Denotational Translation Validation

Thesis advisor

Greg Morrisett

Author

Paul Govereau

Denotational Translation Validation

Abstract

In this dissertation we present a simple and scalable system for validating the correctness of low-level program transformations. Proving that program transformations are correct is crucial to the development of security critical software tools. We achieve a simple and scalable design by compiling sequential low-level programs to synchronous data-flow programs. Theses data-flow programs are a *denotation* of the original programs, representing all of the relevant aspects of the program semantics. We then check that the two denotations are equivalent, which implies that the program transformation is semantics preserving. Our denotations are computed by means of symbolic analysis. In order to achieve our design, we have extended symbolic analysis to arbitrary control-flow graphs. To this end, we have designed an intermediate language called Synchronous Value Graphs (SVG), which is capable of representing our denotations for arbitrary control-flow graphs, we have built an algorithm for computing SVG from normal assembly language, and we have given a formal model of SVG which allows us to simplify and compare denotations. Finally, we report on our experiments with LLVM M.D., a prototype denotational translation validator for the LLVM optimization framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The central thesis of this dissertation is that we can provide a simple and scalable system for validating the correctness of low-level program transformations. Proving that program transformations are correct is crucial to the development of security critical software tools: in particular program optimizers. Checking translations is also an important compiler development tool. Having an effective way of checking translations can help compiler developers identify bugs in new program transformations. Finally, constructing a provably correct compiler is an important benchmark for certified programming.

As evidence to support our thesis, we have designed and built a software tool that validates transformations by compiling sequential low-level programs to synchronous dataflow programs. Theses dataflow programs are a *denotation* of the original programs. The denotations represent all of the relevant aspects of the program semantics. We then check that the two denotations are equivalent, which implies that the program transformation is semantics preserving. We will describe our design in detail

below. First, to motivate our design, we will consider other approaches to validating transformations.

One path to building a provably correct transformation is to prove, directly, the correctness of the transformation algorithms. That is, for a given transformation, $\mathcal{O}$, prove a theorem of the form:

$$\forall p. \vdash [\![p]\!] \equiv [\![\mathcal{O}(p)]\!] \quad .$$

In the above, p is a program and $\mathcal{O}(p)$ is the program after the transformation $\mathcal{O}$ has been applied. The bracket notation, $[\![p]\!]$, represents the semantics of the program p. The notation $\vdash \cdot \equiv \cdot$ represents a proof of equivalence. Thus, we can read this theorem as "for all programs p, the semantics of p and its transformed counter-part are equivalent." Note that this theorem is valid *for all* programs. While this is a powerful theorem, it is also difficult to prove in practice. Experience has shown that in a real-world setting, many complex optimizations resist direct formal correctness proofs of this form (Tristan and Leroy, 2010). However, this approach is not practical for low-level optimizations for another reason. Namely, real-world compilation frameworks are continually changing, and keeping a complex proof of correctness up to date with changes in the compiler seems impractical.

An alternative approach is to relax the constraints on the theorem we prove. Rather than proving a translation is correct for all programs, we can try to prove the translation is correct for each program we encounter. A system with this general design is called a *translation validation* system. More specifically, translation validation is a static analysis that, given two programs, tries to verify that the programs have the same semantics (Pnueli et al., 1998). Hence, we validate the correctness of

translations by inspecting the original and transformed programs on a case-by-case basis.

Several translation validator designs have been proposed. In fact, translation validation has proved to be particularly useful for validating the transformations done by optimizing compilers. Successes range from generic translation validators for moderately-optimizing industrial-strength compilers (Barrett et al., 2005; Necula, 2000; Rival, 2004), to special-purpose translation validators for advanced optimizations (Huang et al., 2006; Pnueli and Zaks, 2008; Tristan and Leroy, 2010). Translation validation is on the verge of becoming a critical tool, both for formal verification of certifying (Necula and Lee, 2004) and certified compilers (Leroy, 2009). In addition, translation validation is useful for compiler engineering where it can greatly simplify debugging and improve testing (Rinard and Marinov, 1999). However, even with all of these positive results, translation validators that can be applied real-world compilers have not been available.

The primary reason that translation validators have not as successful as they could be is they are too tightly coupled with the transformations. In general, previous translation validation systems work by discovering and verifying a simulation relation. That is, previous systems attempt to prove theorems of the form:

$$\mathcal{O} \models \alpha \Rightarrow \ \alpha \vdash [\![p]\!] \equiv [\![\mathcal{O}(p)]\!] \qquad .$$

Where α is a relation between the values of the original and transformed program. For instance, in the validator of Necula (2000), α is a mapping between variables names mapping the original program variables to optimized program variables. The relation α is computed with knowledge of the transformation $\mathcal{O}$. Given the relation α,

we have an equivalence between the original and transformed program. Note that this theorem (if proved) is only valid for a specific optimization, $\mathcal{O}$, and specific program, p.

Typically, the relation α is computed by means of a data-flow analysis on the two programs, taking into account specifics of the transformation. Our design strives to eliminate this dependence of the transformation $\mathcal{O}$. Hence, we will prove theorems of the form:

$$\forall \mathcal{O}. \vdash [\![p]\!] \equiv [\![\mathcal{O}(p)]\!] \quad .$$

Indeed, since we are independent of the transformation, we may apply our system to any number of optimizations:

$$\forall \mathcal{O}, \mathcal{O}'. \vdash [\![p]\!] \equiv [\![\mathcal{O}'(\mathcal{O}(p))]\!] \quad .$$

This flexibility allows us to use our validator to validate a single or a whole pipeline of optimizations at once.

In order to achieve our design, we must overcome a difficulty related to the denotations of programs. In the above systems, the denotation of a program, $[\![p]\!]$, is computed by means of symbolic analysis. That is, $[\![p]\!]$ is a symbolic form of the program p. These symbolic values are denotations in that they admit an equational theory such that equal symbolic values represent equivalent programs. Ideally, the theory of equality is syntactic allowing for a simple implementation. However, traditional symbolic analysis and related theories of equality do not extend to looping code. Therefore, a crucial aspect of our system is an extension of symbolic analysis to arbitrary control-flow graphs. Specifically, we have designed an intermediate language called Synchronous Value Graphs (SVG), which is similar to many data-

flow languages. The SVG language is capable of representing our denotations for arbitrary control-flow graphs. We have built an algorithm for computing SVG from normal assembly language. This algorithm corresponds to the denotation function $[\![\cdot]\!]$ above. Finally, we have given a formal model of SVG which allows us to simplify and compare denotations.

Using these tools, we present a translation validator design that can effectively validate the optimizations of a production compiler. For a production compiler, we chose the LLVM Framework. To be effective, we believe our validator must satisfy the following criteria. First, we must treat the compiler as a "black box"; we do not want to instrument the optimizer in any way. LLVM has a large collection of program transformations that are updated and improved at a frantic pace. Second, we do not want to modify the source code of the input programs. This means that we handle the output of the optimizer "as is." Third, we want to run only one pass of validation for the whole optimization pipeline. This is important for efficiency, and also because the boundaries between different optimizations are not always firm. Finally, it is crucial that the validator can scale to large functions. For instance, the experiments of Kanade et al. (2006) are very impressive and show an exceptionally low rate of false alarms. However, their validator requires heavy instrumentation of the compiler, and the authors admit that their approach does not scale beyond a few hundred instructions. Thus, this design does not fit our criteria. On the other hand, our design requires no instrumentation of the compiler and, in our experiments, we routinely deal with functions having several thousand instructions.

Perhaps, the most important criteria is the one of instrumentation. To our knowl-

edge, only one prior work has proposed a solution that does not require instrumentation of the compiler. Necula (2000) evaluated the effectiveness of a translation validator for GCC 2.7 with common-subexpression elimination, register allocation, scheduling, and loop inversion. The validator is simulation-based: it verifies that a simulation-relation holds between the two programs. Since the compiler is not instrumented, this simulation relation is inferred by collecting constraints which often take advantage of special knowledge about how the optimizations are implemented. The experimental validation shows that this approach scales, as the validator handles the compilation of programs such as GCC or Linux. However, adding other optimizations such as loop-unswitch or loop-deletion is likely to break the collection of constraints. On the other hand, our design does not make use of subtle details about the internals of the compiler; knowledge about how the optimizer behaves is expressed as simple rewrite laws which can be added to the system as needed.

1.1 Design

While in general not decidable, translation validation can be done efficiently if one program is a compiled or optimized version of other. Previous works (Kundu et al., 2009; Necula, 2000; Rival, 2004; Zuck et al., 2003) 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 (Necula, 2000; Rival, 2004) 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.

Design 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 the following facts. First, while many optimization algorithms involve complex analysis, in the end the transformation are relatively simple syntactic transformation of the code. Second, we have a good idea of what kinds of transformations the optimizer would like to make; namely, only those that improve the code. Using this, we can design a validator which is able to effectively reason about syntactic changes geared toward specific kinds of optimizing transformations.

In this dissertation, we describe a specific implementation of this general design for a denotational translation validator. Our system is relatively simple, and our experimental results show that this approach is both effective and scalable. 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. In our prototype system, the theoretical steps that we presented above are realized as follows:

- First, we translate two LLVM functions to our representation language with explicit effects. This step involves a sophisticated compilation step derived from the general problem of computing path expressions on graphs.

- Second, we use a generalization of symbolic evaluation to compute a symbolic value for the program representations.

- Finally, we check that the symbolic values are equivalent using directed normalization and syntactic equality.

The difficulty lies in the generalization of symbolic evaluation to looping code. 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 work, we develop a method for symbolically representing loops. Our method relies both on our compilation process and our internal representation and its semantics. For our method to work efficiently, we have designed a two-level intermediate language. One level is responsible for describing the meaning of instructions and straight-line code, and the other level captures the control-flow.

We have given our language a semantics which allows us to reason about transformations, and built a compilation algorithm for producing our intermediate form from normal assembly code.

Intuitively, 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 (Dijkstra, 1975).

To validate our design, we have implemented a tool, LLVM M.D. (Low Level Virtual Machine Mis-optimization Detector), that we use with the LLVM compilation framework (LLVM, 2010). We have experimented with our tool using the Spec CPU and the programming language shoot-out benchmark programs (Shootout, 2010; SpecCPU, 2006). This optimizer together with these benchmark programs are very challenging to validate. In our experiments, LLVM M.D. is able to achieve 80-100% validation coverage on an end-to-end optimization pass; a very promising result.

We believe that the capabilities of this design are significant: within a single execution, we 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.

1.2 A brief history of translation validation

Researchers have long used symbolic evaluation in several contexts, such as instance testing, bug finding, and generation of verification conditions. As mentioned by Necula (2000) symbolic evaluation has appeared historically under several disguises, such as predicate transformers (Dijkstra, 1975) or value-dependence graphs (Weise et al., 1994). Our own representation is close to a hash-consed symbolic analysis of a Gated SSA form (Havlak, 1993; Tu and Padua, 1995a).

Starting with Necula (2000) 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 to handle global optimizations. The validator of Necula validates part of GCC 2.7 and the experiments show the results of compiling, and validating, GCC 2.91. Four optimizations were considered: Common sub-expression elimination (CSE), with a rate of false alarms of roughly 5% and roughly 7 minutes running time; loop unrolling with a rate of false alarms of 6.3% and roughly 17 minutes running time; register allocation with a rate of false alarms of 0.1% and around 10 minutes running time; and finally, instruction scheduling with a rate of false alarms of 0.01% and around 9 minutes running time. Unfortunately, the only optimization that we can compare to is CSE as we do not handle loop unrolling, and register allocation is part of the LLVM back-end. In theory, we could handle scheduling (with as good results) but LLVM does not have this optimization. For CSE, our results are comparable, however, we are dealing with a more complex

optimization: global value numbering with partial redundancy elimination and alias information, libc knowledge, and some constant folding.

Rival (2004) also uses a variant of symbolic evaluation, which they call a "transfer function", to design a translation validator. Their system can validate a 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. As with Necula's validator, because of the complex data-flow analysis, the validator of Rival is closely tied to a specific version of an optimizer.

Tristan and Leroy (2008) have used symbolic evaluation to implement formally verified translation validators for list and trace scheduling, optimizations limited to extended blocks. The same two authors (Tristan and Leroy, 2010) 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 thesis is the first where symbolic evaluation is generalized to handle control-flow graphs with 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 et al. (1998) and developed in the TVOC validator (Barrett et al., 2005; Zuck et al., 2003). The most advanced validator following this line of work is the one designed by Kundu et al. (2009) and Tatlock and Lerner (2010). 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 specific optimizations.

There is also the lesser known work on translation validation of Kanade et al. (2006) 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. Kanade et al. (2006) validate GCC 4.1.0 and report no false alarms for CSE, LICM, and copy propagation. To our knowledge, this experiment has the best results. However, it is unclear whether their approach can scale. The authors say that their approach is limited to functions with several hundred RTL instructions and a few hundred transformations. In our setting, functions with more than ten thousand instructions are common.

