics that make developing these proofs easier and more concise.

CompCert's correctness proofs for CSE and CP each span nearly 1,000 lines of proof code. Structurally, the correctness proofs for these CompCert optimizations are quite different from the execution engine's correctness proof, because they deal with different challenges. The CSE and CP proofs are mainly devoted to extracting useful facts from the result of the dataflow analysis performed by the transformation. These facts are then used to establish sufficient conditions for semantic preservation. In contrast, the proof of the execution engine focuses on showing that the many-to-many CFG rewrites that the PEC engine performs are correct. This typically involves splitting into two cases: cases where execution is not in the transformed code, which are typically straightforward; and cases where execution is in some region that has been transformed, in which case the proof effort involves either showing the case cannot arise or the simulation relation from PEC applies.

Note that the correctness proof for the PEC execution engine is three times larger than the PEC execution engine itself. However, the engineering effort for developing the proof was at least an order of magnitude greater than the effort for developing the execution engine. This is because we re-engineered the proof several times to make it simpler, cleaner, and more manageable using tactics.

### 6.3 Trusted Computing Base

The trusted computing base (TCB) consists of those components that are trusted to be correct. A bug in these components could invalidate any of the correctness guarantees that are being provided. The TCB for the regular CompCert compiler (without the PEC engine) includes CompCert's implementation of the C semantics, Coq's underlying theory (the Calculus of Inductive Constructions), and Coq's internal proof checker.

When CompCert is extended with the PEC execution engine, the TCB grows because, even though the engine is proved correct in Coq, we trust that the PEC checker correctly checks any simulation relation it returns. Within the PEC implementation this checker is implemented in about 100 lines of OCaml code and makes calls to an SMT solver like Simplify [5] or Z3 [4]. Thus, the PEC engine adds the following to CompCert's TCB: 100 lines of OCaml for the PEC checker, an SMT solver like Simplify or Z3, and the encoding of CompCert's `RTL` semantics to be used by the SMT solver.

### 6.4 Extensibility

With this relatively small increase in TCB comes the following benefit: additional optimizations that are added using PEC *do not require any new manual proof effort, and do not add anything to the TCB*. In contrast, for each new optimization added to CompCert, unless a verified validator has already been specifically designed for it, the new optimization would either have to be proved correct, or if not, it would be trusted, thus increasing the TCB. Thus, the provably correct PEC execution engine brings all of the expressiveness and extensibility shown previously in [7] to CompCert while adding only a small amount to the TCB.

To test the extensibility of our system, we implemented and ran all the optimizations checked by PEC's "Relate" module in [7]. We ran the optimizations on an array of CompCert C benchmarks totaling about 10,000 lines of code. The benchmarks included cryptographic code like AES and SHA1, numeric computations such as FFT and Mandelbrot, and a raytracer. We manually checked that the transformations were carried out as expected.

### 6.5 Correctness Guarantee

While the size of the TCB tells us how much needs to be trusted, it is also important to evaluate the correctness guarantee provided in exchange for this trust. Essentially, the CompCert compiler extended with our PEC execution engine provides the same guarantee as the original CompCert compiler: *if* the compiler produces an output program, then the output program will be semantically equivalent to the corresponding input program.

There are two ways in which this guarantee is not as strong as one may hope for. First, CompCert extended with our PEC execution engine is not guaranteed to produce an output program, even on a valid input program, because some passes from CompCert may abort compilation. For example, during the stack layout phase of CompCert, if a program spills too many variables and exceeds the available stack for a given function, then CompCert is forced to abort without producing an assembly output program. However, the PEC engine itself always produces an output program, and therefore is not a source of incompleteness.

The other weakness in the PEC engine's correctness guarantee is shared by all systems that use verified validation. In particular, those parts of the system that are checked using verified validation may still contain bugs in them. For example, the initial version of our PEC execution engine did not always correctly instantiate fresh nodes for the RHS of a PCFG. However, when this bug was exercised, our verified validator detected that the generated nodes did not have the required freshness property, and prevented the incorrect transformation from being performed. Such bugs therefore manifest themselves not as violations of the input/output equivalence guarantee, but as missed optimization opportunities. The existence of such quality-of-optimization bugs emphasizes the value of having run our PEC engine on real code, as described in Section 6.4, and ensuring that the optimizations operate as expected.

### 6.6 Limitations

The PEC checker is currently not implemented in Coq. Thus, for each PEC rewrite rule $r$, we must translate by hand the simulation relation produced by the PEC checker for $r$ into a Coq term and axiomatize its correctness proof. We intend to develop a version of PEC that directly outputs these simulation relations as Coq terms. Eventually, we plan to also implement all of PEC in Coq and thus eliminate the disconnect between the two systems.

Our current version of parameterized statements like $\mathbf{S}$ in Figure 2 are only able to match fixed length sequences of arbitrary instructions. Although this allows us to simulate parameterized statements of a fixed size, we must properly implement parameterized statements to achieve the full expressiveness of PEC.

