Formal Verification of Compilers
Independent Study Report

Jedd Haberstro joh9446@rit.edu

Rochester Institute of Technology

Abstract. This study, completed as part of a Bachelor level independent study course, aims to survey the current landscape of research into the formal verification of compilers. This report includes a technical perspective of several current and past compiler verification projects conducted by researchers at leading universities and a final ending summary giving impressions on the future of formal compiler verification research.

1 Introduction

1.1 Problem Statement

A world free of computer program bugs – a dream of virtually all software developers. Most often developers are concerned by bugs that occur in their own software and seldom must they think about bugs in their compiler. For all intensive purposes, a compiler should be a black box that transforms source code to executable machine code, keeping the developer's expressly written intent intact. And yet, just like the developer's program, the compiler is a piece of software with a developer and bugs of its own.

A bug in a compiler can be fatal. Any reasoning done about a source program – the program being compiled – can be thrown out the window if there exists a compiler bug that alters the source program's behaviour. It is thus crucial to provide formal guarantees about the compilers that we use if we wish do the same for the programs that we write.

1.2 Formal Verification and Semantics

Before trying to understand current research endeavours into formal verfication of compilers, it's important to have an understanding of what it means to verify a compiler. At a broader level, formally verifying software is the principle of using mathematics, logic and (possibly) a set of accompanying program annotations to check and prove the preservation of a stated property about a piece of software. In modern day formal verification, the act of proving such properties is not done by hand but rather in an interactive theorem prover (also known as a proof assistant), and the act of checking a proof is consequentially performed by the interactive theorem prover itself.

Thus for a compiler to be formally verified it must have been proven to always preserve semantic meaning of the original source program it compiles. This of

course implies that we have a notion of semantics in the first place, of both the source program's language and the compiled program's language and a way to relate the two. To compare different semantics, we also assume to posses a notion of comparable output behaviour. Thus, given some program P such as a source program S or a compiled program C, we say $P \Downarrow B$ to mean "source program P when executed results in observable behaviour B." What *behaviours* actually encompass can vary depending on how strong of guarantees one would like to provide, but in the CompCert project for example, a nearly fully verified C compiler of which we will spend much time discussing throughout this paper, defines behaviour to encompass termination, divergence (aka non-termination), and "going wrong" (the execution of something for which there aren't semantics and thus could, for example, crash the program). Given these definitions, we can now state a precise definition of semantic preservation, as given in [1]:

Definition 1. $\forall B \notin \mathit{Wrong}, \quad S \Downarrow B \Rightarrow C \Downarrow B$

In other words, for all "non-wrong" behaviours, if source program S executes with behaviour B, than compiled program C also executes with behaviour B. It is however important to realize that this definition also assumes deterministic semantics, otherwise behaviours would not meaningfully be comparable [1].

2 CompCert

2.1 Overview

CompCert is a verified compiler for a large subset of the C language that compiles to PowerPC, IA32, and ARM machine code. It is probably most practical to start by reviewing the CompCert project, not only because it is perhaps the greatest practical achievement in formally verified compilers to this point in time, but also because many of the following research projects that will be talked about later in this paper were either inspired by CompCert or even direct improvements upon it.

2.2 Coq Development

What is unique about CompCert, besides the obvious fact that it is formally verified, is that it is written (almost) entirely in the Coq proof assistant – both the compiler itself and its accompanying proof of semantic preservation. This is interesting from an engineering standpoint because it shows that Coq is realistically capable of such engineering endeavours, and what more with Coq being a purely functional language, that the entirety of the compiler's implementation is purely functional as well [1].

At its core, the Coq language is an implementation of the Calculus of Inductive and Coinductive Constructions, a type of constructive logic. However it is more than that, providing a style of pure functional programming descended

from the ML branch of programming languages with recursive functions, inductive datatypes, and ML-style modules, as well as dependent types and higher-order logic [2].

The main advantages of writing a compiler directly in Coq is that every part of the compiler becomes a first class citizen of its accompanying proof. Typically, an additional bridge would be needed between the compiler's implementation and the language of the proof assistant; that is not so when the compiler is written in the proof assistant's language. Now every piece of the compiler can be directly reasoned over by the proof. This absence of the "bridge support" represents the removal of a huge engineering burden and provides motivation for making proof assistant language closer to "normal" programming languages. However, it's important to keep in mind that there is left still the large engineering effort of writing the actual proof, as evidenced by the fact that CompCert's verification code is approximately 6 times larger than the compiler's implementation code [2].

2.3 Architecture

CompCert is composed of a surprisingly large number of compiler passes. Each transformation is in fact almost a mini-compiler in its own right: for every pass there exists a distinct source and target intermediate language and for each intermediate language there is an accompanying well-defined semantics. The compiler is thus the culmination of each pass, where one pass' target intermediate language is the next pass' source intermediate language, the first pass' source intermediate language is the C programming language, and the final pass' target intermediate language is one of the three target machine code languages.