Finally, Tate et al. (2009) recently proposed a system for translation validation. They compute a value-graph for an input function and its optimized counterpart. They then augment the terms of the graphs by adding equivalent terms through a process known as *equality saturation*, resulting in a data structure similar to the E-graphs of congruence closure. If, after saturation, the two graphs are the same, they can safely conclude that the two programs they represent are equivalent. However, equality saturation was originally designed for other purposes, namely the search for

better optimizations. For translation validation, it is unnecessary to saturate the value-graph, and generally more efficient and scalable to simply normalize the graph by picking an orientation to the equations that agrees with what a typical compiler will do (e.g. $1+1$ is replaced by 2, but not the other way around). Their preliminary experimental evaluation for the Soot optimizer on the JVM shows that the approach is effective and can lead to an acceptable rate of false alarms. On SpecJVM they report an impressive rate of alarms of only 2%. However, the version of the Soot optimizer they validate uses more basic optimizations than LLVM, and does not include, for instance, GVN. Given that our results are mostly directed by GVN with alias analysis, it makes comparisons difficult. It is unclear how well this approach would work for the more challenging optimizations available in LLVM, such as global-value-numbering with alias analysis or sparse-conditional constant propagation. In this work, we show that a *normalizing* value-graph translation has both the simplicity of the *saturation* validator proposed by Tate et al. (2009), and the scalability of Necula's constraint-based approach.

1.3 Outline of this dissertation

The remainder of this dissertation is organized into three parts. After an overview of our design in Chapter 2, we give the syntax and semantics of our intermediate language in Chapters 3-5. Then, we switch gears and describe our compilation algorithms in Chapter 6. Finally, we present our experimental results and conclusions in Chapters 7 and 8. The individual chapters are summarized below.

Chapter 2. In this chapter we describe the our validation system informally. We provide several examples of input programs and their corresponding intermediate forms. We will motivate the main features of our intermediate form through a discussion of side-effects and control-flow. We will also describe the normalization process and rewrite rules on our examples. A large example is described in detail at the end of the chapter. This example shows all of the important features of our system and provides good intuition for the remainder of this dissertation. A formal treatment of the input and intermediate languages and their semantics is given in Chapters 3 and 4.

Chapter 3. In this chapter we describe the input language and its semantics. We take as input, a typical assembly language with an unbounded number of registers. The assembly language is equipped with function definition and call mechanisms which hide the details of calling conventions. We give a semantics to our language by translation to a simply-typed lamba calculus. In the next chapter we will extend our translated assembly language to the final intermediate representation which is able to efficiently represent loops and support symbolic analysis.

Chapter 4. In this chapter we present our intermediate language and its semantics. We formally present Synchronous Value Graphs (SVG), our final intermediate form. This language builds on the monadic assembly language described in the previous chapter. Assuming we are starting with monadic assembly language, then compiling to Synchronous Value Graphs will give us referentially-transparent terms which are amenable to translation validation.

The compilation process will be described in Chapter 6. The algebraic rules are justified by a categorical semantics which we describe in Section 4.1, and mechanically formalize in Chapter **??**.

Chapter 5. In this chapter we present a mechanized version of the semantics described in Chapter **??**. We have formalized our semantics in the theorem prover Coq. The formal definition of the model and the category properties justify the rewrite rules based on those laws. Also, this gives us a framework for formally proving additional rewrite laws.

Chapter 6. In this chapter we will describe the main compilation algorithms we use to compute SVG from the input assembly language. The intermediate representation is computed in two steps: first programs are converted to Gated SSA form. Then, the Gated SSA form is evaluated using our denotational model to produce SVG as described in Chapter 3. The core of the compilation process is the Gated SSA transformation, and much of this chapter is devoted to developing an efficient algorithm for computing Gated SSA for arbitrary input programs.

Chapter 7. In this chapter we present an experimental evaluation of our validation prototype. We test our implementation on the Spec CPU and Programming Language Shootout benchmarks. Our analysis looks at the number of functions our tool is able to validate using different configurations of rules, and at the overall effect on run-time when used as a validated optimizer.

Chapter 8. Conclusions and future directions.

Chapter 2

Overview

In this chapter we describe the our validation system informally. We provide several examples of input programs and their corresponding intermediate forms. We will motivate the main features of our intermediate form through a discussion of side-effects and control-flow. We will also describe the normalization process and rewrite rules on our examples. A large example is described in detail at the end of the chapter. This example shows all of the important features of our system and provides good intuition for the remainder of this dissertation. A formal treatment of the input and intermediate languages and their semantics is given in Chapters 3 and 4.

2.1 Introduction

The key intuition behind our validation strategy is that while many optimization algorithms are extremely complex, the result is almost always a simple syntactic transformation of the input program. As pointed out by Necula (2000), 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.

Denotational translation validation, at a high level is a strategy for performing symbolic evaluation on unstructured and looping low-level code. From this viewpoint, the process of computing a denotation for a program can be seen as a form of predicate-transformer semantics. Following Dijkstra (1975), 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.

In the remainder of this chapter we will informally describe the translation process and denotational form. The translation process has two steps: first we prepare the input code for symbolic evaluation, and then we evaluate the code into a denotational form that is suitable for comparison. In the Chapters 3 and 4 we will formally describe

the syntax of each language, and in Chapter 4 we will give a formal semantics for our denotations.

2.2 Normalizing translation validation

Our validation tool is called *LLVM-MD*, which stands for: Low Level Virtual Machine Mis-optimization Detector. LLVM-MD is an optimizer: it takes as input an LLVM assembly file and outputs an LLVM assembly file. The difference between our tool and the usual LLVM optimizer is that our tool certifies that the semantics of the program is preserved. LLVM-MD has two components, the usual off-the-shelf LLVM optimizer, and a translation validator. The validator takes two inputs: the assembly code of a function before and after it has been transformed by the optimizer. The validator outputs a boolean: *true* if it can prove the assembly codes have the same semantics, and *false* otherwise. Assuming the correctness of our validator, a semantics-preserving LLVM optimizer can be constructed as follows (`opt` is the command-line LLVM optimizer, `validate` is our translation validator):

```
function llvm-md(var input) {
  output = opt -options input
  for each function f in input {
    extract f from input as fi and output as fo
    if (!validate fi fo) {
      replace fo by fi in output
    }
  }
  return output
}
```

For now, our validator works on each function independently, hence we are limited to intra-procedural optimizations. We believe that standard techniques can be used to

validate programs in the presence of function inlining. However, for now we concentrate on the workhorse intra-procedural optimizations of LLVM.

At a high level, our tool works as follows. First, it "compiles" each of the two functions into a value-graph that represents the data dependencies of the functions. Such a value-graph can be thought of as a dataflow program, or as a generalization of the result of symbolic evaluation. Then, each graph is normalized by rewriting using rules that mirror the rewritings that may be applied by the off-the-shelf optimizer. For instance, it will rewrite the 3-node sub-graph representing the expression $2 + 3$ into a single node representing the value 5, as this corresponds to constant folding. Finally, we compare the resulting value-graphs. If they are *syntactically* equivalent, the validator returns *true*. To make comparison efficient, the value-graphs are hash-consed (from now on, we will say "reduced"). In addition, we construct a single graph for both functions to allow sharing between the two (conceptually distinct) graphs. Therefore, in the best case—when semantics has been preserved—the comparison of the two functions has complexity $\mathcal{O}(1)$. The best-case complexity is important because we expect most optimizations to be semantics-preserving.

The LLVM-MD validation process is depicted in Figure 2.1. First, each function is converted to Monadic Gated SSA form (Havlak, 1993; Moggi, 1989; Tu and Padua, 1995a). Monadic Gated SSA form is discussed in detail in Chapter 3. For now, simply note that the goal of this representation is to make the assembly instructions *referentially transparent*: all of the information required to compute the value of an instruction is contained within the instruction itself. More importantly, referential transparency allows us to substitute sub-graphs with equivalent sub-graphs without

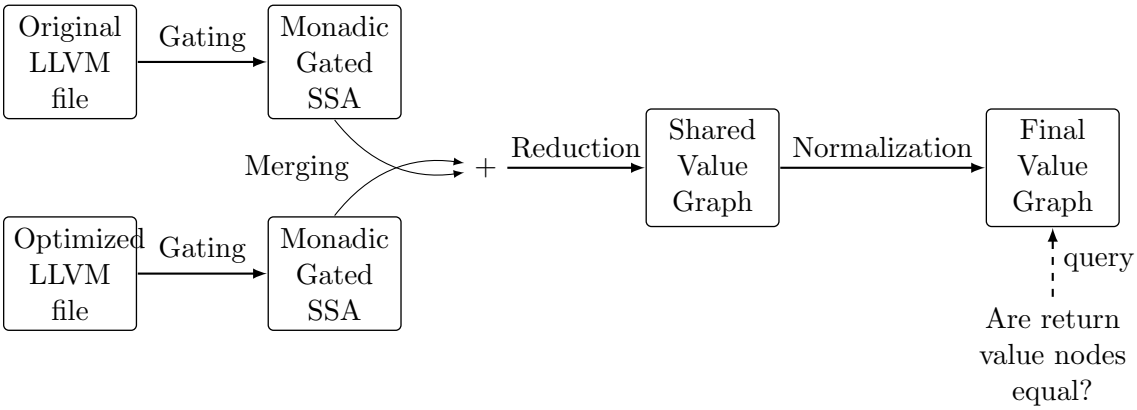


Figure 2.1: LLVM M.D. from a bird's eye view

worrying about computational effects. Computing Monadic Gated SSA form is done in two steps:

1. First, we make side-effects explicit in the syntax by interpreting assembly instructions as monadic commands. For example, a `load` instruction will have a extra parameter representing the memory state.

2. Second, we make the control-flow explicit, by introducing ϕ-nodes with conditions at join points, μ-nodes at loop headers, and η- and σ-nodes with conditions at loop exits. We will see detailed examples of this process in the next section.

Once we have the Monadic Gated SSA form, we compute a shared value-graph by replacing each variable with its definition, being careful to maximize sharing within the graph. Finally, we apply normalization rules and maximize sharing until the value of the two functions merge into a single node, or we cannot perform any more normalization.

It is important to note that the precision of the semantics-preservation property depends on the precision of the monadic form. If the monadic representation does not

model arithmetic overflow or exceptions, then a successful validation does not guarantee anything about those effects. At present, we model memory state, including the local stack frame and the heap. We do not model runtime errors or non-termination, although our approach can be extended to include them. Hence, a successful run of our validator implies that if the input function terminates and does not produce a runtime error, then the output function has the same semantics. Our tool does not yet offer *formal* guarantees for non-terminating or semantically undefined programs.

2.3 Validation by Example

We will now look at several examples of the validation process including computing Monadic Gated SSA form, conversion to value graphs, normalization and comparison.

2.3.1 Basic Blocks

We begin by explaining how the validation process works for basic blocks. Considering basic blocks is interesting because it allows us to focus on the monadic representation and the construction of the value-graph, leaving for later the tricky problem of capturing the control-flow.

As input, we will use a three-address assembly language with an infinite number of registers. For our examples, we will also require the registers be in static single assignment form (SSA), which implies that each register has only one definition. We will relax this requirement in Chapter 6 when we describe our compilation process. We will delay a description of the precise syntax of the input language until Chapter 3, and the syntax of our value graphs until Chapter 4. We hope the programs presented

in this chapter are similar enough to the usual assembly languages that they will be understandable without a formal syntax and semantics.

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 basic blocks this process is very simple. Consider the following basic block, B, where we assume registers a and b are previously defined.

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

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. In this case, 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.

As we can see from this example, for basic blocks we can compute a symbolic value from the concrete assembly values by transcription. More precisely, at the level of basic blocks, the symbolic and concrete languages are identical. The concrete assembly language contains registers like x_1, and the symbolic language contains variables such as x_1. However, the concrete registers and the symbolic variables

have a different interpretations: the registers are machine state, and the variables are simply names for values. The symbolic language also contains analogues $\times$, $+$, *load*, *store*, etc. which are symbolic representations of the corresponding assembly instructions. Therefore, as long as the assembly language represents the entirety of the computation we can convert to symbolic values by reading off the instructions.

