LLVM M.D.: A Denotational Translation Validator

Jean-Baptiste Tristan Paul Govereau Greg Morrisett
Harvard University
{tristan,govereau,greg}@seas.harvard.edu

Abstract

Translation validation is a static analysis that, given two programs, tries to verify that they have the same semantics. In this paper, we present a novel design for translation validators: denotational translation validation. Instead of verifying a simulation relation between two programs—as usual translation validators do—a denotational translation validator computes and compares denotations. We explain how, for the class of programs that can be found in safety-critical code, such a design can be made effective. Finally, we report on our preliminary experiments with LLVM M.D., a denotational translation validator that we test on the LLVM optimization framework.

1. Introduction

Translation validation is a static analysis that, given two programs, tries to verify that they have the same semantics [19]. This method is particularly useful for validating the transformations done by optimizing compilers. Successes range from generic translation validators for moderately-optimizing industrial-strength compilers [2, 16, 21], to special-purpose translation validators for advanced optimizations [8, 18, 27]. Translation validation is on the verge of becoming a critical tool, both for formal verification of certifying [17] and certified compilers [12]. Translation Validation is also useful for compiler engineering where it can greatly simplify debugging and improve testing [20].

While in general not decidable, translation validation can be done efficiently if one program is a compiled or optimized version of other. Previous works [11, 16, 21, 28] demonstrate translation validators can scale to real-world compilers. From a bird's-eye view, these validators must compute and enforce a simulation relation that, in turn, may require dataflow analyses. If the dataflow analyses used by the validator are carefully crafted, then the validator can be very efficient. Some validators [16, 21] also use a pinch of symbolic evaluation, which, as noted by Necula is equivalent to computing predicate transformers. Symbolic evaluation is very effective in this setting because it masks syntactic details such as the order of independent instructions, renaming of local registers, and reuse of independent but identical computations. Symbolic evaluation is also very simple to use in the design of a validator, yet it is limited to extended blocks. If we could extend symbolic evaluation to whole functions, including loops, that would allow us to avoid using simulation relations and dataflow analyses, and do all of the validation symbolically. This kind of validator would compute a symbolic *denotation* of the input programs, and then compare the denotations.

In theory, such a *denotational* translation validator could work as follows.

- First, we translate two programs to purely functional representations where all of the side effects are made explicit.
- Then, we compute a predicate transformer for each of the functional representations.
- Finally, we check that the two predicate transformers (denotations) are provably equivalent.

While there is no evidence that this validation strategy can be made efficient, we observe that if we consider a restricted class of programs—restrictions corresponding to safety-critical code guidelines[1]—we can design an efficient, and therefore practical, denotational translation validator.

In this paper, we describe the design of a denotational translation validator. To our knowledge, it is the first translation validator that does *not* build or verify a simulation relation, but rather computes a denotation that is sufficient to validate optimizations. The theoretical steps that we presented above are realized as follows.

- First, we translate two LLVM functions to our representation language with explicit effects (section 3).
- Then, we use a generalization of symbolic evaluation to compute a symbolic value for each of the representations (section 4).
- Finally, we check that the symbolic values are equivalent using normalization and syntactic equality (section 5).

The difficulty lies in the generalization of symbolic evaluation to while loops. To be useful for translation validation, the symbolic values produced must be resilient to syntactic changes in the source programs. Also, we must be able to compare symbolic values efficiently. In this paper, we propose a method for symbolically representing while loops. We note that for a fixed number of iterations, the symbolic value of a loop could simply be the symbolic value of its unfolding. However, if we were to pursue this idea naively, we would have an unbounded number of symbolic values. Our observation is that, for a given loop, all of these symbolic values can be built out of a finite set of smaller symbolic values. Furthermore, each of these smaller values are resilient to syntactic changes, and easy to compare. Our method is very similar to the classic computation of a predicate transformer for a loop[6]. For our method to work efficiently, we have designed hand-in-hand our intermediate language (GDSA), and the symbolic evaluation itself (section 2).

To validate our design, we have implemented a tool, LLVM M.D. (Low Level Virtual Machine Mis-optimizations Detector), that we use with the LLVM compilation framework [5] (section 6). As an experiment, we have optimized the sqlite3 database [24]

[Copyright notice will appear here once 'preprint' option is removed.]

and used LLVM M.D. to validate the transformations. We also use LLVM M.D. to validate interprocedural optimizations on code generated from Simulink developments.

We believe that the capabilities of this tool are significant: within a single execution, it can validate scheduling optimizations (such as trace scheduling) as well as redundancy elimination (such as lazy code motion and sub-expression elimination based on global value numbering). In addition, our algorithm can validate combinations of optimizations, such as sparse conditional constant propagation, and can take into account basic non-aliasing rules. Our tool can also validate loop optimizations such as loop invariant code motion, and loop deletion, fusion and fission. If a program does not use recursive functions, as should be the case for safety-critical code, our tool can also handle interprocedural optimizations[1]. As we explain our validator, we will present a more detailed list of optimizations that, we think, should be validatable.

2. Overview

The key intuition behind this work is that while many optimization algorithms are extremely complex, the result is almost always a simple syntactic transformation of the program. As pointed out by Necula[16], symbolic evaluation is a very effective way to deal with syntactic differences between programs, and this is the heart of what a translation validation system must do.

Following Dijkstra [6], we construct a predicate-transformer semantics using an intermediate language of guarded commands. Our intermediate language is designed to resemble SSA-form assembly language. The symbolic evaluation of this language computes a weakest precondition, which is a finite symbolic value that we can use to compare two programs. Like Dijkstra, we handle loops by constructing a set of indexed rewrite rules for the registers appearing in the loop body—the weakest precondition of a loop is then the limit of these formulas. We have extended this strategy to nested loops so that we may handle a large amount of real-world code. We note (as does Dijkstra), that unlike an axiomatic semantics, our symbolic values are computable.

2.1 Basic Blocks

We begin by describing how we validate basic blocks. As input, we will use a three-address assembly language with an infinite number of registers. We will also require the registers be in static single assignment (SSA) form (each register has only one definition). For our experiments we have used LLVM as our input language, which satisfies these criteria.

As part of our validation process, we will create a denotation for each register. The denotation for a register will be a symbolic value that represents the computation that produces the value held by that register. Producing the denotation amounts to a symbolic evaluation of the relevant assembly instructions.

For example, consider the following basic block, B1, where we assume registers a and b are previously defined.

$$\begin{aligned} \text{B1:}\quad x_1 &= a \times 3 \\ x_2 &= b \times 3 \\ x_3 &= x_1 + x_2 \end{aligned}$$

Because the basic block is in SSA form, we can construct a symbolic value by replacing each occurrence of a register by its unique definition. This will lead to a set of symbolic values for the registers defined in the block in terms of the previously defined registers. For instance, the symbolic value of the register x_3 can be obtained by replacing x_1 and x_2 by their definitions. The resulting value for x_3 is: $a \times 3 + b \times 3$. This symbolic value is a representation of the computation that will take place at runtime to produce the value stored in x_3.

To clarify the difference between the assembly code and the symbolic values, we annotate each register appearing in a symbolic value. For example x_3^o is the symbolic representation of the register x_3. The symbolic language also contains $\times$, $+$, $load$, $store$, etc.[2] which are symbolic representations of the corresponding assembly instructions.

For simple straight-line code, we compute symbolic values as follows: First, we construct a set of rewrite rules using the assembly instructions. The assembly instruction:

$$x_1 = a \times 3$$

becomes the rewrite rule:

$$x_1 \mapsto a \times 3 \qquad .$$

We have designed our intermediate language to resemble SSA-form so that the translation from assembly code is as straightforward as possible. The process of generating the rewrite rules is slightly more complex for conditionals and loops, as we will describe in the next sections.

Once we have these rewrite rules, we can compute the symbolic value of any register. To compute the symbolic value of register x_3, we start with the symbolic value x_3^o, and then rewrite step-by-step using the rewrite rules.

In this context, the assembly instructions are isomorphic to a set of rewrite rules, and a sequence of instructions is a set of such rules. As we have mentioned, the rewriting process is a form of symbolic evaluation. We represent this process with α, which takes two arguments: a symbolic value to rewrite, and a set of rewrite rules. The rewriting just described is written as:

$$\alpha(x_3^o, \{x_1 \mapsto a \times 3, x_2 \mapsto b \times 3, x_3 \mapsto x_1 + x_2\}) \qquad .$$

The result of the symbolic evaluation is a denotation of the register x_3. Functional programmers will surely notice a similarity between this last formula and the parallel let statement. Indeed, our intermediate language is a simple functional language. However, as we have noted, the syntax is closer to SSA-form assembly language than to traditional functional code.