There are 16 passes in total, 11 of which have different source and target intermediate languages (as of the writing of [1]); they are (in order) *parsing*, *simplifications/type elimination*, *stack preallocation*, *instruction selection*, *CFG construction*, *CSE*, *LCM*, *register allocation*, *branch tunneling*, *code linearization*, *spilling and reloading calling conventions*, *layout of stack frames*, *instruction scheduling*, *PowerPC code generation*, and *assembling/linking*. Of all 16 phases, only the last phase, *assembling/linking*, is unverified, thus providing a near end-to-end verification [1].

The advantage of introducing so many intermediate languages and independent passes is that proving correctness of each pass becomes significantly easier since each pass attempts to do only a single transformation [2]. Thus, since each pass is more focused, so are the corresponding proofs. However, there are disadvantages as well. Despite each pass being independent in functionality from each other, there still exists implicit dependencies between passes. These dependencies arise from the fact that certain passes requires either certain information about its input to be transmitted to it or certain guarantees about its input to be met, thus exchanging this information or fulfilling these guarantees requires careful design of the intermediate languages [2]. By consequence, requirements imposed by one pass becomes an implementation constraint of another pass (specifically, the preceding pass).

2.4 Proving Semantic Preservation, An Example: Register Allocation

The intermediate language over which the register allocation pass operates is called RTL, a language in which functions are represented by a control-flow graph (CFG) of high-level machine instructions and there exists an infinite number of registers available to use. The goal of the register allocation pass is to replace the use of the infinitely numerous registers with either one of the "hardware registers", of which there is a finite set, or a stack slot, optimizing for the maximal utilisation of hardware registers. Although not important to the understanding of how semantic preservation is proved, the essential steps of the register allocation algorithm are as follows:

1. Perform lively analysis of registers via backward dataflow analysis of the equation:
$$LV(l) = \{T(s, LV(s)) \mid \text{s successor of } l\}$$
where l is the current program point, and $T(s, LV(s))$ finds the live registers before program point s. The result of lively analysis is a set from program points to live registers.
2. Build an interference graph where nodes are RTL registers and edges between two registers represent the fact they are live at the same program point.
3. Color the interference graph such there is one color that represents hardware registers, another color that represents stack slots, and that no two connected nodes are colored the same.
4. Rewrite the RTL code using the colored interference graph.

The standard form of proving semantic preservation is by showing that there exists a simulation diagram between two programs. A simulation diagram shows the correspondence between two sequences of transitions; that is to say, every transition in the original program must correspond to one or more transitions in transformed program, where correspondence means "to have the same observable effects" and to preserve some binary relation, $\sim$, that relates transition states. For register allocation, since no instructions are added or deleted (only modified) to the original program, there should be a one-to-one correspondence between transitions [1].

That leaves the question: what is a "transition"? For the RTL language, a transition is one step between execution states as defined by the language's dynamic small-step operational semantics, of the form: $G \vdash S \rightarrow S'$. Here G is an environment that maps function pointers names to function definitions, and S and S' are pairs, called transition states, that are composed of the current program point, the activation record, maps of registers to the values they hold, a heap map, and another structure representing the call stack. Every step/transition also produces an output trace [1].

The only thing left to define is the binary relation, $\sim$. In the context of proving register allocation, most components of the program state are quite

easy to relate. Since there must be a one-to-one correspondence, the current program point, the activation record, and the heap map must exactly match their transformed counterparts, and additionally the transformed control flow graph will be simply equitable based on the graph coloring scheme used to determine the transformation between the two. The only component then left to relate is the register map. This is quite simple as well; since dead registers semantically have no effect on any program point after it, it suffices to define the relation as follows

Definition 2. $\forall r \in LV(l), \quad R(r) = R'(\varphi(r))$

That is to say, register maps R and R' relate if for every live register r at program point l, the value of r in R is equal to value of the coloring of r in R' [1].

3 A Simple, Verified Validator for Software Pipelining

This work, *A Simple, Verified Validator for Software Pipelining* by Jean-Baptiste Tristan, is a natural next research project to approach because it was created as an extension to CompCert and the developed software pipelining optimization operates over the RTL intermediate language that we have already looked at briefly in the previous section.