Using this intuition, we compute symbolic values for straight-line code 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 assembly language so that the translation from assembly code is as straight-forward as possible[1]. Then, using these rewrite rules, we can compute the symbolic value of any register. For instance, to compute the symbolic value of register x_3, we start with the symbolic value named by x_3, and then rewrite step-by-step using the rewrite rules. When we cannot apply any more rewrite rules we have the final symbolic value for x_3. We represent this process with `symbeval`, which takes two arguments: a symbolic value to rewrite, and a set of rewrite rules. The rewriting just described is written as:

$$\texttt{symbeval}(x_3, \{x_1 \mapsto a \times 3, x_2 \mapsto b \times 3, x_3 \mapsto x_1 + x_2\}) \Rightarrow a \times 3 + b \times 3 \qquad .$$

The result of the symbolic evaluation is a denotation of the register x_3. Functional

[1]The process of generating the rewrite rules is slightly more complex for conditionals and loops, as we will describe in the next sections.

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 show that this simple transformation is a very effective way to build translation validators (Necula, 2000; Rival, 2004; Tristan and Leroy, 2008, 2010). To see how this can be done, consider the basic block below, which is an optimized version of block B.

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

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 with the rewrite rules from block B, and y_2 with the rules from block B'. We then check that the two symbolic values are equivalent. In this case, we must check:

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

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 this is strictly not necessary[2], we believe this sort of "configuration" is practical and much easier than we have seen in other translation validation systems.

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

One problem with the simple validation approach described above, is that it does not scale. When programs become large the symbolic values produced by simple substitution can grow exponentially. Also, when we consider loops, a substitution strategy simply does not work. Therefore, in order to make our approach practical, our symbolic values are represented as graphs, which we call *value graphs*. Using a value-graph representation avoids exponential blow-up, and also gives us constant time comparison for symbolic values. We will now describe the value-graph representation as it is used for validation.

Comparing values using value graphs

Recall, our validator uses an SSA-form assembly language with an infinite number of registers. Because we start with SSA-form, producing the value-graph consists of replacing variables by their definitions while maximizing sharing among graph nodes. For example, consider the following basic block, B1:

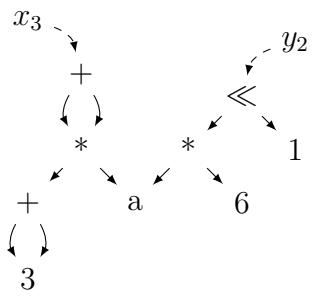
$$\begin{aligned} \text{B1:} \quad & x_1 = 3 + 3 \\ & x_2 = a * x_1 \\ & x_3 = x_2 + x_2 \end{aligned}$$

and its optimized counterpart, B2:

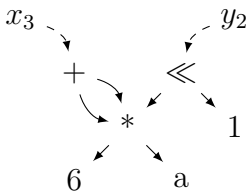
$$\begin{aligned} \text{B2:} \quad & y_1 = a * 6 \\ & y_2 = y_1 \ll 1 \end{aligned}$$

Replacing variables x_1, x_2, and y_1 by their definition, we obtain the value-graph presented below. The dashed arrows are not part of the value graph, they are only

meant to point out which parts of the graph correspond to which program variables. Note that both blocks have been represented within one value graph, and the node for the variable `a` has been shared.



Suppose that we want to show that the variables x_3 and y_2 will hold the same value. Once we have the shared value graph in hand, we simply need to check if x_3 and y_2 are represented by subgraphs rooted at the same graph node. In the value graph above, x_3 and y_2 are not represented by the same subgraph, so we cannot conclude they are equivalent. However, we can now apply normalization rules to the graph. First, we can apply a constant folding rule to reduce the subgraph $3 + 3$ to a single node 6. The resulting graph is shown below where we have maximized sharing in the graph.



We have managed to make the value graph smaller, but we still cannot conclude that the two variables are equivalent. So, we continue to normalize the graph. A second rewrite rule allows us to replace $x+x$ with $x \ll 1$ for any x. In our setting, this rule is only appropriate if the optimizer would prefer the shift instruction to addition. After

replacing addition with left shift, and maximizing sharing, x_3 and y_2 point to the same node and we can conclude that the two blocks are equivalent. This process of producing value-graphs, normalization and comparison is the essence of our validator. The main contribution of this work is showing how to extend this simple model to real-world code with side-effect and loops.

Side Effects

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

$$p_1 = \texttt{alloc}\ 1$$

$$p_2 = \texttt{alloc}\ 1$$

$$\texttt{store}\ x, p_1$$

$$\texttt{store}\ y, p_2$$

$$z = \texttt{load}\ p_1$$

If we naively apply our translation, then the graph corresponding to z would be: $z \mapsto \texttt{load}\ (\texttt{alloc}\ 1)$, 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 sequence of instructions for this block:

$$p_1, m_1 = \texttt{alloc}\ 1, m_0$$

$$p_2, m_2 = \texttt{alloc}\ 1, m_1$$

$$m_3 = \texttt{store}\ x, p_1, m_2$$

$$m_4 = \texttt{store}\ y, p_2, m_3$$

$$z, m_5 = \texttt{load}\ p_1, m_4$$

Here, the current memory state is represented by the m-registers. Each instruction requires and produces a memory register in addition to its usual parameters. This extra register enforces a dependency between, for instance, the `load` instruction and the preceding `store` instructions. 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 (Moggi, 1989). Using these "monadic" instructions, we can apply our transformation and produce a value graph that captures all of the relevant information for each register.

The rewrite rules in our system are able to take into account aliasing information to relax the strict ordering of instructions imposed by the monadic transformation. In our experimental setting (LLVM), we know that pointers returned by `alloc` never alias with each other. Using this information, we are able to replace m_4 with m_3 in the `load` instruction for z. Then, because we have a `load` from a memory consisting of a `store` to the same pointer, we can simplify the `load` to x.

Using the state variables, we can validate that a function not only computes the same value as another function, but also affects the heap in the same way. In

Chapter **??** we show how the same technique can be applied to different kinds of side effects, such as arithmetic overflow, division by zero, and non-termination. For our experiments, we have only modeled memory side-effects in our implementation. Hence, experimentally 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, though doing so may make it harder to validate optimizations.

In Chapter 4 we will present a semantic model for our language. There we will see the entire semantics is parametrized over the monad used in the translation. Therefore, all of the infrastructure and formal development applies equally for any set of (monadic) effects we wish to model. However, our monadic translation is not able to encode control-flow within the semantic values. This is done through a compilation step which we describe in the next sections.

2.3.2 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.

$$
\begin{array}{lll}
\textbf{entry}: & c = a < b & \\
 & \texttt{cbr}\ c, \textbf{true}, \textbf{false} & \\
\textbf{true}: & x_1 = x_0 + x_0 & \text{(True branch)} \\
 & \texttt{br}\ \textbf{join} & \\
\textbf{false}: & x_2 = x_0 * x_0 & \text{(False branch)} \\
 & \texttt{br}\ \textbf{join} & \\
\textbf{join}: & x_3 = \phi(x_1, x_2) & \text{(Join point)}
\end{array}
$$

Notice, in the block labeled **join**, the value of x_3 is represented by a ϕ-node. This ϕ-node means that x_3 can have either the value x_1 or x_2 depending on the control flow. If we naively apply our technique, we have as a symbolic value for x_3:

$$\texttt{symbeval}(x_3, \{\ldots\}) = \phi(x_0 + x_0, x_0 * x_0) \quad .$$

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. For instance, if we reverse the condition so that it is $b \leq a$, 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 split the control-flow.

Adding a condition (or gate) to the ϕ-node makes the ϕ-node referentially transparent with respect to the control flow. Using the *gated* ϕ-node we can distinguish the two programs. In our example, the last instruction would become $x_3 = \phi(c, x_1, x_2)$, which means x_3 is x_1 if c is *true*, and x_2 otherwise.

In order to handle real C programs, the actual syntax of ϕ-nodes has to be a bit more complex. In general, a ϕ-node is composed of a set of possible branches, one for each control-flow edge that enters the ϕ-node. Each branch has a set of conditions, all of which must be true for the branch to be taken.[3]

$$\phi \left\{ \begin{array}{c} c_{11} \ldots c_{1n} \rightarrow v_1 \\ \ldots \\ c_{k1} \ldots c_{km} \rightarrow v_k \end{array} \right\}$$

Given this more general syntax, the notation $\phi(c, x, y)$ is simply shorthand for $\phi(c \rightarrow x, !c \rightarrow y)$.

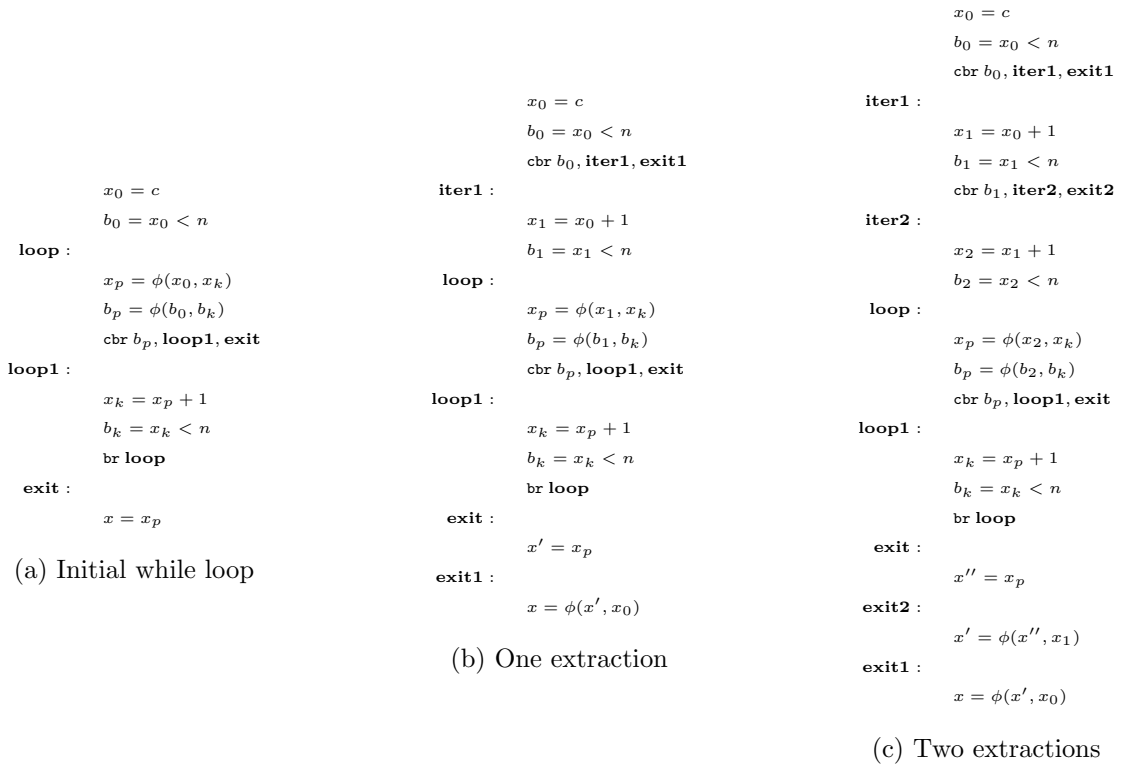
Gated ϕ-nodes come along with a set of normalization rules that we present in the next section. When generating gated ϕ-nodes, it is important that the conditions for each branch are mutually exclusive with the other branches. This way, we are free to apply normalization rules to ϕ-nodes (such as reordering conditions and branches) without worrying about changing the semantics.

It is also worth noting that if the values coming from various paths are equivalent, then they will be shared in the value graph. This makes it possible to validate optimizations based on a global-value numbering that is aware of equivalences between definitions from distinct paths. We will see an example of this in Section 2.4.

[3] ϕ-nodes with several branches and many conditions are very common in C programs. For example, an if-statement with a condition that uses short-cut boolean operators, can produce complex ϕ-nodes.

2.3.3 Loops

In order to generalize our technique to loops, we must come up with a way to place gates within looping control flow (including breaks, continues, and returns from within a loop). Also, we need a way to represent values constructed within loops in a referentially transparent way. We achieve this by introducing three new constructs into our language which we will now describe informally.



(a) Initial while loop

(b) One extraction

(c) Two extractions

Figure 2.2: Extraction of loop iterations

Consider the register x whose value depends on the execution of the while loop presented in figure 2.3a. 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 2.3b, then the symbolic value of x becomes $\phi(b_0, x_p, x_0)$. 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 2.3c, to refine the approximation of x. With two iterations extracted, the symbolic value of x is $\phi(b_0, \phi(b_1, x_p, x_1), x_0)$. 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:

$$\phi(b_0, \phi(..., \phi(b_{n-1}, \phi(b_n, x_p, x_n), x_{n-1}), ...), x_0) \qquad .$$

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 the recurrence relation,

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

and the register b as a sequence defined by,

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

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

$$\phi(c < n, \phi(c + 1 < n, ..., c + 1), c) \qquad .$$

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 to represent values with this shape. In this example, we write:

$$x = \eta(\mu(b_o, b_n), \mu(x_o, x_n)) \qquad .$$

This term indicates that x is a variable modified within a loop with condition b. The condition b varies within the loop according to the recurrence relation defined by b_0, b_n, and x varies within the loop according to the recurrence relation defined by x_0, x_n.