Previous works [16, 21, 26, 27] show that this simple transformation is a very effective way to build translation validators. To see how this is done, consider the basic block below, which is an optimized version of block B1.

$$\begin{aligned} \text{B2:}\quad y_1 &= b + a \\ y_2 &= y_1 \times 3 \end{aligned}$$

If we want to check that the original register x_3 will hold the same value as y_2, we first compute the set of rewrite rules for both blocks. Then, we symbolically evaluate x_3^o with the rewrite rules from block B1, and y_2^o with the rules from block B2. We then check that the two symbolic values are equivalent. In this case, we must check:

$$a^o \times 3 + b^o \times 3 \equiv (b^o + a^o) \times 3$$

which is well within the capabilities of many automated theorem provers. Note, however, that in our experiments we use a very simple equivalence checking algorithm: syntactic equality with a few simple normalization rules, and we are able to achieve very good results. This is because we have chosen our few rewrite rules to correspond to the kinds of rewritings that LLVM will do. While

[1] See Muchnick's compiler textbook [15] for more information on these optimizations

[2] We do not use $+^o$, $\times^o$, etc. because it clutters the presentation, and we feel that it is clear when we are referring to symbolic operators verses assembly instructions.

this is strictly not necessary[3], we believe this sort of "configuration" is much easier than we have seen in other translation validation systems.

Side Effects. The symbolic evaluation we have described up to this point would not be correct in the presence of side effects. Consider the following basic block.

$$\begin{aligned} &p_1 = \texttt{alloc}\ 1 \\ &p_2 = \texttt{alloc}\ 1 \\ &\texttt{store}\ x, p_1 \\ &\texttt{store}\ y, p_2 \\ &z = \texttt{load}\ p_1 \end{aligned}$$

By applying our symbolic evaluation process for z, we arrive at the symbolic value: $z^o \mapsto \texttt{load}\ (\texttt{alloc}\ 1)$, which does not capture the complete computation for register z. In order to make sure that we do not lose track of side effects, we use abstract state variables to capture the dependencies between instructions. A simple translation gives the following rewrite rules for this block:

$$\left\{ \begin{aligned} p_1, m_1 &\mapsto \texttt{alloc}\ 1, m_0 \\ p_2, m_2 &\mapsto \texttt{alloc}\ 1, m_1 \\ m_3 &\mapsto \texttt{store}\ x, p_1, m_2 \\ m_4 &\mapsto \texttt{store}\ y, p_2, m_3 \\ z, m_5 &\mapsto \texttt{load}\ p_1, m_4 \end{aligned} \right\}$$

This translation is the same as we would get if we interpreted the assembly instructions as a sequence of monadic commands in a simple state monad[14]. Using these rewrite rules, the symbolic values now capture all of the relevant information for each register.

Our symbolic evaluation is parametrized by a set of rewrite laws that are applied to simplify the symbolic values. One of these laws takes into account pointer aliasing information. In our setting (LLVM), we know that pointers returned by `alloc` never alias with each other. Using this law and the above rewite rules, the symbolic value for register z is:

$$z^o \mapsto \texttt{load}\ p_1^o, (\texttt{store}\ x^o, p_1^o, (\texttt{alloc}\ 1, m_0)) \mapsto x^o \quad .$$

We must ensure that the abstract state variables we introduce are also in SSA form. Having done this, we may apply our simple symbolic evaluation to the modified rules, and capture all of the relevant parts of the computations. A similar technique can be applied to other kinds of side effects, such as arithmetic overflow, division by zero, and non-termination. Note, thus far we have only modeled memory side-effects in our implementation. Hence, we only prove semantics preservation for terminating programs that do not raise runtime errors. However, our structure allows us to easily extend our implementation to a more accurate model.

2.2 Extended Basic Blocks

We extend our technique to conditionals by introducing a *guarded* ϕ-node to our intermediate language. Consider the following program, which uses a normal ϕ-node as you would find in an SSA-form assembly program.

$$\begin{aligned} \textbf{entry}:&\quad c = a < b \\ &\quad \texttt{cbr}\ c, \textbf{true}, \textbf{false} \\[1ex] \textbf{true}:&\quad x_1 = x_0 + x_0 && \text{(True branch)} \\ &\quad \texttt{br}\ \textbf{join} \\[1ex] \textbf{false}:&\quad x_2 = x_0 \times x_0 && \text{(False branch)} \\ &\quad \texttt{br}\ \textbf{join} \\[1ex] \textbf{join}:&\quad x_3 = \phi(x_1, x_2) && \text{(Join point)} \end{aligned}$$

If we naively apply our technique, we have

$$\alpha(x_3^o) = \Phi(x_0^o + x_0^o, x_0^o \div x_0^o)$$

where we use Φ for the symbolic term corresponding to a ϕ-node. This term represents two possible values for x_3. However, this is not enough to compare programs because information about how we arrived at each branch is lost. If we change the condition to $b < a$, we will have the exact same symbolic value for x_3 even though the semantics of the program has changed. We therefore extend ϕ-nodes with a guard derived from the condition(s) that was used to split the control-flow.

For the extended block above, the set of rewrite rules are:

$$\left\{ \begin{aligned} c &\mapsto a < b \\ x_1 &\mapsto x_0 + x_0 \\ x_2 &\mapsto x_0 \times x_0 \\ x_3 &\mapsto \phi(c, x_1, x_2) \end{aligned} \right\}$$

With these rules, the symbolic value of register x_3 is:

$$\alpha(x_3^o) = \Phi(a^o < b^o, x_0^o + x_0^o, x_0^o \times x_0^o) \quad .$$

We also extend our rewrite laws to allow us to simplify Φ-nodes when the guard is known or the two branches are equivalent.

2.3 While Loops

We now explain how to generalize our algorithm to handle while loops. To do this, we must come up with a finite, computable representation for variables that are modified within a loop, which is *resilient to minor syntactic changes in the source program*. To take an example, consider the register x whose value depends on the execution of the while loop presented in figure 1a. We cannot compute x's symbolic value simply by rewriting because the computation would diverge due to x_p. Also, it is not clear what the guards of the ϕ-nodes in the loop header should be.

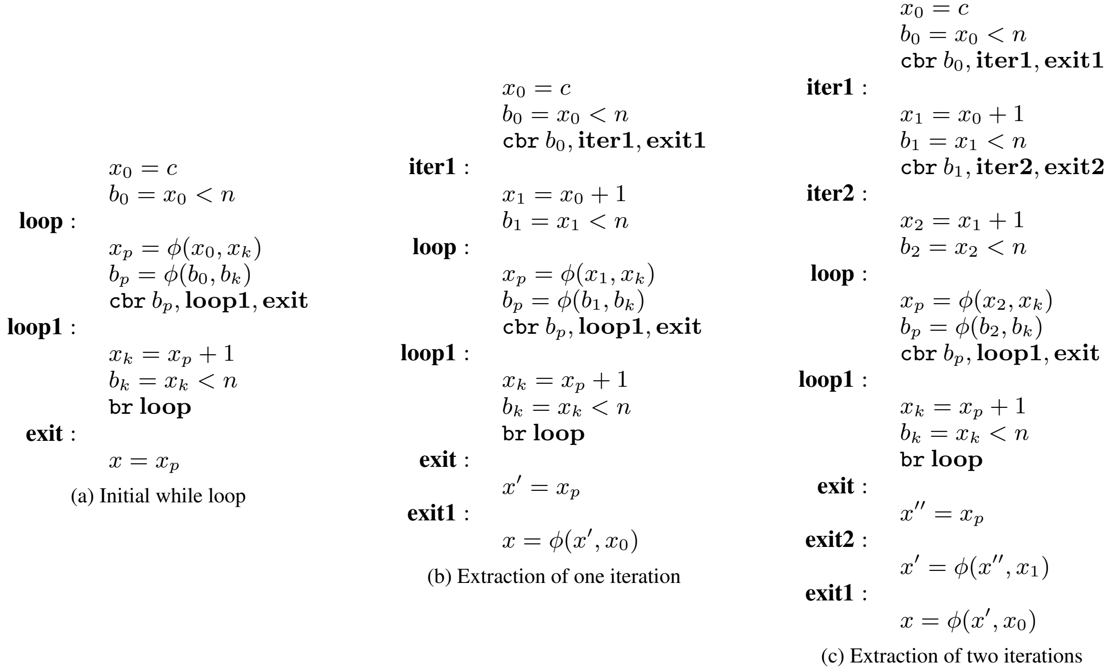
However, if we extract one iteration of the loop (unroll the loop 1 time), as presented in figure 1b, then the symbolic value of x becomes $\Phi(b_0^o, x_p^o, x_0^o)$. We still cannot compute a symbolic value for x_p, but we do have a symbolic value that faithfully represents the value of x if the loop executes zero times.