3.1 Software Pipelining

The aim of this research was to add a verified software pipelining optimisation pass to the CompCert compiler. Software pipelining is a loop transformation that executes several iterations of the loop in tandem, resulting in better instruction-level parallelism and potentially fewer pipeline stalls [3]. It takes a loop such as:

```
i := 0;
while (i < N) { β }
```

and transforms it to:

```
i := 0;
if (N ≥ μ) {
   M := ((N − μ) / δ) × δ + μ;
   P
   while (i < M) { S }
   ε
}
while (i < N) { β }
```

where N does not change over the course of the loop and β is a basic block that does not contain any sort of conditional moves [3]. Essentially what has happened is that the original loop is split into three parts: a prologue P, a new loop body S, and an epilogue ε. The prologue performs μ iterations of the original loop, a single iteration of S performs δ iterations of the original loop, and the epilogue corresponds to 0 iterations of the original loop (it is instead responsible for finishing any operations left incomplete by the new loop body).

3.2 Key Insight

Classically, to prove that our original loop is semantically equal to the software pipelined loop, the following property must be shown to be true:

Proposition 1. *Forall N, symbolic evaluation of the original loop must be equal to symbolic evaluation of software pipelined loop.*

However, this is only decidable for known N's, not all N's, which is necessary for N to be variable at execution time of the program. However, [3] has made the key observation that the above undecidable proposition is implied by the following two decidable equalities:

$$\alpha(\varepsilon; \beta \text{ iterated } \delta \text{ times}) = \alpha(S; \varepsilon)$$

$$\alpha(\beta \text{ iterated } \mu \text{ times}) = \alpha(P; \varepsilon)$$

where α is symbolic evaluation [3]. We will see later both how to use this implication to verify semantic perservation and why this implication is sound.

3.3 Symbolic Evaluation

Before proceeding, it is best to provide a definition of symbolic evaluation. Symbolic evaluation is performed over a small and simple intermediate language. This language consists the following statements: variable to variable moves (assignments), arithmetic operations that are saved into variables, memory loads that are saved into variables, and memory stores (saving a variable value into memory). Each load and store are specialized by a memory quantity (such as 8-bit signed integer or 64-bit float) and a sequence of these language statement is called a basic block.

The symbolic state of a basic block (that is, a symbolic representation yielded by performing symbolic evaluation on a basic block), consists of resources, terms, resource maps, and computation lists. A resource is either a normal variable, a ghost variable Mem (which is just some abstract memory location), or an initial memory ghost variable Mem_0. A term represents either an initial value, the result of an operation, the result of a load, or the result of a store. It's important to note that stores and loads are parameterized by a previous memory term which can either be Mem_0 or a store term. Resource maps are finite maps from resources to terms that at the end of symbolic evaluation represent the final

state of the block after being run. Computations lists are finite sets of terms that represent computations, where computations is taken to mean any term that can have an observable, possibly failing side-effect (i.e. a run-time error) that is not deducible from the final state of the block (i.e. the resource map). Variable moves are the only instruction that when symbolically evaluated dont produce a computation. In order to prove two blocks to be semantically equivalent, they must have equivalent resource maps and equivalent computation lists. Equivalence of resource maps and computation lists is defined as the following:

Resource Maps Given the set θ of all variables in the original code before pipelining, then two resource maps are equivalent if for every resource in $\theta \cup \{Mem\}$, the resource maps map to identical terms.

Computations lists Strict identical equivalence.

Finally, actual symbolic evaluation of basic blocks is then just the composition of symbolic evaluation on individual instructions, and symbolic evaluation of individual instructions is a straightforward translation function by case analysis of the instruction type. The entire translation function won't be given, but a demonstrative example case is given below:

$$\alpha(r := \mathrm{op}(op, \overrightarrow{r})) = (r \mapsto t, \{t\})$$

where $t = \mathrm{Op}(op, \overrightarrow{r})$ (and Op(..) is a term, not the instruction)

3.4 Proving Semantic Preservation: Translation Validation

The final validation algorithm comes down to the follow checks:

$$\begin{aligned}
&\mathit{validate}(i,\ N,\ \beta)\ (P,\ S,\ \varepsilon,\ \mu,\ \delta)\ \theta\ = \\
&\quad \alpha(\beta\ \mathit{iterated}\ \mu\ \mathit{times})\ \approx\ \alpha(P;\ \varepsilon) \\
&\wedge\ \alpha(\varepsilon;\ \beta\ \mathit{iterated}\ \delta\ \mathit{times})\ \approx\ \alpha(S;\ \varepsilon) \\
&\wedge\ \alpha(\beta)\ \sqsubseteq\ \theta \\
&\wedge\ \alpha(\beta)(i)\ =\ \mathit{Op}(\mathit{addi}(1),\ \mathit{initial\ of}\ i) \\
&\wedge\ \alpha(P)(i)\ =\ \alpha(\beta\ \mathit{iterated}\ \mu\ \mathit{times})(i) \\
&\wedge\ \alpha(S)(i)\ =\ \alpha(\beta\ \mathit{iterated}\ \delta\ \mathit{times})(i) \\
&\wedge\ \alpha(\varepsilon)(i)\ =\ \mathit{initial}\ i \\
&\wedge\ \alpha(\beta)(N)\ =\ \alpha(P)(N)\ =\ \alpha(S)(N)\ =\ \alpha(\varepsilon)(N)\ =\ \mathit{initial\ of}\ N
\end{aligned}$$