The η- and μ-nodes (as well as the gated ϕ-nodes) are part of Gated SSA form. We use Gated SSA to represent loops and conditional control flow in a referentially transparent way. For loops, μ, is used to define variables that are modified within a loop. Each μ is placed at a loop header and holds the initial value of the variable on entry to the loop and a value for successive iterations. The μ-node is equivalent to a non-gated ϕ node from classical SSA. The η-node is used to refer to loop-defined variables from outside their defining loops. The η-node carries the variable being referred to along with the condition required to reach the η from the variable definition.

To see these new constructs in action, consider Figure 2.3a which shows a simple while loop in SSA form. The Gated SSA form of the same loop is shown in figure 2.3b. The ϕ-nodes in the loop header have been replaced with μ-nodes, and the access to the x_p register from outside to loop is transformed into an η-node that carries the condition required to exit the loop and arrive at this definition.

With Gated SSA form, recursively defined variables must contain a μ-node. The recursion can be thought of as a cycle in the value graph, and all cycles are dominated

$x_0 = c$

loop :

$x_p = \phi(x_0, x_k)$

$b = x_p < n$

`cbr` b, **loop1**, **exit**

loop1 :

$x_k = x_p + 1$

`br` **loop**

exit :

$x = x_p$

(a) SSA while loop

$x_0 = c$

loop :

$x_p = \mu(x_0, x_k)$

$b = x_p < n$

`cbr` b, **loop1**, **exit**

loop1 :

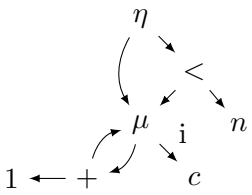
$x_k = x_p + 1$

`br` **loop**

exit :

$x = \eta(b, x_p)$

(b) GSSA while loop



(c) Value graph

Figure 2.3: Representation of while loops

by a μ-node. The value graph corresponding to our previous example is presented in figure 2.3c. Intuitively, the cycle in the value-graph can be thought of as generating a stream of values. The μ-nodes start by selecting the initial value from the arrow marked with an "i". Successive values are generated from the cyclic graph structure attached to the other arrow. This μ-node "produces" the values c, $c + 1$, $c + 2$,... The η receives a stream of values and a stream of conditions. When the stream of conditions goes from *true* to *false*, the η selects the corresponding value in the value stream.

Generally, we can think of μ and η behaving according to the following formulas:

$$\mu(a, n) = a : \mu(n[a/x], n)$$

$$\eta(0 : \overline{b}, x : \overline{v}) = \eta(\overline{b}, \overline{v})$$

$$\eta(1 : \overline{b}, x : \overline{v}) = x$$

Of course, for our purposes, we do not need to evaluate these formulas, we simply need an adequate, symbolic representation for the registers x, x_p and b_p. Our formal semantics, which is presented in Chapter **??**, borrows ideas from data-flow programming languages to provide an interpretation for these constructs which is amenable to comparison.

2.4 Normalization

Once a graph is constructed for two functions, if the functions' denotations are not already equivalent, we begin to normalize the graph. We normalize value graphs using a set of rewrite rules. We apply the rules to each graph node individually. When no more rules can be applied, we maximize sharing within the graph and then reapply our rules. When no more sharing or rules can be applied, the process terminates.

Our rewrite rules come in two basic types: general simplification rules and optimization-specific rules. The general simplification rules reduce the number of graph nodes by removing unnecessary structure. We say *general* because these rules only depend on the graph representation, replacing graph structures with smaller, simpler graph structures. The optimization-specific rules rewrite graphs in a way that mirrors the effects of specific optimizations. These rules do not always make the graph smaller

or simpler, and one often needs to have specific optimizations in mind when adding them to the system.

General Simplification Rules. The notation $a \downarrow b$ means that we match graphs with structure a and replace them with b. The first four general rules simplify boolean expressions:

$$a = a \downarrow \texttt{true} \tag{2.1}$$

$$a \neq a \downarrow \texttt{false} \tag{2.2}$$

$$a = \texttt{true} \downarrow a \tag{2.3}$$

$$a \neq \texttt{false} \downarrow a \tag{2.4}$$

These last two rules only apply if the comparison is performed at the boolean type. In fact, all LLVM operations, and hence our graph nodes, are typed. However, we rarely need to consult the types other than to determine syntactic equality. We will omit typing information unless it is instructive.

There are two general rules for removing unnecessary ϕ-nodes.

$$\phi\{\ldots, \overline{\texttt{true}_\texttt{i}} \to t, \ldots\} \downarrow t \tag{2.5}$$

$$\phi\left\{\overline{c_i \to t}\right\} \downarrow t \tag{2.6}$$

The first rule replaces a ϕ-node with one of its branches if all of its conditions are satisfied for that branch. We write $\overline{x_i}$ for a set of terms indexed by i. In the first rule, we have a set of true values. Note that the conditions for each branch are mutually exclusive with the other branches, so only one branch can have conditions which are all true. The second rule removes the ϕ-node if all of the branches contain the same

value. A special case of this rule is a ϕ-node with only one branch indicating that there is only one possible path to the ϕ-node, as happens with branch elimination.

The ϕ rules 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; d = a;}
  else {a = 2; b = 2; d = 1;}
if (a == b) {x = d;} else {x = 0;}
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 if-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. Hence, the above program and `return 1` have the same normalized value graph, computed as follows:

$$
\begin{aligned}
x \mapsto & \phi(\phi(c,1,2) == \phi(c,1,2), \phi(c,1,1), 0) \\
& \downarrow \phi(\texttt{true}, \phi(c,1,1), 0) && \text{by (2.1)} \\
& \downarrow \phi(c,1,1) && \text{by (2.5)} \\
& \downarrow 1 && \text{by (2.6)}
\end{aligned}
$$

We also have general rules for simplifying η- and μ-nodes. The first rule allows us to remove loops that never execute.

$$\eta(\texttt{false}, \mu(x,y)) \downarrow x \tag{2.7}$$

This rule rewrites to the initial value of the μ-node before the loop is entered, namely x. This rule is needed to validate loop-deletion, a form of dead code elimination. In addition, there are two rules for loop invariants. The first says that if we have a constant μ-node, then the corresponding η-node can be removed:

$$\eta(c, \mu(x, x)) \downarrow x \tag{2.8}$$

$$\eta(c, y \mapsto \mu(x, y)) \downarrow x \tag{2.9}$$

In rule (2.8), the μ-node has an initial value of x, which must be defined outside of the loop, and therefore cannot vary within the loop. Since, x does not vary within the loop the μ-node does not vary, and the loop structure can be removed. Rule (2.9), expresses the same condition, but the second term in the μ-node is again the same μ-node (we use the notation $y \mapsto \mu(x, y)$ to represent this self-reference).

These rules are 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 transforms the program to `return (a + 3)`.

The value graph for x is computed as follows:

$$
\begin{aligned}
& i_n \mapsto \mu(0, i_n + 1) \\
& x \mapsto \eta(i_n < n, \mu(a + 3, a + 3)) \\
& \quad \downarrow a + c \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by 2.8)}
\end{aligned}
$$

Note that the global copy propagation is taken care of "automatically" by our representation, and we can apply our rule (2.8) immediately. The other nodes of the graph (i_n) are eliminated since they are no longer needed.

Optimization-specific Rules. In addition to the general rules, we also have a number of rewrite rules that are derived from the semantics of LLVM, our specific optimization framework. For example, we have a family of laws for simplifying constant expressions, such as:

$$\texttt{add}\ 3\ 2 \downarrow 5$$

$$\texttt{mul}\ 3\ 2 \downarrow 6$$

$$\texttt{sub}\ 3\ 2 \downarrow 1$$

Currently we have rules for simplifying constant expressions over integers, but not floating point or vector types. There are also rules for rewriting instructions such as:

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

$$\texttt{mul}\ a\ 4 \downarrow \texttt{shl}\ a\ 2$$

These last two rules are included in our validator because we know that LLVM's optimizer prefers the shift left instruction. While preferring shift left may be obvious,

there are some less obvious rules such as:

$$\texttt{add}\ x\ (-k) \downarrow \texttt{sub}\ x\ k$$

$$\texttt{gt}\ 10\ a \downarrow \texttt{lt}\ a\ 10$$

$$\texttt{lt}\ a\ b \downarrow \texttt{le}\ a\ (\texttt{sub}\ b\ 1)$$

While these transformations may not be optimizations, in the strictest sense, LLVM performs these transformations to give the instructions a more regular structure.

Finally, we also have rules that make use of aliasing information to simplify memory accesses. For example, we have the following two rules for simplifying loads from memory:

$$\texttt{load}(p, \texttt{store}(x, q, m)) \downarrow \texttt{load}(p, m) \tag{2.10}$$

$$\texttt{load}(p, \texttt{store}(x, p, m)) \downarrow x \tag{2.11}$$

when p and q do not alias. Our validator can use the result of a may-alias analysis. For our experiments, we only use simple non-aliasing rules, such as: two pointers that originate from two distinct stack allocations may not alias; two pointers forged using `getelemptr` with different parameters may not alias, etc.

2.4.1 Efficiency

At this point it is natural to wonder why we did not simply define a normal form for expressions and rewrite our graphs to this normal form. This is a good option, and we have experimented with this strategy using external SMT provers to find equivalences between expressions. However, one of our goals is to build a practical tool which optimizes the best case performance of the validator (we expect most

optimizations to be correct). Using our strategy of performing rewrites motivated by the optimizer, we are often able to validate functions with tens of thousands of instructions (resulting in value graphs with hundreds of thousands of nodes) with only a few dozen rewritings. That is, we strive to make the amount of work done by the validator proportional to the number of transformations performed by the optimizer.

To this end, the rewrite rules derived from LLVM semantics are designed to mirror the kinds of rewritings that are done by the LLVM optimization pipeline. In practice, it is *much* more efficient to transform the value graphs in the same way the optimizer transforms the assembly code: 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 cannot recognize 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.

2.4.2 Extended Example

We now present a larger example where all these laws interplay to produce the normalized value-graph. Consider the C code below:

```
int f(int n, int m) {
  int * t = NULL;
  int * t1 = alloca(sizeof(int));
  int * t2 = alloca(sizeof(int));
  int x, y, z = 0;
  *t1 = 1; *t2 = m;
  t = t1;
  for (int i = 0; i < n; ++i) {
    if (i % 3) {
      x = 1; z = x << y; y = x;
    } else {
      x = 2; y = 2;
    }
    if (x == y)  t = t1;
    else t = t2;
  }
  *t = 42;
  return *t2 + *t2;
}
```

First, note that this function returns $m + m$. Indeed, x is always equal to y after the execution of the first conditional statement in the for-loop. Therefore, the second conditional statement always executes the left branch and assigns t_1 to t. This is actually a loop invariant, and, since t_1 is assigned to t before the loop, t_1 is always equal to t. t_1 and t_2 are pointers to two distinct regions of memory and cannot alias. Writing through t_1 does not affect what is pointed to by t_2, namely m. The function

therefore returns $m + m$. Since the loop terminates, an optimizer may replace the body of this function with $m \ll 1$, using a blend of global-value numbering with alias analysis, sparse-conditional constant propagation and loop deletion.

Our value-graph construction and normalization produces the value-graph corresponding to $m \ll 1$ for this example. The initial value-graph is presented in Figure 2.4a. Some details of the graph have been elided for clarity. We represent `load`, `store`, and `alloca` nodes with `ld`, `st`, and `al` respectively. To make the graph easier to read, we also used dashed lines for edges that go to pointer values.

A potential normalization scenario in shown in Figure 2.4 and continued in Figure 2.5. The reduction shown in these figures corresponds to the following rewrite rules being applied in order.

1. The arguments of the == node are shared; it is rewritten to *true*.

$$x = x \downarrow true$$

2. The gate of the ϕ node is *true*; its predecessor, μ, is modified to point to the target of the true branch of the ϕ node rather than to ϕ itself.

$$\phi(true, a, b) \downarrow a$$

3. The arguments of this μ node are shared; its predecessor, η, is modified to point to the target of μ instead of μ itself.

$$\mu(x, x) \downarrow x$$

4. The condition of the η node is terminating, and its value acyclic; Its predecessor,

`st` 42, is modified to point to the target of η instead of η itself.

$$\eta(a, b) \downarrow b \qquad \text{b constant}$$

5. It is now obvious that the load and its following store use distinct memory regions; the load can "jump over the store."