We can extract another iteration of the loop, as presented in figure 1c, to refine the approximation of x^o. With two iterations extracted, the symbolic value of x is $\Phi(b_0^o, \Phi(b_1^o, x_p, x_1^o), x_0^o)$. This symbolic value faithfully represents the value of x if the loop executes zero or one times. If we continue this unrolling process, the symbolic term for x will have the shape:

$$\Phi(b_0^o, \Phi(..., \Phi(b_{n-1}^o, \Phi(b_n^o, x_p, x_n^o), x_{n-1}^o), ...), x_0^o) \quad .$$

More generally, as the loop continues, the values defined at iteration i, must be defined in terms of values with index i or $i - 1$ (or no index at all if they do not vary within the loop). We can think of the values of register x as a mathematical sequence defined by

[3] Choosing specific rewrite rules makes the system more efficient.



(a) Initial while loop

(b) Extraction of one iteration

(c) Extraction of two iterations

Figure 1: Extraction of iterations

the recurrence relation,

$$x^o_n = \begin{cases} c & \text{if } n = 0, \\ x^o_{n-1} + 1 & \text{otherwise.} \end{cases}$$

and the register b as a sequence defined by,

$$b^o_n = \begin{cases} x^o_0 < n^o & \text{if } n = 0, \\ x^o_n < n^o & \text{otherwise.} \end{cases}$$

Using these formulas, we can "fill in" the symbolic value for x:

$$\Phi(c^o < n^o, \Phi(c^o + 1 < n, ..., c^o + 1), c^o) \qquad .$$

We could use this (possibly infinite) sequence of Φ-nodes as the symbolic value of x. However, in order to keep the denotation finite and computable, we introduce a new symbolic value Ω to represent values with this shape. In this example, we write:

$$x^o = \Omega[(x_o, x_n), (b_o, b_n)] \qquad .$$

This term indicates that x is a variable modified within a loop with condition b, and that the recurrence relations defined by x_0, x_n and b_0, b_n will be needed to compute the final value. In general, an Ω value will carry all of the recurrence relations that are needed to define the variable of interest, in this case, x.

To compare two Ω values, we must check that they represent the same sequence. We do this by checking that they use the same recurrence relations to generate the sequence, and that they both begin with the same initial values. The Ω values are finite representations of the sequences generated by a loop. It is not surprising that we can construct these finite representations: after all, the input program is one such representation, albeit one that is not resilient to syntactic changes in the source program. Using Ω in this way we can compare the symbolic values of loops, and this generalization of symbolic evaluation allows us to extend our translation validation algorithms to a larger class of programs.

To summarize, we have shown that, as we extract iterations from a loop, we can refine the symbolic value of variables defined within the loop, and argued that any refinement can be rebuilt from finite, computable, symbolic representation. However, in our input programs, the loops are not in this form, and we have not yet shown how we can represent the limit of this extraction process.

Every time we extract an iteration from a loop, we repeat the following two patterns. First, as we extract iterations from the loop body we must preserve the SSA structure. We do this by indexing variables with their iteration number, effectively putting the program in dynamic single-assignment form. We produce instructions such as $x_1 = x_0 + 1$, $x_2 = x_1 + 1$, etc. The limit of this process is a set of rewrite rules that can be represented as a comprehension $\{x_n \mapsto x_{n-1} + 1 \mid n \in \mathbb{N}^*\}$. Second, we modify the control flow so that for each extracted iteration, we can proceed though the extracted body or not based on the condition. In order to represent the limit of this control-flow structure, we introduce a new operator in our language of guarded commands: $\mu_n(b, x)$, where n is the iteration number, b is the loop condition, and x is the variable being defined. We can generate any refinement of the control-flow from this operator, by assigning it with the following identity:

$$\mu_n(b, x) = \phi(b_n, \mu_{n+1}(b, x), x_n)$$

The variable x that is modified within the loop is then defined by the rewrite rule:

$$x \mapsto \mu_0(b, x) \qquad .$$

Once again, there is a striking similarity to traditional functional languages if we think of μ as a fixpoint combinator applied to a function of n, b, and x.

In the end, we represent loops in our intermediate language as a complete unrolling, leading to a language in dynamic single assignment form with a μ recursor. As an example, for the loop

presented in figure one, we will produce the GDSA program:

$$\left\{\begin{array}{r}x_0 \mapsto c\\ b_0 \mapsto x_0 < n\\ x \mapsto \mu_0(b,x)\\ x_n \mapsto x_{n-1}+1\\ b_n \mapsto x_n < n\end{array}\right\}$$

Using these rules, we can, in theory, iteratively reconstruct the result of the extraction process:

$$\begin{aligned} x &\mapsto \mu_0(b,x)\\ &\mapsto \phi(b_0,\mu_1(b,x),x_0)\\ &\mapsto \phi(b_0,\phi(b_1,\mu_2(b,x),x_1),x_0)\end{aligned}$$

where each of the variables x_n and b_n are defined by a set comprehension.

2.4 The validation process: the big picture

To recap, our translation validator has three important steps. First, a compiler takes an LLVM function and its optimized version as inputs, and computes two intermediate representations. The intermediate representation that we use is in dynamic-single assignment form and uses guarded ϕ-nodes. Therefore, we refer to it as the guarded dynamic-single assignment language (GDSA). Each GDSA representation has three components: a value returned by the function (if any), a value representing the final state, and a set of rewrite rules.

The second step in the validation process uses symbolic evaluation to construct two symbolic values for each function: one starting from the return value, and one starting from the final state. In our examples, the final state represents the contents of memory after the function has completed, but it may also contain information about run-time errors or non-termination.

In last step, the symbolic values representing the result of the functions, and the symbolic values representing the final states are compared. If they match, the validator acknowledges the optimization, and more optimizations can take place. Otherwise, the validator raises an alarm. This alarm may indicate a semantic mismatch, or it could be a false alarm, and the validator has to be updated accordingly.

So far, we have described how our rewrite rules and symbolic language allow us to do translation validation. However, we have not yet described how we compute a GDSA representation from an LLVM source program. We will describe this compilation process in section 3. First, we will describe the GDSA language more formally.

3. GDSA

The guarded dynamic single assignment language (GDSA) is an intermediate language used to represent a program as a set of rewrite rules. If we symbolically evaluate a variable with respect to these rewrite rules, we will compute a symbolic value representing that variable. The syntax of GDSA is given below.

The GDSA language contains variables (**var**) and identifiers (**ident**). The variables are unique tokens usually originating from the LLVM source program. Identifiers are variables that may contain one or more indexes. The indexed variables refer to recurrence relations for variables that are modified within a loop. For example, if a variable, x, is modified within a loop we will likely see identifiers: x, x_0, x_n and x_{n-1} in the GDSA terms; these last three indexed variables coming from the recurrence relations for x. In the case of nested loops, an identifier may have several indexes. In our syntax, the indexes on variables are formal parameters. We will sometimes refer to specific values, such as x_7 or x_k, but x_n and x_{n-1} should be thought of as defining a comprehension for a set of rewrite rules.

var	::=	$x, y, z, \ldots$	Variables
ident	::=	**var** | $\mathbf{ident}_n$	Identifiers
ℓ	:=	`mem`, `overflow`, ...	States
term	::=	1, true, null, ...	Constants
	|	**ident**	Variable
	|	**inst** ($\overrightarrow{\mathbf{var}}$)	Instruction
	|	**call** ($\overrightarrow{\mathbf{var}}$)	Function Call
	|	**var** { ℓ = **var** }	State Modification
	|	**var**.ℓ	State Projection
	|	ϕ(**var**,**var**,**var**)	Phi node
	|	μ(**var**,**var**)	Mu node
RS	::=	$\{\ \overrightarrow{\mathbf{ident} \mapsto \mathbf{term}}\ \}$	Rewrite Rule Set
P	::=	(**var**,**RS**)	Program

Figure 2: Syntax of the guarded single assignment language

A complete program, **P**, is represented as a rewrite rule set (**RS**) together with a variable representing the result of the program. This variable gives us a starting point for symbolic evaluation. As the name implies, the rewrite rules for a program form a set, with at most one rule for each identifier. A single rewrite rule maps one identifier to a GDSA term.