of which, upon closer inspection, each constituent check can be naturally realized either from the key insight remarked upon above or by closer inspection of software pipeline transformation of the loop. What's interesting to note is that what is presented here is a *validator*. Instead of proving directly the soundness of the mechanics of the software pipelining algorithm, only properties about the algorithm's end result are tested for soundness. This is a useful verification technique when trying to verify algorithms that are potentially quite complex and thus difficult to reason about, but whose inputs and outputs aren't. However, to be truly verified, the validator must instead be verified, so it's essential that the verification of validator won't be more complex than the verification of the algorithm.

4 Automatically Proving the Correctness of Compiler Optimizations

4.1 Overview

CompCert's goal is to create a fully verified, end-to-end compiler. This is a respectable goal that has broad benefits, but it perhaps holds the most value for industries such as the medical and aviation fields where program error can literally have life or death consequences. However, providing such an end-to-end verification is a challenging engineering undertaking that unfortunately, at the moment, is not easily amenable to re-purposing such that the efforts that CompCert researchers have contributed can be reused in the context of other formally verified compilers.

This next project, Cobalt, aims to solve a smaller (yet not small!), but more generalized problem. That is, providing an extensible general-purpose method of automatically proving correct certain classes of compiler optimizations that can (more) easily be incorporated into existing compilers. Though not providing an end-to-end verification guarantee, such a method could potentially eliminate a significant source of compiler bugs or be part of a greater end-to-end compiler verification project.

The main contributions of this paper [5] are:

- Cobalt, a domains specific language in which programmers can express compiler optimizations over a control flow graph of a C-like intermediate representation.
- Examples of such optimizations (such as constant propagation, dead code elimination, and common subexpression elimination).
- A method to automatically prove sound Cobalt optimizations.
- An "execution engine" that can run optimizations written in Cobalt within the Whirlwind compiler.

4.2 The Cobalt Language

Cobalt is fundamentally a very simple pattern-matching based language over propositional logic. A Cobalt optimization can fall under one of three categories:

- Forward transformation, of the form:

$$\psi_1 \textbf{ followed by } \psi_2 \textbf{ until } s \Rightarrow s' \textbf{ with witness } P$$

- Backward transformation, of the form:

$$\psi_1 \textbf{ preceded by } \psi_2 \textbf{ since } s \Rightarrow s' \textbf{ with witness } P$$

- Pure analysis, of the form:

$$\psi_1 \textbf{ followed by } \psi_2 \textbf{ defines } \mathit{label} \; s \Rightarrow s' \textbf{ with witness } P$$

A forward transformation can be read as "transform s to s' if on all code paths that lead to statement s, there is a statement that satisfies ψ_1 and no statements between that statement and s don't satisfy ψ_2. The text refers to ψ_1 as the "enabling condition" and ψ_2 as the "innocuous condition". Additionally, P is a witness that captures the conditions of the enabling statement for later use; it has no impact on the semantics of the optimization.

A backward transformation works in a similar fashion, but by working on code paths from the end of the procedure to statement s, where the innocuous condition must precede the enabling condition.

Finally the last form, pure analysis, is similar to forward transformations except that no transformations are performed; rather, information about the control flow graph nodes is computed and attached to the nodes (marked using the *label*) for use by later optimization transformation.

To actually make sense of these transformations, it is of course important to understand what ψ_1/ψ_2, s/s', and the witness are concretely.

ψ_1/ψ_2 are propositional logic formulas over "labels" (which in of themselves are user defined properties, such as $stmt(x := 5)$ or $mayDef(y)$ [5], that are attached to nodes in the control flow graph) with syntactic sugar for case-expressions/pattern-matching.

s/s' are abstract statements written in the intermediate representation language that contain pattern variables for expressions, variables, and constants. These pattern variables are used to find matching concrete instantiations of the statements in the program that are candidate statements to be optimized/transformed.

The witness, though not having an impact on the semantics of the optimization, is a crucial component needed for proving correct the optimization. The witness is a predicate over execution states. We will see how the witness is used to prove the optimization transformation correct.