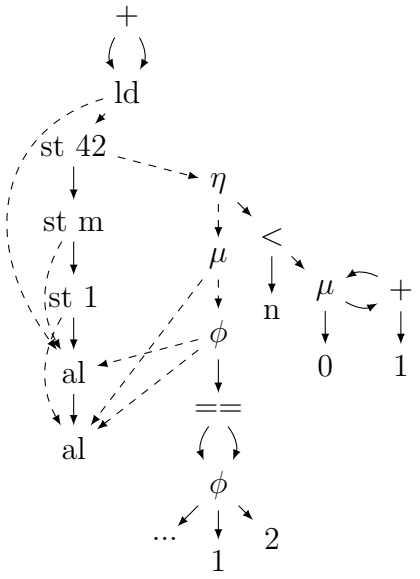
$$\texttt{load}(p, \texttt{store}(x, q, m)) \downarrow \texttt{load}(p, m)$$

6. The load and its new store use the same memory region; the load is therefore replaced by the stored value, m.

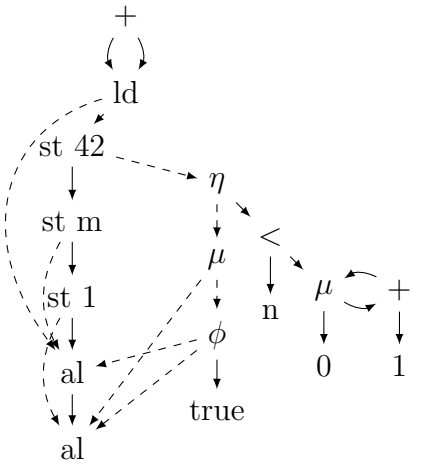
$$\texttt{load}(p, \texttt{store}(x, p, m)) \downarrow x$$

7. The arguments of the + node are shared; The + node is rewritten into a left shift.

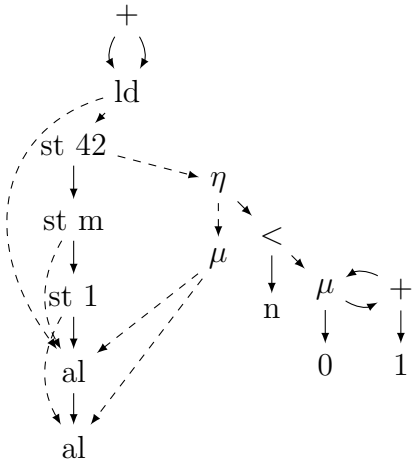
$$\texttt{add}\ a\ a \downarrow \texttt{shl}\ a\ 1$$



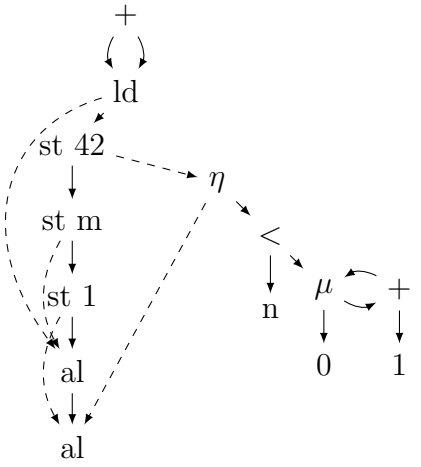
(a) Initial shared value-graph



(b) The arguments of the == node are shared; it is rewritten to *true*.

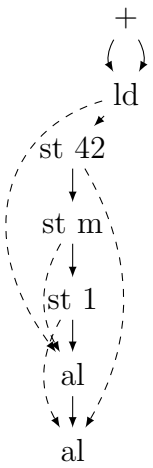


(c) The gate of the ϕ node is *true*; its predecessor, μ, is modified to point to the target of the true branch of the ϕ node rather than to ϕ itself.

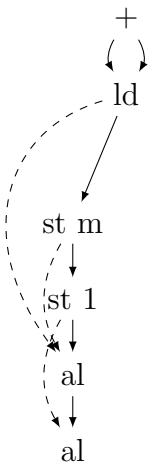


(d) The arguments of this μ node are shared; its predecessor, η, is modified to point to the target of μ instead of μ itself.

Figure 2.4: Normalization of Large Example.

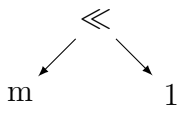


(a) The condition of the η node is terminating, and its value acyclic; Its predecessor, `st` 42, is modified to point to the target of η instead of η itself.



(b) It is now obvious that the load and its following store use distinct memory regions; the load can "jump over the store."

(c) The load and its new store use the same memory region; the load is therefore replaced by the stored value, m.



(d) The arguments of the + node are shared; The + node is rewritten into a left shift.

Figure 2.5: Normalization of Large Example (continued).

Chapter 3

Assembly Language

As described in the overview, our validator compares unoptimized and optimized programs. This is done by converting the programs into value graphs, performing rewritings and comparing the results. In this chapter we describe the input language and its semantics. We take as input, a typical assembly language with an unbounded number of registers. The assembly language is equipped with function definition and call mechanisms which hide the details of calling conventions. We give a semantics to our language by translation to a simply-typed lamba calculus. In the next chapter we will extend our translated assembly language to the final intermediate representation which is able to efficiently represent loops and support symbolic analysis.

3.1 Assembly Language Syntax

As a starting point, we assume a typical assembly language composed of functions containing blocks of instructions terminated by control-flow operations. Our assembly language is designed to be similar to LLVM in that we use the same basic assembly instructions. However, we do not require the input code be in SSA form. The syntax of assembly language functions is show in Figure 3.1. Each function has a return type,

function	::=	t name($\overline{t\ x}$) {$\overline{\texttt{block}}$}	Function
block	::=	ℓ : $\overline{x = \texttt{statement}}$; `control`	Block
statement	::=	$t\ v$	Value
	|	i	Instruction
control	::=	**return** [v]	Return from function
	|	**br** ℓ	Branch to label
	|	**cbr** $v\ \ell\ \ell$	Conditional Branch

Figure 3.1: Syntax of Assembly Language Functions

a name, a number of formal parameters with their types (written as $\overline{t\ x}$), and a list of blocks. The first block in the list is the entry block, and is where execution starts. Each block is composed of a label ℓ, a list of assignments to variables, and a final control-transfer instruction. Each block only has one control-transfer instruction, and it is always the last instruction in the block. The control-transfer instructions include `return` which optionally takes a value v to return, branch (`br`) and conditional branch (`cbr`). The branch instruction, `br` ℓ, transfers control to the block with label ℓ. The conditional-branch instruction, `cbr c` ℓ_1 ℓ_2, transfers control to ℓ_1 if condition c is true, and ℓ_2 otherwise.

Within a block, each variable is assigned a value computed by a statement. Statements can be values or assembly instructions applied to values (the syntax of values and instructions is described below). We require all variables referenced by instructions within a block to be defined on all control-flow paths leading to that block. In addition, each variable must be assigned a single type. The syntax of types is show in Figure 3.2. The base types include integers of different widths, void and label types.

$$
\begin{array}{rcll}
t & ::= & \texttt{int}_n \mid \text{void} \mid \text{label} & \text{Base Types} \\
& \mid & *t & \text{Pointer Type} \\
& \mid & t[n] & \text{Array Type} \\
& \mid & \bar{t} \to t & \text{Instruction Type}
\end{array}
$$

Figure 3.2: Assembly Language Type System

On a 32-bit architecture, machine words are represented with type $\texttt{int}_{32}$. Boolean values are given type $\texttt{int}_1$ with the value 1 representing `true` and 0 representing `false`. The `void` type is used for functions that do not return values, and `label` is the type of static addresses. Types also include pointers $*t$, arrays $t[n]$, and instruction types. The instruction type $a\ b \to c$ represents an instruction that takes two parameters of type a and b and returns a result of type c. Instructions can only be called with all of their arguments (we do not allow partial application of instructions in the assembly language).

Our assembly language contains a set of primitive values which consist of fixed-width integers and variables. For convenience we also define a few derived constants. The constant `true` is equivalent to 1, and `false` and `null` are both equivalent to 0. The constant `false` has type $\texttt{int}_1$ and `null` can have any pointer or array type. We

v	::=	1,2,...	Fixed Width Integers
	|	true | false | null	Derived Constants
	|	x, y, z, x1 ...	Variables

Figure 3.3: Assembly Lanugauge Values

could also allow values to include expressions over values (as does LLVM), but this does not add any additional expressiveness, and it is not necessary for our purposes.

Finally, Figure 3.4 gives the instructions of our assembly language which are used to construct statements. Each instruction takes a number of arguments, all of which must be values. Instructions include pointer arithmetic with `getelemptr`: if x has

i	::=	**getelemptr** $t\ v\ \overline{v}$	Pointer Arithmetic
	|	**alloc** $t\ v$	Stack Allocation
	|	**load** $t\ v$	Load from Memory
	|	**store** $t\ v\ v$	Store to Memory
	|	**binop** $t\ v\ v$	Binary Operator
	|	**conv** $t\ v\ t$	Type Conversion
	|	**select** $t\ v\ v\ v$	Select on Condition
	|	**call** name($\overline{v}$)	Function Call

Figure 3.4: Assembly Language Instructions

type $*\texttt{int}_{32}[10]$, then the instruction (`getelemptr` $\texttt{int}_{32}$ x 0 1) points to the second element of the first array pointed to by x. We also have the memory operations: `alloc`, `load`, and `store`. The `alloc` instruction allocates some number of elements of type t on the stack. The `load` instruction reads a value from a location in memory, and the `store` instruction stores a value to a location in memory. An example of

these instructions is shown below.

```
int32 memory() {
  p = alloc int32 1    ; allocate 1 word on stack
  store int32 p 7      ; store the value 7 a location p
  seven = load int32 p ; seven = 7
  return seven
}
```

The function `memory` allocates a single 32-bit word, stores a value in memory, and then reads the value from memory and returns the result. In this case, the function returns the value 7.

Instructions also include binary operators, type conversions, and an instruction-level conditional called `select`. The exact set of binary operators and type conversions does not need to be fixed ahead of time; new operations can be easily added without affecting the system. For our purposes we will consider a typical complement of integer operations; these operations are shown in Figure 3.5. We have inte-

conv	::=	trunc | zext | sext	Width Conversions
	|	ptr2int | int2ptr | bitcast	Type Conversions
binop	::=	add | sub | mul	Arithmetic
	|	udiv | sdiv | urem | srem	(Un)signed division
	|	shl | shrl | shra	Logical/Arithemic shift
	|	and | or | xor	Bitwise operators
	|	eq | ne	Equality
	|	sgt | sge | slt | sle	Signed Comparison
	|	ugt | uge | ult | ule	Unsigned Comparison

Figure 3.5: Assembly Language Operations

ger widening and truncation, pointer-integer conversion, and `bitcast` for converting

between pointer types. The binary operators include basic arithmetic and logical operations for signed and unsigned integers. We have not included floating point operations in the formal syntax, however they are included in our implementation. Our implementation also handles structure types and a richer language of pointer arithmetic which we do not detail here.

An example of the assembly language is shown in Figure 3.6. The left-hand side of the figure shows a function written in C that defines a simple loop. The right-hand side shows the corresponding assembly language. The assembly language code

```
int loop() {
 int t = 1;
 int i;
 for (i = 0; i < 10; i++) {
   t += i;
 }
 return t;
}
```

C code with simple loop

```
int32 loop() {
entry:
 t = int32 1
 i = int32 0
 br header
header:
 c = slt int32 i, 10
 cbr c, body, end
body:
 t = add int32 t, i
 i = add int32 i, 1
 br header
end:
 ret int32 t
}
```

Corresponding Assembly Code

Figure 3.6: Loop Example

contains four blocks. The `header` and `body` blocks form a loop. The function begins execution at the `entry` block, which branches to the loop header. When we first enter the `header` block the value of i is 0, so c is `true` and we branch to `body`. The `body`

block updates the t and i variables and continues back to the `entry` block. After several iterations, c becomes `false` and we branch to `end` which returns the value t.

3.1.1 Working with Assembly Language

In this section we describe a number of concepts for working with assembly language blocks, labels, and variables. Each assembly language function is made up of a number of labeled basic blocks. Each basic block within a function must have a unique label—no two blocks within a function can have the same label. We use the notation $labels(f)$ to refer to the set of labels used within a function. Because the labels used in a function are unique, we can define a total function from $labels(f)$ to the basic blocks of f. We call this finite mapping a *block map* and it is defined as:

Definition 1 (Block Map)**.** *Given the assembly function*

$$tf(\overline{t_i\ x_i})\{\overline{b_i}\} \qquad ,$$

then for each $\ell \in labels(f)$, the **block map** *for f is defined as:*

$$BM_f(\ell) = b \mid b \in \{\overline{b_i}\} \wedge b = \ell : \overline{x_i = s_i}; c$$

for some x_i, s_i and c.

Note that the domain of BM is equal to $labels(f)$, and the codomain is equal to $\{\overline{b_i}\}$.