The GDSA terms contain constants such as 1, $true$, and $null$. We also have a set of instructions (**inst**) that correspond to the instructions of the source assembly language. In our case of LLVM, we have instructions such as: **add-nuw** add unsigned with no wrapping, **getelemptr** computing a pointer to an element of a structured type, and **fcmp-ueq** floating point comparison. We are able to abstract over the exact set of instructions because the symbolic treatment of the different instructions is similar[26]. The exact instructions only come into play when we begin adding rewrite laws to simplify terms. At which point, we need to understand the semantics of the source language. The **call** instruction is treated separately from the other instructions so that we may easily perform inlining of symbolic values when necessary.

The GDSA language also includes a term to modify or project out components of the state variables. As described in the previous section, in order to handle effects, we thread state variables through the instructions. A state variable may represent many different side-effects of a computation. The state modification term allows us to specify, precisely what effects each instruction has, and the projection term allows us to focus on just one of these effects. For example, an **alloc** instruction returns a pointer, modifies the heap, and may fail. We can check for failure, or refer to just the pointer, or just the new heap, by projecting out the relevant portion of the result. Our use of state variables is inspired by state monads[14]. We think of the abstract state variables as a record containing different aspects of the machine state, and the modification/projection terms allows us to name those different parts of the state.

The ϕ term is used to represent conditional execution. The term $\phi(c, x, y)$ is equivalent to x if c is equivalent to **true**, and y if c is equivalent to **false**. In our setting of LLVM, **true** and **false** are one-bit integers with values 1 and 0 respectively. A term is equivalent to **true** (**false**) if it is a one-bit integer with value 1 (0).

The final term in the GDSA language is μ. As described in the overview, a μ term indicates that a variable is modified within a loop. During symbolic evaluation we will need to construct an Ω value from the set of relevant recurrence relations. In order to construct the Ω value we need to know where to find the condition

of the loop, and which variable we are representing. The term $\mu(c, x)$ provides us with this information.

Converting a μ term into an Ω value is part of the symbolic evaluation process, which is described in section 4. In order to construct an Ω value from a μ term, we need the set of relevant rewrite rules. That is, a μ term doesn't make sense unless the set of rewrite rules in which we find it contains all of the relevant rewrite rules. Making sure this is always the case is one of the jobs of the compiler from LLVM to GDSA, which we describe in the next section.

3.1 Compiling to GDSA

Validation is done in three steps: compilation which converts LLVM programs to GDSA, then symbolic evaluation which computes symbolic values, and finally, comparison. In this section we describe the compilation step. Figure 3 gives an overview of the compilation process[4].

Compilation begins with an LLVM function definition. The first stage inserts state variables which make the side-effects of the instructions explicit. We described in the overview how we do this for memory: each memory instruction takes in a memory variable, and produces a (possibly) new memory. The same pattern can be extended to other side-effects such as arithmetic overflow and non-termination. This stage of the compiler is simple, but can have a large impact on the number of false alarms. The more precisely the machine state is modeled, the more rewrite laws you can add which will reduce the number of false alarms.

In the next stage, we convert the basic blocks of a function into a collection of sets of rewrite rules. For each basic block, we will have a set of rewrite rules, one for each instruction in the block. We also record the control-flow graph of the function for the next stage.

The third stage of the compilation process is structural analysis. Structural analysis is a control-flow analysis that attempts to compute a structured program from the control-flow graph[23]. This stage of the compiler can fail if the LLVM optimizer produces an irreducible control-flow graph, in which case we cannot validate the function. We perform structural analysis to make sure that we can understand the control-flow graph as conditionals and while loops. One could image alternative strategies that do not involve structural analysis, however we found this approach simple. Also, for the code we are most interested in (safety critical subsets of C), structural analysis is able to categorize all of the control flow.

After structural analysis, we compute the guards for all of the ϕ-nodes. This stage of the compiler could be done much earlier (even first) using the control-flow graph. We placed this stage after structural analysis because the algorithm is very simple with the program structure in hand.

The final stage of the compiler computes the μ nodes for each of the while loops. The algorithm proceeds over the structure of the function. For each while loop, the loop header will have a ϕ-node for each variable that is modified within the loop. The first argument of each ϕ-node has the termination condition of the μ node. The second argument is the initial value of the variable, and the last argument gives us the incremental value of the variable in terms of the previous value. With this information it is a simple matter to construct the μ nodes. This process is simple because the code is in SSA-form, and because we have computed guards for the ϕ-nodes. Note that the problem of computing the smallest set of μ-nodes corresponds to the problem of computing the Minimal SSA-form[4].

[4] We have elided some details of the compiler front-end such as parsing, handling of modules, and detection of inlining, that do not bear on the main ideas.

3.2 Example

Again, applying the compiler to the loop example presented in figure 1a yields the following GDSA program:

$$\left\{\begin{array}{c} x_0 \mapsto c \\ b_0 \mapsto x_0 < n \\ x \mapsto \mu_0(b, x) \\ x_n \mapsto x_{n-1} + 1 \\ b_n \mapsto x_n < n \end{array}\right\}$$

Figure 4: Running example from figure 1a

The first two rules come from the instructions before the loop. The loop defines variable x, therefore, we add one mu rule $x \mapsto \mu_0(b, x)$. As we unfold the definition of μ, indexed versions of b and x appear. These are represented by the two rules $x_n \mapsto x_{n-1} + 1$ and $b_n \mapsto x_n < n$ which are defined for values of n greater than 0. These two rules are computed from the loop body, placing the index according to the information in the ϕ-nodes of the loop header. Finally, note that we do not produce a μ rule for b as its value has to be `false`.

3.3 Discussion

Predicate transformers semantics does not require that we start with a language of guarded dynamic single assignment, as we have done. However, we believe that this choice of first compiling to GDSA before performing symbolic evaluation is interesting for two reasons. First, it is very easy to compute the GDSA representation starting with a program in SSA form. Second, it makes the symbolic evaluation process mostly a matter of rewriting, which we believe leads, in practice, to a simpler algorithm than if we had to maintain a state of symbolic values as is done in previous definitions of symbolic evaluation [26].

4. Generalized symbolic evaluation

After we have compiled a program to GDSA, we can use our symbolic evaluation to produce a symbolic value that we can use to compare programs. We will now describe the symbolic evaluation process.

The syntax of symbolic values is given below. Symbolic values are trees (although, in practice, they are implemented with sharing and form DAGs). The nodes of the trees are instructions, Φ- and Ω-nodes. At the leaves are variables and constants.

$$\begin{array}{lcl} \textbf{expr} & ::= & \textbf{var}^o \\ & | & \textbf{const} \\ & | & \textbf{inst}\ (\overrightarrow{\textbf{expr}}) \\ & | & \Phi(\textbf{expr},\textbf{expr},\textbf{expr}) \\ & | & \Omega_b \overrightarrow{\textbf{expr}}_i \end{array}$$

Our symbolic evaluation produces these values using two distinct transformations. First, it translates a GDSA program into a symbolic value by applying the rewrite rules to a start value; for lack of a better term, we call this process *substitution*. Second, symbolic evaluation applies rewrite laws to simplify the resulting values; we call this process *simplification*. Given a starting value x^o, and a set of rewrite rules $\mathcal{S}$, we can define symbolic evaluation as:

$$\alpha(x^o, \mathcal{S}) = \text{simplify}(\text{substitute } \mathcal{S}\ x^o)$$

4.1 Substitution

For a given term t, substitution proceeds by picking a variable in t that has a rewrite rule in $\mathcal{S}$, and replacing the variable according to