It is perhaps useful to provide a simple example of a cobalt (forward) transformation. The following is an implementation of basic constant propagation [5], where X and Y are variable patterns, and C is a constant pattern:

$stmt(Y := C)$
`followed by`
$\neg\ mayDef(Y)$
`until`
$X := Y \Rightarrow X := C$
`with witness`
$\eta(Y) = C$

Finally, there is one last key piece to a Cobalt program – the profitability heuristic. This is a heuristic that actually decides whether for a given candidate statement, should the optimization/transformation actually be performed (i.e. is it "profitable"?). Thus an optimization is written as one part transformation pattern and one part profitability heuristic. Because the two are completely modular, only the transformation needs to be proven sound (which means, where ever a valid matching statement is found, the transformation can always be

performed without altering semantic meaning), the profitability heuristic is free to be written in any programming language without compromising soundness. This is also advantageous because often the transformation itself is the simplest component and the bulk of an optimization's complexity lies in the profitability heuristic.

4.3 Proving Soundness

To prove soundness of a transformation, the paper shows that all that is necessary is (1) to prove that the witness is established by ψ_1 and preserved by ψ_2 and that (2) the witness implies that s and s' have the same semantic effect [5]. This can all be done automatically by the Cobalt implementation.

As noted in the discussion of profitability heuristics, only the transformation pattern need be proved sound because the paper also shows that sound transformations can always be applied without changing semantic meaning, thus the frequency (profitability heuristic) a transformation is applied is not a factor.

There is one final complication when considering multiple transformations: it is not guaranteed that a transformation that has been shown to be sound in isolation will be sound when applied alongside other sound transformations. This can happen if backward transformations were allowed to use forward pure analyses (consider an implementation of concurrent execution of dead assignment elimination and redundant assignment elimination on the statements $x := 5; x := 5$). However, Cobalt circumvents this issue by explicitly disallowing such a construct from being valid Cobalt code.

4.4 Pros and Cons

The major win of such a system like Cobalt is that new optimizations can be added to a compiler without adding to the "trusted computing base" since Cobalt has the ability to automatically prove sound such optimizations. Additionally, these optimizations, because being written in a DSL, require less compiler domain knowledge to use (one doesn't need to be a compiler "expert").

However, there still remain several disadvantages to such a problem. The first being that the Cobalt "framework" is not an end-to-end solution and there exists a sizable trusted computing base (unverified code in the implementation) that includes the correctness checker/automatic theorem prover, the manual proofs that accompany what is discharged by the correctness checker, and the the execution engine that executes Cobalt optimizations in the Whirlpool compiler framework. Additionally, Cobalt can't express inter-procedural optimizations or one to many transformations, it is hard to express optimizations that normally require the construction of complex data structures, and not all sound optimizations that are expressible in Cobalt will be accepted as sound (the Cobalt prover is not complete).

5 Bringing Extensibility to Verified Compilers

In large part, this paper represents the beginnings of the merger between projects such as CompCert and works such as PEC, Cobalt, Rhodium, etc of the DSL style verification approach. That is to say, CompCert provides a (near) complete guarantee of semantic preservation through the entire compiler pipeline. However, every part of CompCert has a corresponding correctness proof – a tedious process that needs to be altered and added upon each time the compiler is modified or extended. PEC and the likes instead are frameworks in which certain constraints are imposed and certain facts proven beforehand (and only once) allow the programmer to extend the compiler (the optimization phases of the compiler in this work) without the need for any such additional correctness proofs. This work, called XCert, is the first effort that tries to extend CompCert with such a DSL optimization language.

A discussion of PEC, the DSL language presented in this paper, and how individual optimizations are proved correct will be skipped here, as many of the concepts are similar to that of Cobalt, which has been discussed previously in this paper. Suffice it to say, the rewrites performed by PEC are shown to be correct by way of showing that simulation relations are preserved

Again, we see the choice of performing optimizations over CompCert's RTL intermediate language, much like we saw in Tristan [3]. Here RTL was chosen because it is CompCert's highest level control flow graph-based IR in which all source language constructs have been compiled away but no target language constructs have been added. However, despite this nice property, there still exists several challenges that arise from the choice of RTL. Previous DSL based efforts, such as Cobalt, operated over abstract syntax tree representations. Since RTL is a CFG-based representation, a new CFG pattern matcher had to be implemented (and verified) which was a much more complex undertaking compared to the corresponding AST-based pattern matcher due to the fact that CFGs will inherently contain cycles [6]. Due to the difficulty of reasoning about such a pattern matcher in Coq, verified validation was used instead to verify the pattern matcher's results. An additional complexity that arises from a CFG-based representation is the necessity of relinking ingoing and outgoing edges of the optimized region of code. Luckily, doing so is quite easy: outgoing edges naturally remain linked as a side-effect of the optimization rewrite algorithm, and relinking incoming edges is handled by updating the entry point instructions of the to-be optimized code region with the first instructions of the already optimized code region [6].