The control-transfer instructions from the basic blocks form a graph of blocks called the *control-flow graph* where each block's successors are the possible destinations of its control-flow instruction. We can capture this graph precisly by defining the *free labels* occuring in a block.

Definition 2 (Free Labels)**.** *The* **free labels** *of a block, are the set of labels that appear in a block's control transfer instruction. The free labels are defined by:*

$$FL(\ell : \overline{x_i = s_i}; c) = fl(c)$$

$$fl(\textbf{return}) = \emptyset$$

$$fl(\textbf{return}\ v) = \emptyset$$

$$fl(\textbf{br}\ \ell) = \{\ell\}$$

$$fl(\textbf{cbr}\ v\ \ell_1\ \ell_2) = \{\ell_1, \ell_2\}$$

The FL function takes a block as an argument, however we will also write:$FL(\ell)$ to mean $FL(BM(\ell))$, where ℓ is a label. With the definition of free labels in hand, we can define the control-flow graph.

Definition 3 (Control-Flow Graph)**.** *The* **control-flow graph** *for a function*

$$tf(\overline{t_i\ x_i})\{\overline{b_i}\} \qquad ,$$

is a set of blocks B and a set edges E defined as:

$$B = \{\overline{b_i}\}$$

$$E = \{(\ell_1, \ell_2) \mid \ell_1 \in labels(f) \wedge \ell_2 \in FL(\ell_1)\}$$

We also define the sets of *free variables* of a block. These sets include all of the variables that are referred to by the statements within the block. More formally,

Definition 4 (Free Variables)**.** *For a block, $b = \ell : \overline{x_i = s_i}; c$, the* **free variables** *of b are defined as:*

$$FV(b) = \bigcup_{s \in s_i} FV(s)\ \cup FV(c) \qquad .$$

Where $FV(s)$ and $FV(c)$ are the free variables of statements and control-transfer instructions respectively. The free variables of statements and control-transfer instructions are the set of variables which appear syntactically in the statement or control-transfer instruction. We elide the definition of free variables for statements and control-transfer instructions as they are both obvious and verbose. As with free labels, we will also write, $FV(\ell)$, where ℓ is a label, with the following meaning:

$$FV(\ell) = FV(BM(\ell)) \quad .$$

Finally, we sometimes need to refer to the set of variables referred to by a block and any of its successor blocks. We write this as FV^*, and it is defined below.

Definition 5 (Transitive Free Variables)**.** *For a function f, and a label $\ell \in labels(f)$, the* **transitive free variables** *of a block labeled by ℓ is the least fixed point of the equation:*

$$FV^*(\ell) = FV(\ell) \cup \bigcup_{\ell' \in FL(\ell)} FV^*(\ell') \quad .$$

The assembly language we have described up to this point forms the inputs to our validation process. In the next section we will describe the simply-typed language we use to give a semantics to the assembly language.

3.2 Target Language

The first step in our validation process is a compilation step. The compilation step converts the input assembly language into our intermediate representation. The compilation step happens in two phases. The first phase converts each basic block into a simply-typed lambda term. This term will be a function which will take as

arguments the free variables of the basic block, and an abstract machine state, and it will return a new machine state and a value. The functions derived from the basic blocks are specified using the simply-typed lambda calculus described in this section. These functions have two purposes: first, they give a meaning to the basic blocks; second, they form the inputs to the second phase of the compilation process. The remainder of this chapter is concerned with the first phase of the translation including the simply-typed lambda calculus, and the translation of the basic blocks to lambda terms. In the next chapter will deal with the full intermediate language and the second phase of the translation.

3.2.1 Target Language Syntax

Basic blocks are translated into a variant of the simply-typed lambda calculus with sums and products (Barendregt, 1984). The syntax is shown in Figure 3.7. The types extend the assembly language types by adding a function type, a product type, and a sum type. Expressions e, include assembly values and fully applied assembly instructions. We also have a unit value and conditional expressions. Abstractions are explicitly typed. However, note that since we have a simply typed lambda calculus, these types can be reconstructed from the type annotations on values and instructions (Giannini et al., 1993). We also have the usual pair constructor and projections `fst` and `snd`. Finally, sums are constructed with `inl` and `inr`, and case analysis on sums is performed with `case`.

Dynamic Semantics. The dynamic semantics of our lambda calculus is standard call-by-value. The set of values is the subset of well-typed expressions that cannot be

τ	::=	t	Assembly Types
	|	`unit`	Unit type
	|	$\tau \to \tau$	Functions
	|	$\tau * \tau$	Products
	|	$\tau + \tau$	Sums
e	::=	v	Assembly Value
	|	$i\ \overline{e}$	Applied Assembly Instruction
	|	`unit`	Unit Value
	|	`if` e `then` e `else` e	Conditionals
	|	$\lambda x{:}\tau.e$	Abstraction
	|	$e\ e$	Application
	|	(e, e)	Product Constructor (pairs)
	|	`fst`(e) | `snd`(e)	Projections
	|	`inl`(e) | `inr`(e)	Sum Constructors
	|	`case` e `of` $\{$`inl`$(x).e$ | `inr`$(x).e\}$	Case Analysis

Figure 3.7: Syntax of the Extended Assembly Language

reduced according to the dynamic semantics. For our language, the values are:

$$val ::= v \mid i\ \overline{val} \mid \texttt{unit} \mid \lambda x{:}\tau.e \mid (val, val) \mid \texttt{inl}(val) \mid \texttt{inr}(val)$$

The primary reduction rules for our language are:

$$(\lambda x{:}t.e)\ val \to e[x \mapsto val]$$

$$\texttt{fst}(val_1, val_2) \to val_1$$

$$\texttt{snd}(val_1, val_2) \to val_2$$

$$\texttt{case inl}(val)\ \texttt{of}\ \{\texttt{inl}(x).e_1 \mid \texttt{inr}(x).e_2\} \to e_1[x \mapsto val]$$

$$\texttt{case inr}(val)\ \texttt{of}\ \{\texttt{inl}(x).e_1 \mid \texttt{inr}(x).e_2\} \to e_2[x \mapsto val]$$

where $e[x \mapsto val]$ indicates capture-avoiding substitution of the value val for the variable x within the expression e. All other reductions are structural rules following standard call-by-value evaluation order (Barendregt, 1984).

Static Semantics. As mentioned before, our lambda calculus is simply-typed. Each valid expression can be assigned a single type. The type of an expression is not changed by reduction. A full account of the simply-typed lambda calculus and its static semantics can be found in Barendregt et al. (1992).

Syntactic Sugar. For convenience, we define a few abbreviations, or "syntactic sugar", to improve the readability of sample code. First, although we only have simple pairs (a, b), we will sometimes use longer tuples such (a, b, c). This syntax is understood to mean $(a, (b, c))$. We will use the projection operators π_n on tuples, which is understood to be a combination of `fst` and `snd`. For example, $\pi_2(a, b, c)$ is understood to mean $\texttt{fst}(\texttt{snd}(a, (b, c)))$. Pairs can also be used to encode finite lists, with `unit` representing the empty list. We will use the following syntactic sugar for

lists:

$$[a, b, c, .., z] = (a, (b, (c, ...(z, \texttt{unit})...)$$

$$\texttt{head}\ l = \texttt{fst}(l)$$

$$\texttt{tail}\ l = \texttt{snd}(l)$$

We will also use a shortened function syntax, where $\lambda x{:}\tau_1\ y{:}\tau_2.e$ is understood to mean $\lambda x{:}\tau_1.\lambda y{:}\tau_2.e$. Also, when it is clear from context, we will sometimes omit the type annotation on the lambda term. Finally, we will use a let-syntax as a shorthand for lambda-binding variables: the expression (let $a{:}\tau = b$ in c) is understood to mean: $(\lambda a{:}\tau.c)\ b$. All of these short-hand notations are consistent with the dynamic and static semantics (Barendregt, 1984; Barendregt et al., 1992).

3.2.2 Translation

We will describe translation from assembly language to our target language in two steps. First, we use a purely syntactic transformation which puts the assembly code into a "monadic" form (Moggi, 1989). This translation makes the sequencing between instructions explicit by introducing a sequencing operator called "bind". We write the sequencing operator using the infix syntax $\gg\!=$. The bind operator is not part of the language (as we would need polymorphism to represent it), it is simply a place-holder where we will later insert code which implements the binding operation. We will also use the syntax `lift` v as a place-holder for code which implements the meaning of values[1]. We will write the transformation using oxford brackets ($[\![]\!]$). The

[1]The syntax `return` v is often used in place of `lift` v. However, to avoid confusion with our assembly language `return`, we choose to use the less common "lift".

translation for basic blocks, sequences of statements, values, and control instructions is shown below.

$$[\![\ell : \overline{x_i = s_i}; c]\!] \equiv \ell = \lambda\ FV^*(\ell).[\![\overline{x_i = s_i}; c]\!]$$

$$[\![x = s; \overline{x_i = s_i}; c]\!] \equiv [\![s]\!] \gg\!= \lambda x.[\![\overline{x_i = s_i}; c]\!]$$

$$[\![t\ v]\!] \equiv \texttt{lift}\ v$$

$$[\![\texttt{return}]\!] \equiv \texttt{lift unit}$$

$$[\![\texttt{return}\ v]\!] \equiv \texttt{lift}\ v$$

$$[\![\texttt{br}\ \ell]\!] \equiv [\![\ell]\!]$$

$$[\![\texttt{cbr}\ v\ \ell_1\ \ell_2]\!] \equiv \texttt{if}\ v\ \texttt{then}\ [\![\ell_1]\!]\ \texttt{else}\ [\![\ell_2]\!]$$

$$[\![\ell]\!] \equiv \ell\ FV^*(\ell)$$

The translation converts a block into a function that takes as arguments the transitive free variables of the block: any of the block's free variables, or variables required by its successor blocks. In the case that the set of transitive free variables is empty, we will use `unit` as a sole argument. Sequences of instructions are translated and sequenced together using $\gg\!=$. Values and the return instructions are translated to `lift`-expressions, the unfolding of which will be defined in a later section. The branch instruction is translated into a function call where the function has the name of the target label, and the arguments are the transitive free variables of the target block (or `unit`). The conditional branch is translated to an if-expression which selects one of two blocks to evaluate.

Consider the code shown earlier in Figure 3.6. This code defines a simple loop, and has four basic blocks. If we apply our translation as defined so far (translation

for instructions is not yet defined) we get the following lambda terms for each block.

$$
\begin{aligned}
\mathit{entry} =& \lambda \texttt{unit.lift } 1 \gg= \\
& \lambda t.\texttt{lift } 2 \gg= \\
& \lambda i.\mathit{header}\ t\ i \\
\mathit{header} =& \lambda t\ i.[\![\texttt{slt int32 i, 10}]\!] \gg= \\
& \lambda c.\texttt{if } c \texttt{ then } \mathit{body}\ t\ i \texttt{ else } \mathit{end}\ t \\
\mathit{body} =& \lambda t\ i.[\![\texttt{add int32 t, i}]\!] \gg= \\
& \lambda t.[\![\texttt{add int32 t, i}]\!] \gg= \\
& \lambda i.\mathit{header}\ t\ i \\
\mathit{end} =& \lambda t.\texttt{lift } t
\end{aligned}
$$

3.3 Side Effects

To complete our lambda encoding, we must define the translation for assembly instructions. However, many of the assembly instructions have side-effects that we must track in order to properly model the semantics of programs. For example, the division operation may raise a division by zero error, and we need to model this possible effect. Using our translation so far, we can model side-effects by defining a state monad and giving implementations for each of the instructions within this monad. State monads are a specific instance of a monadic semantics (Jones, 1995). State monads have been used to model side-effects in many contexts, including programming languages like Haskell (Launchbury and Peyton Jones, 1994), formal semantics (Moggi, 1989), and even dependent type theories (Nanevski et al., 2006, 2009). A

more detailed analysis of monads is given in Chapter 4.

For our purposes, a state monad is a function from states to states and a value. For convenience we will give this type a name, ST, defined as:

$$ST\ a = state \to (state, a)$$

where *state* is a fixed but arbitrary type. To complete the definition of our state monad we need to define the transformations `lift` and $\gg=$ which we used in our translation. Values of type $ST\ a$ are called *computations*. The `lift` transformation allows us to build a computation of type $ST\ a$ from a value of type a. The definition is given below.

$$\texttt{lift}\ x = \lambda s{:}state.(s, x)$$

Recall, that `lift` and $\gg=$ are not part of our lambda calculus. Rather, they are short-hand notations which we will replace with their definitions. Therefore, we do not define `lift` as a function from its argument x. Indeed `lift` with no argument is not defined. Note also that each term `lift` x will have a simple type which depends on x.