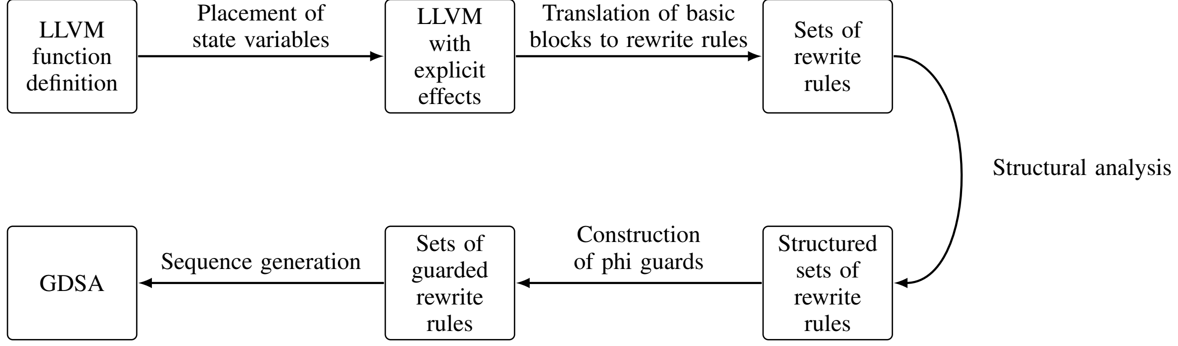


Figure 3: Compilation from an LLVM function down to its GDSA representation.

the rule. More formally, let $\text{Vars}(t)$ be the function that returns the set of variables appearing in t, and $\text{Dom}(\mathcal{S})$ be the set of variables that have a rewrite rule in $\mathcal{S}$. Then, substitution is defined by the pseudo-code below:

substitute $\mathcal{S}$ t =
  pick $x \mapsto u$ **such that** $x \in \text{Vars}(t) \cap \text{Dom}(\mathcal{S})$
  match u **with**
    | $\mu_0(b, x) \Rightarrow$ **let** $v = \text{limit}(\mathcal{S}, b, x)$ **in**
      substitute $\mathcal{S}$ $(t[x \leftarrow v])$
    | _ $\Rightarrow$ substitute $\mathcal{S}$ $(t[x \leftarrow u])$

If the variable is to be replaced by a μ, then we need to compute the symbolic value of the loop (using limit). Otherwise, we can apply the rewrite rule directly, and proceed with substitution.

The function `limit` computes the symbolic value of a loop: an Ω value. To compute an Ω value for a variable x defined with a loop with condition b, we first compute the representation of the sequences for x and b (using `sequence`). Then, we gather up all of the sequences that our initial sequences depend on. This is done by the call to `deps`.

$\text{limit}(\mathcal{S}, b, x)$ =
  let $(b_i, b_n) = \text{sequence}(\mathcal{S}, b)$
  and $(x_i, x_n) = \text{sequence}(\mathcal{S}, x)$ **in**
  let $ds = \text{deps}(\mathcal{S}, b, x)$ **in**
  $\Omega(\{(x_0, x_n), (b_0, b_n)\} \cup ds)$

A sequence of values for a variable, x, is represented by the rules for rewriting each instance of x as the loop progresses; that is, the rules for x_0 and x_n.

$\text{sequence}(\mathcal{S}, x)$ =
  let $t_0 = \text{substitute}(\mathcal{S}, x_0)$
  and $t_n = \text{substitute}(\mathcal{S}, x_n)$ **in**
  (t_0, t_n)

Note, that substitution of x_n may result in occurrences of x_{n-1}, however we will never rewrite those occurrences because we never generate rewrite rules for x_{n-1}. Therefore, the `sequence` function will terminate.

4.1.1 Examples

Recall our running example from figure 4. If we start the symbolic evaluation process with variable x, function `substitute` picks the rule $x \mapsto \mu_0(b, x)$. This GDSA term cannot be substituted, and function `limit` is called. The sequence computed (by the `sequence` function) for x_n^o and b_n^o are:

$$x_n^o = \begin{cases} c & \text{if } n = 0, \\ x_{n-1}^o + 1 & \text{otherwise.} \end{cases}$$

$$b_n^o = \begin{cases} x_0^o < n^o & \text{if } n = 0, \\ x_n^o < n^o & \text{otherwise.} \end{cases}$$

These sequences do not depend on any other recursive variables, so we don't need to compute any more dependencies. The symbolic value for x is therefore:

$$\Omega[(c, x_{n-1} + 1), (x_0 < n, x_n < n)] \quad .$$

To further clarify the symbolic evaluation of loops, consider the following example (for clarity, we use a C-like syntax instead of LLVM assembly).

```
x = 0; y = 0; z = 0;
for (i = 0; i < k; ++i) {
  x = 2 \times x;
  z = x + y;
  y = 2 \times y;
}
return z;
```

The translation to GDSA gives the start variable z, and the following set of rewrite rules (the variable c is generated by the LLVM compiler to represent the condition).

$$\left\{ \begin{array}{lll} x_0 \mapsto 0 & x_n \mapsto 2 \times x_{n-1} & \\ y_0 \mapsto 0 & y_n \mapsto 2 \times y_{n-1} & x \mapsto \mu_0(c, x) \\ z_0 \mapsto 0 & z_n \mapsto x_n + y_{n-1} & y \mapsto \mu_0(c, y) \\ i_0 \mapsto 0 & i_n \mapsto i_{n-1} + 1 & z \mapsto \mu_0(c, z) \\ c_0 \mapsto i_0 < k & c_n \mapsto i_n < k & \end{array} \right\}$$

The symbolic value of z is

$$\begin{array}{ll} \Omega[(0, (2 \times x_{n-1}) + y_{n-1}), & \text{sequence for z} \\ \quad (i_0 < k, i_n < k), & \text{sequence for c} \\ \quad (0, 2 \times x_{n-1}), & \text{sequence for x} \\ \quad (0, 2 \times y_{n-1}), & \text{sequence for y} \\ \quad (0, i_{n-1} + 1)] & \text{sequence for i} \end{array}$$

where the first two sequences have been computed by limit, and the other sequences are the dependencies of the first two, ordered by appearance. It is important to note that the sequences of symbolic values for x and y do not refer to each other: if they were not both used by z, those two sequences would be separated. This is why we are able to validate loop fusion and fission: to compute the symbolic value of the body, we completely unfold the loop until the value is expressed only in terms of variables indexed by $n - 1$, and hence we know which sequences are depended upon.

4.1.2 Implementation Details

So far, in our Ω notation we have used the names of the variables appearing in the source program to represent the relevant sequences. While this is adequate, it does not permit an effective *syntactic* equality on Ω values. Therefore, our implementation uses a different representation. In our implementation, all of the sequence variables appearing under an Ω are replaced with de Bruijn indices. The sequence for the variables being defined is always placed first, followed by the sequence for the condition. Then, we place other relevant sequences in the order that they appear in the dependency computation. Using this representation allows us to check equality between Ω values using a very efficient, syntactic check.

5. Simplification laws

The simplification process is responsible for applying the rewrite laws. The simplification process rewrites a symbolic value from the bottom up, applying rewrite laws when applicable. We write our rewrite laws using the symbol $\downarrow$. For instance, there are three laws for reducing Φ-terms.

$$\Phi(\texttt{true}, t, _) \downarrow t \qquad (1)$$

$$\Phi(\texttt{false}, _, t) \downarrow t \qquad (2)$$

$$\Phi(_, t, t) \downarrow t \qquad (3)$$

These laws are required to validate sparse conditional constant propagation (SCCP) and global value numbering (GVN). The following example can be optimized by both:

```
if (c) {a = 1; b = 1;} else {a = 2; b = 2;}
if (a == b) {x = 1;} else {x = 2;}
return x;
```

Applying global-value numbering followed by sparse conditional constant propagation transforms this program to `return 1;`. Indeed, in each of the branches of the first conditional statement, a is equal to b. Since $a == b$ is always true, the condition of the second if statement is constant, and sparse conditional constant propagation can propagate the left definition of x. The above program and `return 1` have the same normalized symbolic value, computed as follows ($\rightarrow$ denotes a step of substitution):

$$\begin{array}{lll} & x^o & \\ \rightarrow & \Phi(a^o == b^o, 1, 2) & \\ \rightarrow & \Phi(\Phi(c,1,2) == b^o, 1, 2) & \\ \rightarrow & \Phi(\Phi(c,1,2) == \Phi(c,1,2), 1, 2) & \\ \downarrow & \Phi(\texttt{true}, 1, 2) & \text{(by 3)} \\ \downarrow & 1 & \text{(by 1)} \end{array}$$

There are also two laws for simplifying Ω terms. The first says that if we have an Ω for a variable, and the sequence for that variable is not recursive (it does not mention any variable with an index), then we can replace the Ω with a Φ[5]. This law is written formally below where both x and y are not recursive.

$$\Omega[(x,y),(b_0,...),...] \downarrow \Phi(b_0, x, y) \qquad (4)$$

This law is necessary to validate loop invariant code motion. As an example, consider the following program:

```
x = a + 3; c = 3;
for (i = 0; i < n; i++) {x = a + c;}
return x;
```

In this program, variable x is defined within a loop, but it is invariant. Moreover, variable c is a constant. Applying global constant propagation, followed by loop-invariant code motion and loop deletion (a form of dead code elimination) transforms the program to `return (a + 3);`. the symbolic value is computed as follows:

$$\begin{array}{lll} & x^o & \\ \rightarrow & \Omega[(a^o+3, a^o+3),(0, i_{k-1}+1)] & \\ \downarrow & a^o + 3 & \text{(by 4)} \end{array}$$

Note that the symbolic evaluation transforms the μ into an Ω. The Ω contains the sequence describing the values of x within the loop. The variable c^o, as it does not depend on the loop computation is replaced by its value, 3. This shows how an optimization such as global constant propagation is validated (validating global constant propagation does not require any specific law, nor does global copy propagation, common sub-expression elimination, or global scheduling).