5.1 Proving Correctness

The main challenges of proving optimizations correct in the XCert arose from the discrepancies in representation between CompCert's semantics and PEC's semantics. That is, the largest challenge in proving correctness was that small-steps in the original program and the transformed/optimized program didn't

execute in lock stop, which meant the main property of correctness would no longer hold:

$$\eta_l \sim \eta_r \wedge \eta_l \rightarrow \eta_l' \Rightarrow \exists \eta_r', \eta_l' \sim \eta_r' \wedge \eta_r \rightarrow \eta_r'$$

where the ηs are program states, program states subscripted l belong to the original program, program states subscripted r belong to the optimized program, $\sim$ is a relation between program states, and $\rightarrow$ denotes a small-step of a small-steps operational semantics [6]. To address the issue of misalignment, an *L-step* and an *R-step* semantics are introduced. An *L-step* is meant to exactly correspond/align to an *R-step*. Then the normal stepping semantics of the original program (using CompCert's small-steps) can be shown equivalent to *L-step* semantics, and likewise the *R-step* semantics will be shown equivalent to semantics of the transformed program (which use the small-steps semantics of PEC). If non-terminating computations didn't exist, then *L-step* and *R-step* semantics could be defined to be exactly like the original normal stepping semantics except that they "big-step" over transformed regions of code. Unfortunately, this "big-step" behaviours provides no guarantees about non-termination in these stepped over transformed regions of code, and thus renders this approach insufficient. However, PEC does provide guarantees of its own about non-termination preservation. This leads to the further observation that PEC reasons over code using small-step relations; they just haven't (yet) been shown to be equivalent to CompCert's small-step representation. Thus, PEC was modified directly to return the simulation relation it generates for its small-steps simulations. This finally leads to the following definition for *L-step* and *R-step* semantics:

1. If outside of a transformed region, their semantics are the same to the original small-steps (of CompCert).
2. If inside the transformed region, then *L-step* and *R-step* step from entry to entry in the simulation relation.

This finally allows XCert to show whether or not the above property of correctness holds for optimized code regions.

6 Formalizing the LLVM Intermediate Representation for Verified Program Transformations

6.1 Overview

As LLVM gains more and more industry traction, it would be useful to have a agreed upon semantics with which properties about LLVM programs could be proven. This paper, [7], is the first attempt to give such semantics. The main contribution of this paper, called Vellvm, gives formalizations of a static semantics, a dynamic semantics, and a memory model to the LLVM project's intermediate representation. The formalization were mechanized with Coq and also included additional implementations in Coq to interact with LLVM. Finally, to demonstrate the practicality of this new formalization, a verified version of

the SoftBound LLVM IR compiler pass (an algorithm that hardens C programs against buffer overflows and other spatial memory safety violations) was developed in Coq.

6.2 Static Semantics

Vellvm requires that programs are in SSA form. To ensure this, there are several properties that programs must exhibit, and in whole, if all these properties are met, then the program can be said to be in SSA form.

The first property that Vellvm requires of its programs is that every variable is well-typed. By default, all LLVM IR components are accompanied with a type, so the complexity in ensuring this property is determining that the types themselves are well-formed. For a type to be well-formed, its definition must not contain any degenerate cycles; this means that any sum types whose components are of that same sum type must be a pointer. This ensures that all types' sizes are finite and known at compile-time.

The second property is that all variables must be well-scoped, and the third property is that all variables must be assigned exactly once. Both these two properties can be satisfied if a program satisfies what is called dominator analysis correctness. This states that (1) every function entry block (where a block is from a control flow graph, and which composes together to form functions in said control flow graph) dominates itself, and (2) given an immediate successor block b_2 of b_1, that all strict dominators of b_2 also dominate b_1. Domination is defined for blocks to mean that l_1 dominates l_2 if all execution paths that reach l_2 first go through l_1; for instructions, i_1 dominates i_2 if (within the same block) i_1 appears before i_2 (keeping in mind that all variables are restricted to being assigned exactly once).

6.3 Dynamic Semantics

Memory Model Vellvm adopts the memory model that is used by CompCert's. As a result, Vellvm inherits several properties, the important of which are: (1) only single threaded programs are supported, (2) pointers are 32-bit and 4-byte aligned, and (3) memory is infinite. The CompCert memory model for the purposes of Vellvm has however been extended to support arbitrary bit-width integers, paddings, and alignment.

There are several other notable properties about Vellvm's memory model. First, memory is dynamically typed. That means that memory carries with it information about its size, layout, and alignment, and that *load* and *store* commands much check these for consistency. Additionally, Vellvm's memory model supports physical subtyping – that is to say, a structured value can be read as a different structured value so long as the two structured types map to the same low-level byte-oriented representation. Finally, some behaviours are left undefined by LLVM, such as loading from unallocated addresses, loading with improper alignments, loading from uninitialized memory, etc., and thus this is reflected in Vellvm's memory model.