The second transformation, $\gg=$, sequences an ST-computation with a function. The definition is shown below.

$$(\gg=) :: ST\ a \to (a \to ST\ b) \to ST\ b$$

$$m \gg= f = \lambda s{:}state.\text{let}\ (s', x) = m\ s\ \text{in}\ f\ x\ s'$$

The state monad works by threading an abstract state variable (of type *state*) through the instructions. A state variable may represent many different kinds of side-effects. For example, to model division by zero we can use as our state variable a single

boolean which is `true` when an overflow has occured. We then give the following definition for division:

$$[\![\texttt{udiv}\ x\ y]\!] \equiv \lambda s.\texttt{if}\ s \vee y = 0\ \texttt{then}\ (true, 0)\ \texttt{else}\ (false, \texttt{udiv}\ x\ y)$$

Each division instruction becomes a function which takes the current overflow state, s. If there has not yet been an overflow, and the second parameter y is not zero, we perform the division. If the division is performed, we know there will not be an overflow (since we checked $y \neq 0$), so the overflow state is $false$. Otherwise, the returned overflow state is $true$ and the result of the operation is invalid. For simplicity, we return 0 for the invalid value.

For the other (non-division) instructions, we must check the overflow flag before continuing with normal operation. For example, addition would be defined as:

$$[\![\texttt{add}\ x\ y]\!] \equiv \lambda s.\texttt{if}\ s\ \texttt{then}\ (true, 0)\ \texttt{else}\ (false, \texttt{add}\ x\ y)$$

The new version of `add` is identical to the old version if the overflow state is $false$. Otherwise, the new version does nothing. This pattern can be generalized to any (non-division) instruction:

$$[\![\texttt{i}_t\ \overline{x}]\!] \Longrightarrow \lambda s.\texttt{if}\ s\ \texttt{then}\ (true, 0_t)\ \texttt{else}\ (false, \texttt{i}\ \overline{x})$$

For any instruction i, with return type t, we simply check the overflow state and call the instruction as normal if it is false. Otherwise we pass on the overflow state and an arbitrary value of type t (written 0_t).

Although we have been discussion overflow, this technique is independent of the set of state variables and the transformation we perform on individual instructions.

We can easily extend our state to be a tuple of values representing different kinds of side-effects. For example, an **alloc** instruction returns a pointer, modifies the heap, and may fail. We can add new state variables to track possible failure and changes to the heap. Our use of state variables is inspired by state monads (Moggi, 1989). This design allows us some flexibility in how closely we model the state. By changing how we interpret the instructions, and the parts of the state, we can be more or less precise with our model of the side-effects.

3.3.1 Modeling Memory

For our experiments, we have modeled the effects of instructions on memory. Our model of memory uses a state variable which captures what is known about the heap at any given point in the computation. The state variable is a list of memory operations that have occurred. The operations are either allocate or store. We represent the state variable using tuples constructed with the following functions:

$$Salloc = \lambda n\ s.(0, n, s)$$

$$Sstore = \lambda p\ v\ s.(1, p, v, s)$$

These two functions add information to the current state. The first function, *Salloc* adds to s a tag, 0, indicating an allocation has occurred, along with the number of bytes allocated. The second function, *Sstore* adds to s a tag 1, indicating a store has occurred, along with the pointer and value stored. With these functions, we can

define the state monad versions of `alloc` and `store` which track the state of the heap:

$$[\![alloc\ n]\!] \equiv \lambda s.(Salloc\ n\ s, \texttt{alloc}\ n)$$

$$[\![store\ p\ v]\!] \equiv \lambda s.(Sstore\ p\ v\ s, \texttt{store}\ p\ v)$$

To see how these definitions work, consider the sequence:

```
p = alloc 1; store p 1; x = load p
```

After translation this becomes:

$$\lambda s0.\texttt{let}(s1, p) = (Salloc\ n\ s0, \texttt{alloc}\ n)\ \texttt{in}$$

$$\texttt{let}(s2, \_) = (Sstore\ p\ 1\ s1, \texttt{store}\ p\ 1)\ \texttt{in}$$

$$\texttt{let}(s3, x) = (s2, \texttt{load}\ p)\ \texttt{in}$$

$$(s3, p, x)$$

If we simplify just the `load` term, we have:

```
( (1,p,v,(0,1,s0)), load p )
```

This expression contains all of the relevant information about the computation at this point in the code. We know that one word has been allocated, and that it contains the value v[2]. Therefore we can simplify the `load` expression to v. We will describe this and other simplification laws in a later chapter.

3.4 Completing the Semantics

The translation we have described so far only applies to basic blocks. We can extend our translation to functions by including a fixed-point operator in our target

[2]Although it is not evident in the syntax, there is a link between the memory state for the allocation and the variable p. Our intermediate representation uses graphs and variables are replaced with edges in the graph. In this representation we can link the store state with the initial allocation.

language. For instance, if we add recursive let binding to our target language, then we can define the meaning of functions as:

$$[\![tf(\overline{t_i\ x_i})\{\overline{b_i}\}]\!] \equiv \lambda\overline{x_i : t_i}.\left\langle [\![entry]\!], \overline{b_i = [\![b_i]\!]}\right\rangle$$

where the notation $< a, \overline{b} >$ forms a fixed point over the equations $\overline{b}$ and evaluates to a[3]. If we wish to run a function, we can supply this result with an initial state value. The result of evaluating the application will be a final state and a return value.

Using this translation for top-level functions we can give a complete semantics to a while program. This approach was taken by Necula (2000) to engineer a translation validation system for GCC. This approach does indeed work well for some optimizations, however, for many cases, it requires additional analysis to discover the relationship between the initial and optimized programs. The additional analysis can make the validator sensitive to the specific optimizations, and even the specific implementations of the optimizations being considered. In addition, it is not at all clear how this approach can be extended to structure-changing optimizations such as loop-fusion, loop-fission, or loop-invariant code motion.

These observations have been made by a number of authors (Huang et al., 2006; Necula, 2000; Pnueli and Zaks, 2008; Rival, 2004; Tristan and Leroy, 2010), but to the best of our knowledge the reasons why this style of semantics is problematic have not been clearly explained. In the remainder of this chapter we will show how this approach causes difficulties for translation validation. We will then provide a more precise semantics based on λ-graphs which will allow us to motivate our final intermediate form.

[3]In Standard ML this would be written as: `let rec` $\overline{b}$ `in` a.

3.4.1 Translation Validation

Translation validation requires us to compare the terms of our intermediate form for equality. The most basic equality checking is simply syntactic equality: two terms are equal if they are syntactically identical. If we can loosen our definition of equality, we will be able to validate more programs. For instance, if our intermediate form is the lambda calculus we have described in this chapter we can add α-equivalence which changes the rule for comparing lambda terms to:

$$\frac{a[x \mapsto z] \equiv b[y \mapsto z] \qquad z \notin FV(a) \cup FV(b)}{\lambda x.a \equiv \lambda y.b}$$

We can also add a rule which allows us to equate terms with their reductions. For our simple lambda calculus this amounts to adding the β-equivalence rule.

$$(\lambda x.a)b \equiv a[x \mapsto b]$$

These two rules satisfy an important property: together, when interpreted as a directional rewrite system they are *confluent*. This means that no matter how we apply these rules the equivalence relation gives the same result. A system which is not confluent captures fewer equalities and is less useful for translation validation.

If we extend our language to include a fix-point operator as described in this section, we need to look for additional equality rules we can use to capture equivalences between terms. An obvious choice is to use the λ_μ calculus which contains a fix-point operator and is known to be confluent with the following equality rule which corresponds to the new reduction rule:

$$\mu x.e \equiv e[x \mapsto \mu x.e] \qquad .$$

Unfortunately, the standard λ_μ calculus is ineffective in our setting, and cannot be used for translation validation.

To see why λ_μ is ineffective, consider the example of a function for testing the evenness of a natural number defined in terms of two mutually recursive functions.

$$isEven\ x = \left\langle even\ x \;\middle|\; \begin{array}{l} even = \lambda x.\text{if } x == 0 \text{ then } true \text{ else } odd(x-1) \\ \\ odd = \lambda x.\text{if } x == 0 \text{ then } false \text{ else } even(x-1) \end{array} \right\rangle$$

Using our bracket syntax, we write $isEven$ with the function body $even\ x$ on the left of the bracket, and the set of mutually recursive bindings on the right of the bracket. We can encode this (or any) bracket definition using λ_μ. To do so we form a tuple containing the bodies of the mutually recursive bindings. Then we will apply the fix-point operator to the tuple and project out the component we are interested in.

$$[\![isEven]\!] = \pi_1 \mu p. \left(\begin{array}{l} \lambda x.\text{if } x == 0 \text{ then } true \text{ else } \pi_2 p(x-1), \\ \\ \qquad \lambda x.\text{if } x == 0 \text{ then } false \text{ else } \pi_1 p(x-1) \end{array} \right)$$

Now, suppose we apply some optimizations to our original $isEven$ function. For example, we can inline *odd* into the body of *even*, then apply constant folding followed by dead-code elimination. The result is that the helper function *odd* has been removed.

$$isEven'\ x = \left\langle even\ x \;\middle|\; \begin{array}{l} even = \lambda x.\text{if } x == 0 \text{ then } true \\ \qquad\qquad \text{else if } x == 1 \text{ then } false \text{ else } even(x-2) \end{array} \right\rangle$$

If we translate $isEven'$ to λ_μ we get the following definition.

$$\begin{array}{l} [\![isEven']\!] = \mu\ even. \\ \qquad \lambda x.\text{if } x == 0 \text{ then } true \\ \qquad \text{else if } x == 1 \text{ then } false \text{ else } even(x-2) \end{array}$$

The problem of translation validation is to show that $isEven$ is semantically equivalent to $isEven'$. Using λ_μ as our intermediate language, this means we must prove these two functions are equivalent using our equality rules. However, these two terms are *not* equivalent with the rules of λ_μ.

$$\not\vdash [\![isEven]\!] \equiv_\mu [\![isEven']\!]$$

It appears that our theory of equality is too weak to reason about these optimizations. We could try to enhance our system of equality to handle specific cases, however such attempts invariable lead to loss of confluence (Ariola and Klop, 1994).

The difficulty with λ_μ is not that we do not have enough equality rules, but rather that the equality rules we have are too coarse. In order to reason about transformations "under a μ," we need to break the reduction and equality up into smaller operations. We can do this by formulating a calculus that exposes the underlying graph structure of the lambda terms. Then we can develop a calculus of graph transformations; this type of calculus is normally called a "theory of cycles." In developing

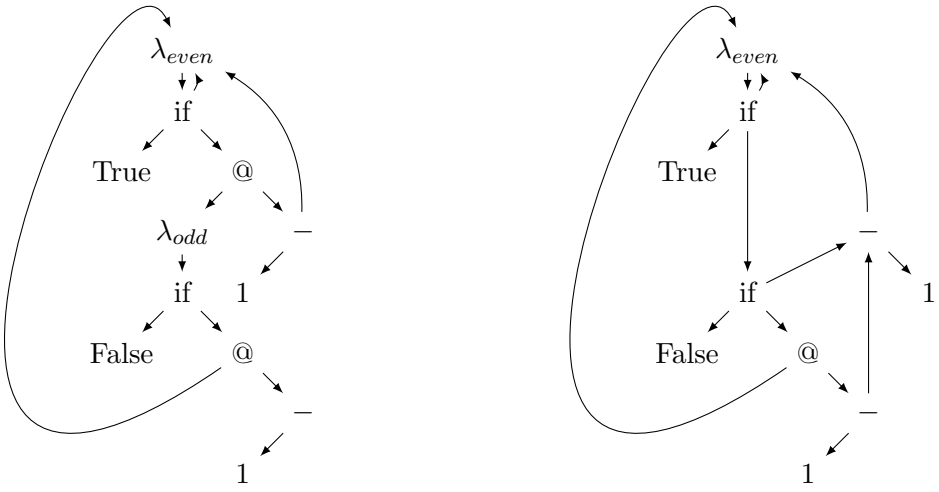
such a theory, we must be careful not to break confluence lest we make no progress. Ariola and Blom (1997) summarizes the situation this way:

> We conclude that a theory of cycles is necessary if one wants to reason about compilation optimization and execution of programs.
>
> What makes a theory of cycles difficult to develop is that, once lambda-abstraction and cycles are admitted, confluence is lost [...] To regain confluence, current formulations of cycles either impose restrictions, such as disallowing reduction under a lambda-abstraction or on a cycle, or adopt a framework based on interaction nets.
>
> Ariola and Blom (1997)

As we will see in Chapter 4, our approach falls somewhere in between: we have restrictions on cycles, and we also use a very simple form of interaction net (Girard, 1989). First, we will look at a semantics based on λ-graphs powerful enough to handle translation validation under optimizations. Then, we will show how we can restrict this to arrive at the basic design of our intermediate form.