The second law for Ω is a bit more complex. The law says that if an Ω term has a sequence that is not recursive, then that sequence can be removed from the Ω. In order to "remove" a sequence, we have to substitute the body of the sequence into the other sequences and rebuild the Ω-term. A formal statement of this rule is given below, where the sequence at position k is not recursive.

$$\Omega[s_0, ...s_k, s_{k+1}, ...s_n] \downarrow \Omega[s_0', ...s_{k-1}', s_{k+2}', ...s_n']$$

where

$$s_i' = s_i[k \mapsto s_k, k+1 \mapsto s_{k+1}]$$

In addition to these kinds of fundamental laws, we also have a number of rewrite laws that are derived from the semantics of LLVM. For example, we have a family of laws of the form:

$$\texttt{add}\ a\ b \downarrow a + b$$

$$\texttt{mul}\ a\ b \downarrow a * b$$

where a and b are constants. There are also laws for reducing instructions, such as:

$$\texttt{add}\ a\ a \downarrow \texttt{shl}\ a\ 1$$

We also have laws that handle optimizations that make use of aliasing information. For example, we have the following two laws for simplifying loads:

$$\texttt{load}(p, \texttt{store}(y, q, \texttt{store}(y, p, m))) \downarrow \texttt{load}(p, \texttt{store}(x, p, m)) \qquad (5)$$

$$\texttt{load}(p, \texttt{store}(x, p, m)) \downarrow x \qquad (6)$$

when p and q are not aliasing. Our validator can use the result of a may alias analysis. For now, we only use simple non-aliasing rules: two pointers that originate from two distinct stack allocations may not alias; two pointers forged using `getelemptr` with different parameters may not alias, etc. The following example shows how we can use these laws.

```
int[2] t;
t[0] = 1; t[1] = 2;
return t[0];
```

In this case, $t[0]$ and $t[1]$ are different addresses, so writing at address $t[1]$ does not affect a read at $t[0]$. Applying constant propagation using alias information, this program transforms to `return 1;`. The symbolic value is computed as follows:

$$\begin{array}{lll} & \texttt{load}(\&t^o, m_2^o) & \\ \rightarrow & \texttt{load}(\&t^o, \texttt{store}(\&(t^o+1), 2, m_1^o) & \\ \rightarrow & \texttt{load}(\&t^o, \texttt{store}(\&(t^o+1), 2, (\texttt{store}(\&t^o, 1, m^o)) & \\ \downarrow & \texttt{load}(\&t^o, \texttt{store}(\&t^o, 1, t^o) & \text{(by 5)} \\ \downarrow & 1 & \text{(by 6)} \end{array}$$

We use $\&t$ for the address of t *in lieu* of the `geteltptr` instruction of LLVM to simplify the presentation. The "m" variables are the abstract state variables introduced by our compiler.

[5] When we extend our implementation to non-termination effects, this rule will need to be redesigned to not overlook possible non-termination or runtime errors.

Finally, if the program does not contain recursive functions, as would be the case if it was written following safety-critical software guidelines, we can also replace each function call by its symbolic value. This allows us to validate inter-procedural optimizations and inlining. Consider for example the following program:

```
int f(int x, int y) {
 int z;
 if (x == y) { z = 4;} else { z = 81; };
 return z;
}

int main(int u,int v,int argc) {
 int x, y, z, i, a, b;
 a = u + 4; z = 4;
 if (a < v) { x = 1; y = x; }
    else {x = 2; y = 2;};
 for (i = 0; i < argc; i++) { z = f(x,y); };
 return (a + (u + z));
}
```

This program can be optimized in several ways. Inlining f into g allows us to optimize $main$ with global value-numbering, loop-invariant code motion, global common-subexpression elimination, and constant folding. This could transform the program into `return ((u + 4) << 1);`, where f becomes dead code. This result could also be achieved with interprocedural versions of those optimizations.

We can validate these optimizations by replacing the call to f with its symbolic value. If we symbolically inline f in this way, then this example reduces to our previous examples.

5.1 Efficiency, effectiveness

The simplification laws that are derived from LLVM semantics are designed to mirror the kinds of rewritings that are done by the LLVM optimization pipeline. In theory we could design a unique normal form for symbolic values, and use this normal form for comparisons. However, in practice, it is *much* more efficient to transform the symbolic values in the same way as the optimizer. So, if we know that LLVM will prefer `shl` a 1 to $a + a$, then we will rewrite $a + a$ but not the other way around.

As another, more extreme, example, consider the following C code, and two possible optimizations: SCCP and GVN.

```
a = x < y;
b = x < y;
if (a) {
   if (a == b) {c = 1;} else {c = 2;}
} else {c = 1;}
return c;
```

If the optimization pipeline is setup to apply SCCP first, then a may be replaced by `true`. In this case, GVN can not recognized that a and b are equal, and the inner condition will not be simplified. However, if GVN is applied first, then the inner condition can be simplified, and SCCP will propagate the value of c, leading to the program that simply returns 1. The problem of how to order optimizations is well-known, and an optimization pipeline may be reordered to achieve better results. If the optimization pipeline is configured to use GVN before SCCP, then, for efficiency, our simplification should be setup to simplify at join points before we substitute the value of a.

Effectiveness. The goal of a generic translation validator is to validate all of the transformations a real-world compiler (such as LLVM) will make. Although we do not know how to characterize precisely and rigorously what transformations should be validatable by our translation validator, this is not an uncommon situation. Translation validators largely rely on empirical evidence as to their effectiveness. Thankfully this is a quality-of-engineering issue, and not one of correctness. In the next section we will present our experimental results.

Note, however, symbolic evaluation at the level of extended blocks is empirically know to be very effective. As said in the introduction, it is resilient to syntactic changes such as: order of independent instructions, renaming of local registers, and reuse of independent but identical computations. Many optimizations at the level of extended blocks are a combination of these transformations, which is why symbolic evaluation is such an effective tool for translation validation.

We claim, that our generalization of symbolic evaluation to loops preserves this quality. Indeed, as sketched in the previous section through examples, our generalization is resilient to: order of independent instructions, renaming of local registers, and reuse of independent but identical computations. Many global optimizations are a combination of these transformations, as noted by Kanade, et al. [10]. Which is why we believe that our approach will prove to be an effective tool for translation validation.

6. Experimental validation

Our validation tool, LLVM M.D., is implemented in Haskell[9]; the complete source code can be downloaded from our project website[6]. The implementation is conceptually simple. However, dealing with all of the details of LLVM leads to some complexity. Even so, the core data structures and algorithms have been implemented in under 4000 lines of code. There are two interesting things to point out about our implementation. First, the symbolic values are represented with some sharing. Second, our implementation is designed to take advantage of aliasing information during simplification.

Using our prototype implementation of LLVM M.D., we have run three experiments. First, we have a number of difficult, hand-written examples that we have optimized using LLVM or by hand. These hand-written examples are designed to test extreme cases of both correct and incorrect optimizations. Those examples include optimizations such as global constant and copy propagation and folding, common-subexpression elimination with global value numbering or lazy code motion, list, trace and global scheduling, loop fission and fusion, loop invariant code motion, loop deletion, sparse conditional constant propagation, and loop unswitching.