Operational Semantics Vellvm interestingly (and conveniently) provides 4 related operational semantics for the LLVM IR. The "base" or most general semantics is a small-step, non-deterministic operation semantics. From that, a deterministic, small-step semantics was created, and from that again there are two determinisitic, big-step operational semantics, one of which big-steps entire function evaluations, and the other of which big-steps "sub-blocks" (code regions between two function calls).

The non-deterministic, small-step operational semantics are non-determinisitic in areas where LLVM has specifically left ambiguity to give flexibility to optimization passes, or where the non-determinism would be deterministic under a concrete memory implementation. Specifically, non-determinism arises from the use of $undef$, and from certain memory errors.

$undef$ is a LLVM IR constant that represents a set of possible bit patterns of which LLVM compilers are allowed to freely pick from in whichever way is most beneficial. How exactly does $undef$ lead to non-determinism? Since $undef$ represents a set of possible values, this has the effect that either (a) a local variable could take on one of any of $undef$'s possible values, and (b) a small-step relation itself could relate one state to many states if the step itself depends on an $undef$ value (i.e. for example, a branch instruction whose condition is dependent on an $undef$ value).

To remove the nondeterminism, the determinisitic, small-step operational semantics differ only in that $undef$ is treated as a zero initializer, so now variables that map to $undef$ can only take on one possible value. As noted in the paper, this is not a realistic assumption to make of a C compiler, but it's perfectly reasonable in many other type-safe programming languages.

Finally, as stated above, there are two big-step operational semantics that big-step functions or big-step the blocks between functions. The advantage in having these equivalent big and small step, deterministic semantics is that some proofs are more amendable to one form or another. For example, many optimization passes work upon blocks, and thus it is difficult to establish correctness properties at the granularity of single instructions that small-step operational semantics provide.

6.4 SoftBound

Finally, to show the efficacy of Vellvm's various semantics, LLVM's optimization algorithm SoftBound was verified. The actual details of the SoftBound algorithm are not important to understand. Rather, how Vellvm facilitates verifying that SoftBound is "correct" (which in the case of this specific algorithm means that the program either doesn't have spatial memory violations [or acts in a predictable way if does] and respects the original semantics of the program) is what is important to grasp. The general outline of what was needed to done to prove such a thing is as follows [7]:

1. A stronger version of one of the original Vellvm IR semantics was created – a version that has been supplemented to provide information about the properties that SoftBound tries to enforce.

2. Prove that under those semantics, the additional properties in sum do indeed enforce a general property (in the case of SoftBound, no spatial memory violations).
3. Implement the actual translation pass from regular LLVM IR to the LLVM IR that carries the additional property information to support the new semantics.
4. Prove that if indeed the actual translation pass succeeds in yielding a program with the additional property information, then if that program is ran using the original (deterministic) semantics, it would be equivalent to running it with the "new semantics".

In general, this may not seem astonishing – this is just the typical proof technique. The important contribution is that proving an algorithm such as SoftBound, which was once impossible to do in LLVM due to the lack of formal semantics, is now possible and has been demonstrated successfully. A large step of the proof process, defining semantics, has been done beforehand.

7 Mechanized Verificaton of CPS Transformations

This paper [4] gives the first mechanically proven continuation-passing style (CPS) transformation for a realistic core lambda calculus that supports n-arity functions. Being the basis of the paper, it is important to understand what is continuation-passing style. Simply put, it is a style where in lieu of returning values, functions take an additional parameter (which in turn is expected to be a function) to which the result of the function is passed. For example, a simple function such as $x => x + x$ becomes $xf => f(x + x)$ after transformation to CPS.

The original CPS transformation algorithm from which the paper builds upon is the following:

$$[\![x]\!] = \lambda k.\text{k x}$$
$$[\![\lambda x.M]\!] = \lambda k.\text{k } (\lambda x.\, [\![M]\!])$$
$$[\![MN]\!] = \lambda k.\, [\![M]\!]\, (\lambda m.\, [\![N]\!]\, (\lambda n.\text{m n k}))$$

The downfall of this algorithm is that it produces a lot of intermediary beta-redexs (i.e. places where superfluous lambda functions have been introduced and could be simplified via function application),