## 7. Future Work

There are several directions for future work that we intend to explore. First, we plan to systematically and thoroughly compare the quality of existing CompCert optimizations with their corresponding PEC versions. Our evaluation will consider the runtime performance of generated code and the number of missed optimization opportunities. This comparison will enable us to fine tune our PEC optimizations and execution engine which, eventually, we hope will match the optimization capabilities currently found in CompCert. More broadly, we will also evaluate the relative effort of adding optimizations using XCert versus coding them directly in Coq or within other optimization frameworks.

We also plan to explore further reductions to the TCB. When our PEC execution engine is added to CompCert, the TCB grows

because the PEC checker becomes trusted. However, if we reimplement the PEC checker in Coq and formally prove its correctness, then our PEC engine would not at all increase the size of the TCB. The core of the PEC checker consists of only 100 lines of stateless OCaml code, which we anticipate will be easy to implement and reason about in Coq. However, this core checker makes queries to an SMT solver (like Z3) which could be challenging to integrate into Coq. Fortunately, there are several reasons to be optimistic. First, some SMT solvers have recently been re-engineered to produce proof terms, which we should be able to automatically translate to Coq terms and thus integrate into a Coq proof (possibly using the Coq Classical extension to accommodate for the refutation based proof strategies common in SMT solvers). Second, the PEC checker's SMT queries tend to be simple and highly stylized. Thus, it may instead be possible to implement a sophisticated tactic in Coq's tactic language to discharge these obligations directly. We plan to investigate both of these approaches, with the ultimate goal of implementing a verified PEC checker in Coq.

Finally, we would also like to investigate extending XCert to support the "Permute" module from PEC [7]. This would allow additional loop optimizations to be easily added to CompCert, such as loop reversal and loop distribution. Adding such support to XCert would require formally developing the general theory of loop reordering transformations found in [19], upon which the PEC checker's "Permute" module is based. Doing this will be challenging because it's not clear how to express the above theory of loop transformations in a way that meshes well with CompCert's existing support for correctness proofs using simulation relations. Nonetheless, formalizing such a theory in Coq is worthwhile, as it would not only enable support for "Permute" optimizations in XCert, but could also be broadly useful within CompCert.

## 8. Related work

Our work is closely related to three lines of research: verified compilers, extensible compilers, and translation validation.

***Verified Compilers*** Verified compilers are accompanied by a fully checked correctness proof which ensures that the compiler preserves the behavior of programs it compiles. Examples of such compilers include Leroy's CompCert compiler [9], Chlipala's compilers within the Lambda Tamer project [2, 3], and Nick Benton's work [1]. At a lower level, Sewell et. al.'s work [14] on formalizing the semantics of real-world hardware like the x86 instruction set provides a formal foundation for other verified tools to build on.

However, none of these compilers are easily extensible – extending these compilers with additional optimizations requires either modifying the proofs or trusting the new optimizations without proofs. The main goal of our work is to devise a mechanism to cross this extensibility barrier for verified compilers. Although our work was done in the context of the CompCert compiler, the general approach that we took for integrating PEC into a verified compiler could be applied to other verified compilers.

***Extensible Compilers*** There has been a long line of work on making optimizers extensible. The Gospel language [18] allows compiler writers to express their optimizations in a domain-specific language, which can then be analyzed to determine interactions between optimizations. The Broadway compiler [6] allows the programmer to give detailed domain-specific annotations about library function calls, which can then be optimized more effectively. None of these systems, however, are geared at proving guarantees about correctness. The Rhodium [8] and PEC [7] work took the extensible compilers work in the direction of correctness checking. In these systems, correctness is checked fully automatically, but the execution engine is still trusted. Our current work shows how to bring a trusted execution engine to such systems.

***Translation Validation*** Translation validation [10–13] is a technique for checking the correctness of a program transformation after it has been performed. Indeed, it is often easier to check that a particular instance of a transformation is correct than to show that transformation will always be correct. Although these techniques may increase our confidence that a compiler is producing correct code, only a *verified* translation validator can guarantee the correctness of the *a posteriori* check performed by the validator. Tristan et. al. examine such techniques for using verified translation validation to add more aggressive optimizations to CompCert while keeping the verification burden manageable [15–17].

## Acknowledgments

We thank Xavier Leroy, Jean-Baptiste Tristan, and the rest of the CompCert team for developing and releasing a well-documented and well-engineered tool. We also thank the anonymous reviewers for their careful reading and helpful comments. Finally, we thank the UCSD Programming Systems group for many useful conversations and insightful feedback during the development of XCert.

## References

[1] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI*, 2009.

[2] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI*, 2007.

[3] A. Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010.

[4] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[5] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[6] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of IEEE*, 93(2), 2005.

[7] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.

[8] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[9] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.

[10] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[11] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.

[12] M. Rinard and D. Marinov. Credible compilation with pointers. In *Workshop on Run-Time Result Verification*, 1999.

[13] H. Samet. Proving the correctness of heuristically optimized code. *Commun. ACM*, 21(7):570–582, July 1978.

[14] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, , and J. Alglave. The semantics of x86-cc multiprocessor machine code. In *POPL*, 2009.

[15] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *POPL*, 2008.

[16] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. In *PLDI*, 2009.

[17] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL*, 2010.

[18] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, Nov. 1997.

[19] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, 2005.