Our second experiment uses the sqlite3 embedded database library. This library is comprised of approximately 100,000 lines of C code in over 1300 different functions. We optimize the sqlite3 library using different optimizations and try to validate each transformed function. The results we have obtained are summarized in Table 1, where GVN stands for global value numbering, SCCP stands for sparse conditional constant propagation, LICM stands for loop invariant code motion, DCE stands for dead code elimination, and LD stands for loop deletion (please see our project website for updated results). Note that the current implementation of LLVM uses alias analysis, and eliminates some partially redundant computations and redundant loads. We believe that it implements an AWZ-like GVN analysis.

The second column of Table 1 is the number of functions for which we observe one or more transformations. For instance, for GVN (using alias information), we have observed that 350 functions were transformed, of which we validated 326, and produced 24 alarms. In our initial experiments, we had more false alarms. For GVN, most of these false alarms came from a lack of simplification rules which make use of aliasing information. For instance, in the following code:

[6] `http://llvm-md.seas.harvard.edu`

	# optimized functions	# Validated optimizations	# Alarms	Ratio
GVN	350	326	24	≈ 93%
SCCP	42	35	7	≈ 83%
LICM	79	76	3	≈ 96%
DCE	43	43	0	100%
LD	25	25	0	100%

Table 1: Results of experiments on LLVM optimization for sqlite3

```
*p = 4; f(q); x = *p;
```

GVN can replace `x = *p` by `x = 4` if it can statically determine that we cannot transitively reach `p`'s memory region through pointer `q`. For SCCP, the false alarms result from the lack of arithmetic laws. While we predicted that we would have false alarms on for GVN and SCCP that require more laws, we were surprised by the presence of the 3 false alarms in the loop invariant code motion experiment. It turns out that 2 of those have an interesting cause: the optimization that LLVM applied was not really a loop invariant code motion, but rather the scalar promotion of a memory slot. We could validate such an optimization, but it has not yet been our focus. The lesson is that in a "real" compiler, we may not be able to focus on one optimization at a time, and we must be ready to deal with any valid transformation at any time. This gives us a lot of hope that our approach will be effective. The third alarm appears to be from a bug in our front-end that we have not yet fully diagnosed. Another lesson is that understanding the origin of an alarm can be time consuming. In the future, we plan to develop tools to automate the process of understanding alarms, as was done for the ASTRÉE analyzer [22].

Finally, for our third experiment, we have used LLVM M.D. to validate optimizations on C code produced by The Mathworks Real-time workshop from a combined Stateflow/Simulink model [13]. The model is a classic example of the temperature of a plant along with it's cooling system in Simulink, and a control-command program that takes as input the current temperature and decides when fans should be turned on or off. The important conclusion from this experiment is that for such code, that respects the safety-critical guidelines for C, we can restructure every function and apply the validator, even for inter-procedural optimization.

Overall, our validator caught one mis-compilation. In one experiment, we used `llvm-extract` to break an object file up into its individual functions, optimized each function, and then recombined them. Our validator caught a number of faulty dead-store eliminations caused by the LLVM tools incorrectly marking global data as private in the individual object files. On close reading of the documentation, it is not clear (to us) if the tools are meant to be used in this way. However, thanks to LLVM M.D., we quickly realized that we cannot use the tools this way and preserve semantics.

During the course of our experimentation, we realized that our tool could be invaluable for understanding how to properly map a higher-level language onto a system like LLVM. The correctness of a memory optimization depends on the semantics of the programming language, more precisely on its memory model. If a language has a C-like memory model, then compiling to LLVM and applying memory optimizations based on alias analysis is safe, but it may not always be the case. The designer of a language that uses LLVM for compilation must translate down to LLVM code, but it can be treacherous. Some optimizations we have observed (which initially raised false alarms) having to do with memory, function calls, and aliasing, are difficult to justify, and use precise properties of LLVM's memory model. Thankfully, LLVM M.D. can point out such optimizations, helping the front-end designer close the gap between the two languages.

In the long run, we believe that a tool such as LLVM M.D. could play an important role, maybe not just to catch errors in well-understood, well-tested optimizations, but rather in the interaction between all the analyzes, transformations, and tools that are provided by an open compilation framework such as LLVM.

7. A semantics for GDSA

Ideally, we would like to formally prove that whenever LLVM M.D. does not find discrepancies between the symbolic values of a program and its optimized version, then the semantics are preserved. However, our main goal, thus far, has been to scale translation validation to modern compilation frameworks. In our setting, a proof would require a formal semantics for LLVM—a difficult and daunting task. We could build such a proof by proving that the translation from LLVM to normalized symbolic values preserves semantics. However, there may be a more elegant and automatable way that builds on previous results from formal verification of symbolic evaluation.

The crux of idea is that we can equip the GDSA language with a formal semantics that describes the execution of a GDSA program as a two-fold process: first, building symbolic values, and then computing their denotation. We first explain this semantics, and then discuss why it could be of interest.

7.1 A semantics for GDSA

We define the predicate $\mathcal{S} \vdash t \Rightarrow v$ where ρ is an assignment of the input variables, $\mathcal{S}$ is a set of rewrite rules, t is the current symbolic value, and v is the final symbolic value. We have decided to use a natural semantics because of its simplicity. The downside is that, for now, we only model terminating executions.

The symbolic value t only contains ϕ and μ symbols at its leafs, never as nodes. This results from the use of the symbolic context presented in figure 5a, which makes it impossible to substitute within a ϕ or a μ. The final symbolic value, **v**, will not have ϕ- or μ-nodes.

$$
\begin{array}{lcl}
\mathbf{v} & ::= & \mathbf{const} \mid \mathbf{inst}\ (\overrightarrow{\mathbf{v}}) \\
\mathbf{t} & ::= & \mathbf{v} \\
 & \mid & \mathbf{inst}\ (\overrightarrow{\mathbf{t}}) \\
 & \mid & \phi(\mathbf{var},\mathbf{var},\mathbf{var}) \\
 & \mid & \mu_i(\mathbf{var},\mathbf{var})
\end{array}
$$

The semantics is presented in figure 5. Rule (1) formalizes how execution begins and terminates. Given a GDSA program $(x, \mathcal{S})$, we start the execution with the symbolic value x^o, in an environment of rewrite rules containing $\mathcal{S}$ and one rule for each variable of the environment ρ such that $x \mapsto c$ if $\rho(x) = c$. This execution returns a control-free symbolic value, and we return the denotational semantics of this symbolic value. Rule (2) finds a variable in the symbolic value that has an associated rewrite rule, and replaces the variable with its definition. Rule (3) and (4) describe the execution of a ϕ-node. Contrary to symbolic evaluation, we do not rewrite those variables. Instead, we completely evaluate the condition, and then proceed with rewriting one definition or the other. Rule (5) shows that a μ-leaf is unfolded into its definition. $\mu_k(b, x)$ becomes $\phi(b_k, \mu_{k+1}, x_k)$, and the execution can proceed by evaluating the condition, which decides whether or not to continue with the loop.

$$\mathbf{E} ::= [] \mid \mathbf{inst}(\ldots, \mathbf{E}, \ldots)$$

(a) Evaluation contexts

$$[\![c]\!] = c \qquad [\![\mathbf{inst}(\overrightarrow{e})]\!] = \mathbf{inst}(\mathbf{map}([\![\cdot]\!])\overrightarrow{e})$$

(b) Denotational semantics of closed expressions

$$\frac{\mathcal{S} \cup \rho \vdash x^o \Rightarrow v}{\rho \vdash (x, \mathcal{S}) \Rightarrow [\![v]\!]} \ (1) \qquad \frac{X \mapsto \mathbf{inst}(\overrightarrow{y}^o) \in \mathcal{S} \quad \mathcal{S} \vdash E[\mathbf{inst}(\overrightarrow{y}^o)] \Rightarrow v}{\mathcal{S} \vdash E[x^o] \Rightarrow v} \ (2)$$

$$\frac{\mathcal{S} \vdash b^o \Rightarrow v_b \quad [\![v_b]\!] = \mathtt{true} \quad \mathcal{S} \vdash E[x^o] \Rightarrow v,}{\mathcal{S} \vdash E[\phi(b^o, x^o, y^o)] \Rightarrow v} \ (3) \qquad \frac{\mathcal{S} \vdash b^o \Rightarrow v_b \quad [\![v_b]\!] = \mathtt{false} \quad \mathcal{S} \vdash E[y^o] \Rightarrow v}{\mathcal{S} \vdash E[\phi(b^o, x^o, y^o)] \Rightarrow v} \ (4)$$

$$\frac{\mathcal{S} \vdash E[\phi(b_n^o, \mu_{n+1}(b, x), x_n^o)] \Rightarrow v}{\mathcal{S} \vdash E[\mu_n(b, x)] \Rightarrow v} \ (5) \qquad \frac{}{\mathcal{S} \vdash v \Rightarrow v} \ (6)$$

Figure 5: Dynamic semantics of the guarded dynamic single assignment intermediate language.