They also discuss two other alternatives that improve this CPS transformation algorithm; the first of which unfortunately doesn't eliminate all redexs, and the second of which introduces a denotation that specializes on one of its arguments, thus making the number of cases for multiple arguments combinatorial and thus ill-suited for the goals of the paper. Finally, a third and final improvement is proposed:

$$(\lambda x.M)@_\beta A = M\{x \leftarrow A\}$$

$$M @_\beta N = M\ N$$

$$[\![x]\!] \triangleright k = k\ @_\beta\ x$$
$$[\![\lambda x.M]\!] \triangleright k = k\ @_\beta\ (\lambda x.\lambda k.\ [\![M]\!] \triangleright k)$$
$$[\![M\ N]\!] \triangleright k = [\![M]\!] \triangleright \lambda m.\ [\![N]\!] \triangleright \lambda n.\ \text{m n k}$$

where the $@_\beta$ is called the "smart application" constructor and it performs beta reductions "on the fly" when the first argument is a lambda-abstraction (aka a function) and the second argument is an atom. In other words, it expands the lambda-abstraction and performs the free variable substitution with the atom as the transformation "runs" instead of leaving it to "runtime" or some later optimization pass that would need to be proved as well).

To demonstrate the practically of the new CPS transformation, two small functional languages are defined that contain variables, recursive and non-recursive functions, data constructors/algebraic datatypes, and patterns matching. The two languages are nearly identical, with the distinguishing factors being that (1) the target language introduces a second kind of variable type (continuation variables) and that (2) each function takes an additional parameter (for the continuation). Big step operational semantics for each language are then given.

To then show that a CPS transformation preserves semantic meaning, it was show that if a source program P evaluates to a value v, then the transformed program, when applied to the "initial continuation/identity function", will evaluate to a CPS transformed version of v. To prove this for the proposed "smart application" CPS transformation algorithm, this property was first proved for the original CPS transformation algorithm (using an adapted version for their source and target language instead of the bare untyped lambda calculus), and then shown that the proposed CPS transformation algorithm was equivalent to the original.

8 Conclusion

It is evident that CompCert has had a significant impact on the landscape of formal compiler verification. It would be hard to refute the claim that CompCert will continue to have a large influence upon future compiler verification research, especially as CompCert picks up traction within Avionics and similar industries. However, the challenges that lie ahead for such projects are mostly those of the engineering sort. The CompCert project has taken years of engineering effort to create a compiler for C, a language which is relatively simple as far as programming languages are concerned. CompCert also posses very little to no automaticity or extensibility. Every part of CompCert must be verified by hand in Coq. Changing existing code is cumbersome, adding new passes is tedious (as it also requires creating new intermediate languages with formal semantics), and even the thought of changing the architecture of CompCert will surely incite nightmares. It is thus my opinion that the next great improvements

in formal compiler verification will be in the art of compiler verification. Programming language research is often attempting to make the life easier for users of the programming languages researchers create. It's now time to make life easier for the researchers themselves. Such endeavors have all ready been started by some, such as Adam Chlipala and his work in using Coq's dependent types to increase researcher productivity [8] and parametric higher-order abstract syntax to alleviate reasoning about variable binder manipulation [9].

At the same, I do believe we'll still continue to see researchers work on actual verification of compiler problems. In particular, I believe we will start seeing researchers and programmers alike try to encode more information about their programs in type systems of the languages that they're using. The aim of such projects would not be to provide definitive guarantees about entire programs, but rather provide stronger guarantees about more aspects of our programs. Already, some such efforts have been undertaken, as demonstrated by the SystemF compiler of Louis-Julien Guillemette and Stefan Monnier which uses Haskell's generalized algebraic data types to verify mechanically the preservation of types between each phase of the compiler [10].

In short, much progress has been made in the past 10 years and much more is surely to be made in the following 10. Formal compiler verification is just starting to sprout and it will be an exciting time to watch it fully blossom.

References

1. Xavier Leroy. Formal verification of a realistic compiler. Communications of the ACM, 52(7):107-115, 2009.
2. Xavier Leroy. A formally verified compiler back-end. Journal of Automated Reasoning, 43(4):363-446, 2009.
3. Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In 37th symposium Principles of Programming Languages, pages 83-92. ACM Press, 2010.
4. Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007, volume 4790 of Lecture Notes in Artificial Intelligence, pages 211-225. Springer, 2007.
5. Sorin Lerner and Todd Millstein and Craig Chambers. Automatically proving the correctness of compiler optimizations. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 220-231. ACM Press, 2003.
6. Zachary Tatlock and Sorin Lerner. Bringing Extensibility to Verified Compilers. 2003.
7. Jianzhou Zhao and Santosh Nagarakatte and Milo M. K. Martin and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL). 2012
8. Adam Chlipala. An Introduction to Programming and Proving with Dependent Types in Coq. Journal of Formalized Reasoning (JFR). 3(2). 1-93, 2010.

9. Adam Chlipala. A Verified Compiler for an Impure Functional Language. Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10). January 2010.
10. Louis-Julien Guillemette and Stefan Monnier. A Type-Preserving Compiler in Haskell. The International Conference on Functional Programming. 2008.