Rule (6) states that the rewritting process stops when there are no more rewritable variables, ϕ-nodes or μ-nodes.

While the result of symbolic evaluation represents all the symbolic executions of a function, the semantics we have equipped GDSA with builds the symbolic value for only one trace. This is why the symbolic values in the GDSA semantics do not have Φ- or Ω-nodes: the control has been determined. This restricted set of symbolic values and their denotation is well-understood and has been formally studied.

The correctness of symbolic evaluation can then be stated as follows.

Property 1. *Given two sets of rewrites rules $\mathcal{S}$ and $\mathcal{S}'$, and two variables x and y. If $\alpha(x^o, \mathcal{S}) = \alpha(y^o, \mathcal{S}')$, then for all environments ρ and all symbolic values v, $\rho \vdash (x, \mathcal{S}) \Rightarrow v$ if and only if $\rho \vdash (y, \mathcal{S}') \Rightarrow v$.*

Even though the proof of the above property is conceptually simple, it is likely to be complex because we need to take into account every rewrite law. We have sketched proofs taking into account a few rewrite laws. However, in the long run, a proof assistant should be used to automate this proof.

It may be worth noting that if we prove this property, then to prove semantics preservation of the whole system, we only need to give a denotational semantics to symbolic values that do not contain Φ nor Ω nodes, as they appear in the GDSA semantics. Such denotational semantics has already been studied, and its properties have been formally verified [27]. The properties about Φ and Ω required to prove the above theorem are about syntatic unfolding, which is very *a propos* since they are meant to be uninterpreted symbols and since in GDSA, the control is not symbolic.

8. Related work

Researchers have used symbolic evaluation for a long time in several contexts, such as: instance testing, bug finding, and generation of verification conditions. As mentioned by Necula [16] symbolic evaluation has appeared in history under several disguises such as predicate transformers or value-dependence graphs.

Starting with Necula [16] symbolic evaluation has also been used to build translation validators. In Necula's work, symbolic evaluation is limited to extended blocks, and the validator has to resort to dataflow analysis and must compute a simulation relation to handle global optimizations. Rival [21] uses a variant of symbolic evaluation called a transfer function to design a translation validator that handles the whole compilation pipeline and is formalized by abstract interpretation. Again, symbolic evaluation in this work is limited to extended blocks, and the validator must resort to dataflow analysis and a simulation relation to validate global transformations. Tristan and Leroy [26] have used symbolic evaluation to implement formally verified translation validators for list and trace scheduling, optimizations limited to extended blocks. The same two authors [27] also showed how symbolic evaluation can be used to design a translation validation algorithm for software pipelining, a specific loop optimization. While their work validates loop transformations, the validators are tailored to software pipelining, and do not validate global optimizations. The work presented in this paper is the first where symbolic evaluation is generalized to handle control-flow graphs with while loops in such a way that global optimizations can be validated without resorting to dataflow analysis.

Another line of research on translation validation has followed the approach sketched by Pnueli and others [19] and developed in the TVOC validator [2, 28]. The most advanced validator following this line of work is the one designed by Tatlock [11, 25]. We believe that this kind of translation validator and those based on symbolic evaluation may have roughly the same validation capabilities as ours, however they differ in the amount of necessary configuration to produce a complete and efficient validator. We believe that setting up rewrite laws may be easier than setting up a simulation relation and dataflow analyses tailored to a specific optimization, but this is yet to be demonstrated in practice.

There also exists a lesser known work on translation validation [10] where the authors make use of the observation that advanced optimizations often boil down to simple rewritings of the control-flow graph. They have instrumented the GCC compiler to output a trace of all the transformations applied to a function's control-flow graph, and they check, using the PVS model checker, that each of the rewrites are valid. The complexity of such an algorithm is high, but it is likely that a practical generic translation validator will use a subtle mix of symbolic evaluation, dataflow analysis, and compiler instrumentation.

9. Future work

We have already mentioned a few of the improvements that we plan to make to our prototype implementation. In the short term, we must improve the precision of our model for non-termination and runtime errors, which will allow more rewrite laws and potentially less false alarms. We must also design tools to help interpret the origin of the alarms. Finally, we could improve the efficiency of the tool through a better implementation of sharing.

In the middle term, we would like to use an automated theorem prover such as Z3 to discharge equalities between symbolic values. First, this will help us with optimizations such as reassociation that make it difficult to efficiently compute a normal form. Also, when

we cannot ascertain that the semantics of two programs are the same, we would like to use Z3 to try to find an example of inputs that lead to the discrepancy. Following the ideas implemented in the snugglebug tool [3], we could try to find a model for the negation of the equivalence statement.

In the long term, we would like to have a formal proof of correctness of our validator, if only for a subset of LLVM. Also, as already mentioned, if we restrict our attention to C code that satisfies the guidelines for safety-critical software, we can build an entirely validated compiler from C to assembly that uses LLVM as an optimization pipeline.

As a side note, it may be worth noting that with such a tool, one can use program translations that have a low probability of being incorrect while having a much better complexity, such as global value numbering based on random interpretation [7]. If validated, an incorrect transformation can be caught, and the transformation can to be redone. The combination of random interpretation for global value numbers with our validator would always yield correct optimizations, but its execution time would be probabilistic, and may be better than the execution time of the classical global value numbering algorithm.

10. Conclusion

We have presented a new translation validator design that extends previous uses of symbolic evaluation, leading to what we call a denotational translation validator. To test our design, we have implemented a tool, LLVM M.D., that we use to validate transformations on LLVM code. In the end, our translation validator is conceptually simple, and easy to implement with only a few thousand lines of code. We believe that our design will scale to real-world compilers, and our preliminary experimental results are promising.

References

[1] The Motor Industry Software Reliability Association. Guidelines for the use of the c language in critical systems. http://www.misra.org.uk, 2004.

[2] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification, 17th Int. Conf., CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.

[3] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *Proceedings of the 19 Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 363–374. ACM, 2009.

[4] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufman, 2004.

[5] The LLVM development team. The llvm compiler infrastructure. http://llvm.org, 2010.

[6] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communication of the ACM*, 18(8):453–457, 1975.

[7] Sumit Gulwani and George Necula. Global value numbering using random interpretation. In *31st symposium Principles of Programming Languages*, pages 342–352. ACM, 2004.

[8] Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *Static Analysis, 13th Int. Symp., SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 281–300. Springer, 2006.

[9] Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 1998.

[10] Aditya Kanade, Amitabha Sanyal, and Uday Khedker. A PVS based framework for validating compiler optimizations. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 108–117. IEEE Computer Society, 2006.

[11] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI '09: Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 327–337. ACM, 2009.

[12] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[13] The Mathworks. http://www.mathworks.com/automotive/standards/misra-c.html.

[14] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23. IEEE, 1989.

[15] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[16] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.

[17] George C. Necula and Peter Lee. The design and implementation of a certifying compiler (with retrospective). In *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pages 612–625. ACM, 2004.

[18] Amir Pnueli and Anna Zaks. Validation of interprocedural optimization. In *Proc. Workshop Compiler Optimization Meets Compiler Verification (COCV 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[19] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[20] Martin Rinard and Darko Marinov. Credible compilation with pointers. In *Workshop on Run-Time Result Verification*, 1999.

[21] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.

[22] Xavier Rival. Understanding the origin of alarms in `astrée`. In *Lecture Notes in Computer Science*, pages 303–319. Springer, 2005.

[23] Micha Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3/4):141–153, 1980.

[24] The sqlite development team. sqlite. http://www.sqlite.org, 2010.

[25] Zachary Tatlock and Sorin Lerner. Bringing extensibility to certified compilers. In *Proceedings of the 20 Conference on Programming Language Design and Implementation (PLDI 2010)*, pages 111–121. ACM, 2010.

[26] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17–27. ACM Press, 2008.

[27] 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.

[28] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.