3.4.2 λ-Graphs

A lambda term can be represented as a graph. Each variable is replaced by a graph edge which points back to the variable binding site. For example, the graph representations of the $isEven$ and $isEven'$ functions are shown in Figure 3.8. We can reason about recursive definitions and optimizations by developing an algebra which corresponds to primitive operations on these graphs. one such algebra is a the representational calculus developed by Ariola and Blom (1997); Ariola and Klop (1994).



(a) Original Funtion (b) Optimized function

Figure 3.8: Graph representation of isEven function

β Reduction:	$(\lambda x.M)\ N = \langle M \mid x = N \rangle$
External Substitution:	$\langle C[x] \mid x = M, ... \rangle = \langle C[M] \mid x = M, ... \rangle$
Tree Shaking:	$\left\langle M \mid x = N, \overline{D} \right\rangle = \left\langle M \mid \overline{D} \right\rangle$
	$x \notin FV(\overline{D})$
Associativity:	$\langle M \mid x = \langle N \mid D \rangle, Y \rangle = \langle M \mid x = N, D, Y \rangle$
Internal Substitution:	$\langle M \mid x = C[y], y = N \rangle = \langle M \mid x = C[N] \rangle$
	$x \neq y \wedge$ y not shared

Table 3.1: Subset of λ-graph rewriting rules

Recall, that we would like to represent an assembly function as the fixed point of the denotation of its basic blocks:

$$[\![tf(\overline{t_i\ x_i})\{\overline{b_i}\}]\!] \equiv \lambda \overline{x_i : t_i}. \left\langle [\![entry]\!], \overline{b_i = [\![b_i]\!]} \right\rangle$$

To use this for translation validation, we need to be able to reason about transformations which occur within the set of recursive definitions. That is, we need a set of equality rules which allow us to equate the graphs in Figure 3.8. The representational calculus allows us to do this.

Some of the equality rules of representational calculus are shown in Table 3.1. The first three rules give us the equivalent of the usual β-reduction. The first rule says that we can rewrite a lambda application as a recursive definition in which the bound variable is assigned the value of the function argument. The second rule, "External Substitution", allows us to perform an explicit substitution on the externally visible parts of the expression. Finally, the third rule, "Tree Shaking", allows us to remove unused definitions. These three rules give us the equivalent of β-reduction, but broken up into smaller operations. If we use these rules as a left-to-right rewrite system,

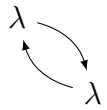
applying a function is done in three steps.

$$(\lambda x.e)\ a \rightarrow \langle e \mid x = e \rangle \rightarrow \langle e[x \mapsto e] \mid x = e \rangle \rightarrow \langle e[x \mapsto e] \rangle$$

To equate $isEven$ and its optimized version, $isEven'$ we also need the "Internal Substitution" rule. Using this rule we can perform the critical step of inlining the definition of *odd*. This followed by application of the other rules allows us to conclude that $isEven$ and $isEven'$ are equivalent.

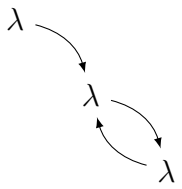
Completeness. The fragment of the representational calculus we have presented is incomplete. The incompleteness is fundamentally linked to cycles in the graph representations of our lambda terms. To see why this is so, consider the following term involving two mutually recursive lambda functions.

$$\langle y \mid y = \lambda z.w, w = \lambda x.y \rangle$$



This mutually recursive term forms a small cycle as we can see from the λ-graph representation. We can unfold this term one time to obtain an equivalent, yet larger version.

$$\langle y \mid y = \lambda z.w', w' = \lambda x.y', y' = \lambda x.w' \rangle$$



The two λ-graphs shown above are equivalent, however we cannot yet prove this is the case. Intuitively, we can see they are equal by providing a mapping between the variables in the original and unfolded terms. If we use the mapping:

$$\{w' \mapsto w, y' \mapsto y\}$$

then we can substitute in the unfolded term to arrive back at the original term.

To create a complete system, we can generalize the process of creating a mapping to equate terms. Following Ariola and Blom (1997), we introduce the "copy rule" shown below.

$$M = N \qquad \exists \sigma : Var \to Var.\sigma(N) = M$$

This rule says that if we can come up with a mapping of variables that when applied to one term makes it equivalent to the other term, then we can conclude the two terms are equivalent. The copy rule makes our system complete, but makes the decision procedure undecidable because we may need to find substitution functions at any point in the process.

The σ functions appear in many translation validation under various names. The σ functions are similar to Necula's bisimulation relation which are computed heuristically based on intimate knowledge of the compiler (Necula, 2000). The σ functions are similar to Tristan's `st_eq` function which is provided by the compiler to aid validation (Tristan and Leroy, 2008, 2010). Finally, the σ functions are similar to Pneulli's

α function which is also provided by the compiler to aide validation (Pnueli et al., 1998).

LLVM M.D. In order to design a system which is scalable, we have chosen not to include σ functions in our design. This design allows our tool to operate independently of the compiler. In the most general terms, this implies that our system is incomplete. The degree to which this incompleteness is important in practice is a question we address in Chapter **??**. However, we know that our situation is improved by the fact that our input programs are in a special form.

Our input programs consist of functions with a top-level recursive binding of simply-typed definitions. That is, if we have a top-level function of the form

$$\langle M | \overline{x_i = N_i} \rangle$$

we know that each N_i is a simply typed lambda term with no nested recursions. We can take advantage of this structure by pre-processing each function, aggressively applying the β-reduction and associativity rules. If we aggressively apply these two rules we end up with a set of first-order recursive equations. Therefore, we can consider each function to be a set of first-order, possibly recursive, equations. In the next chapter we will introduce a graph-based language for combining basic blocks into complete programs which treats each function as a set of first-order recursive equations.

Chapter 4

Synchronous Value Graphs

In this chapter we present our intermediate language and its semantics. We formally present Synchronous Value Graphs (SVG), our final intermediate form. This language builds on the monadic assembly language described in the previous chapter. Assuming we are starting with monadic assembly language, then compiling to Synchronous Value Graphs will give us referentially-transparent terms which are amenable to translation validation. The compilation process will be described in Chapter 6. The algebraic rules are justified by a categorical semantics which we describe in Section 4.1, and mechanically formalized in Chapter **??**.

Our intermediate language is called Synchronous Value Graphs (SVG). This language is a derivative of Gated Single Assignment form (Havlak, 1993; Tu and Padua, 1995a). The language is formalized as a directed graph where each node represents a part of the overall computation. The nodes are constructed from the terms described in Section 4.0.3, and these may include the simply-typed lambda terms described in the previous chapter.

Assuming we are starting with the monadic assembly language previously described, then compiling to SVG form will give us a referentially-transparent intermediate representation that captures all of the relevant aspects of the original assembly code. Our final terms are referentially transparent both with respect to the side-effects modeled by our monad, and with respect to the control-flow. With our graph representation in hand, we can discard the control-flow graph and symbolically manipulate our representation using algebraic rules.

Why Synchronous? The name Synchronous Value Graphs refers to the semantic model of our intermediate language. Each function is represented as a graph of computational nodes with the edges representing dependencies between the nodes. Semantically, the graph is interpreted as a circuit where each node consumes inputs from its in-edges and produces outputs on its out-edges. The *synchronous* moniker refers to our semantic model in which each edge represents exactly one value at any given point in the computation. Contrast this with a non-synchronous model in which each edge represents a (possibly infinite) buffer of values.

While it is possible to use a non-synchronous model for our value graphs, we have chosen a synchronous model. Our choice to use a synchronous model is motivated by

several concerns. First, our synchronous model allows us to more easily use certain algebraic rewrite laws. Note, that our semantics is similar to a data-flow semantics (Caspi and Pouzet, 1996; Kahn, 1974). In such a semantics, synchrony allows for simplifications which correspond to the process of deforestation (Wadler, 1990) in functional languages. Because the edges of the graph do not have any storage requirements, deforestation is very easy to justify (Wadler, 1984). It is possible to use deforestation-like rewritings in non-synchronous settings (Gill et al., 1993), however they are more difficult to justify (Meijer et al., 1991), and may not apply in our setting.

Second, our synchronous semantics ensures that we have a well-defined reduction semantics. Because of this, it is easy to implement as an abstract machine interpreter. Our original semantics was not synchronous, and we found that implementing an interpreter for value graphs can be very difficult, especially in the presence of nested loops or diverging programs. The synchronous requirement, and the associated reduction semantics, ensures that a simple interpreter will not consume unbounded memory. Currently, we do not need to interpret our graphs to perform validation. However, future work may involve interpreting value graphs or compiling graphs back to assembly code.

Finally, in our estimation, the synchronous semantics is more elegant. The synchronous requirement refines the space of graphs we are considering, and requires us to be very precise in our translation to SVG. Other graph rewriting systems are by-and-large synchronous: both implementation projects such as data-flow languages (Berry and Gonthier, 1992; Halbwachs et al., 1991; Wadge and Ashcroft, 1985), and

theory projects such as the geometric representation of linear logic (Girard, 1989). These similar systems give us confidence that our choice of a synchronous semantics is a good one.

The remainder of this chapter is organized as follows: First, we will present the formal syntax of Synchronous Value Graphs's with an informal description of its semantics and accompanying examples. We will then give some background on the categorical constructs we will use in our semantics. Finally, we will present the formal categorical semantics for our value graphs. A mechanized version of our semantics can be found in Chapter **??**.

4.0.3 SVG Syntax

P	::=	(x,**DS**)	Program
DS	::=	$\{ \overrightarrow{x \mapsto \textbf{term}} \}$	Definition Set
term	::=	x, y, z, ...	Variables
	|	`const` **expr**	Constants
	|	`map` **expr term**	Function Application
	|	`zip`(**term**, **term**)	Pair Construction
	|	ϕ(**term**,**term**,**term**)	Choice
	|	μ(**term**,**term**)	Sequence
	|	η(**term**,**term**)	Selection
	|	σ(**term**,**term**)	Buffer

Figure 4.1: Syntax of Synchronous Value Graphs

The syntax for Synchronous Value Graphs is show in Figure 4.1. The SVG intermediate form represents a program as a set of assignments of variables to terms: we

call this a *definition set*, **DS**. A complete program, **P**, is represented as a definition set together with a variable representing the final result of the program. This variable gives us a term from which we can extract the final value using the definition set.

Note that Synchronous Value Graphs do not include a control-flow graph, nor any blocks, labels nor control constructs. The SVG form is referentially transparent, and all control-flow dependency is made explicit in the terms. Once we have SVG form, there is no additional information encoded in the control-flow graph. Therefore, we dispense with the control-flow graph at this point, and rely only on the definition set.

The term language includes the language of the expressions (**expr** or e) we used to define the monadic forms of instructions in the previous chapter. The `const` and `map` terms are used to lift expressions up into the term language, and `zip` is used to create streams of pairs from two streams. The ϕ-term is similar to the SSA ϕ-node described in Chapter 3, however the new ϕ-term also carries a condition which tells us which of the two remaining terms will be the final value. The last three terms are used to encode loops.

Streams. Our goal is to interpret programs built from our term language as a "circuit" of terms. Each term is connected to its inputs and outputs forming a precise, symbolic represetation of the program. This way of modeling our langauge is inspired by synchronous data-flow programs. In this setting every term represents an infinite stream of values. We will use a Haskell-like notation to describe streams informally. The notation:

$$x : s$$

represents the stream with first element x, followed by the stream s. We will also use Haskell-like equations to describe streams. For example, the equation:

$$ones = 1 : ones$$

represents the infinite stream where each element is the number 1. This last equation can be understood as replacing the name *ones* with its definition *as needed* to produce a sequence.

In order to give our term language an operational semantics, we will also need to allow streams to have "empty values", written as ϵ. An empty value can be thought of as a missing element in a stream. The empty value allows a stream producer to fail to produce an element without changing the frequency of items being produced. As an example, below is a stream of prime numbers with empty values for the composite numbers:

$$[\epsilon, 2, \epsilon, \epsilon, 5, \epsilon, 7, \ldots] \qquad .$$

Such a stream might be produced by filtering a stream of natural numbers; the stream continues to produce elements at the same rate with ϵ standing in for the non-prime numbers. Including the empty values will allow us to align this stream with, say, the original stream of natural numbers. We will return to the issue of element frequency and empty values when we discuss loops.

Lifting. The language of expressions we defined in the previous chapter contains, integers, booleans and pairs, but not streams. In order to use our expressions on streams we need to "lift" them up into the term language. The term language provides three mechanisms for doing this. The simplest of these is the `const` term. The term

`const` e creates a stream in which each element is the expression e. Informally, we can define `const` as:

$$\texttt{const}\ v = v : \texttt{const}\ v \quad .$$

The second lifting term is `map`. The term `map` f s applies the function f to each element in the stream s. If s contains an empty value, the an empty value is returned instead of applying the function. Informally, `map` can be defined as:

$$\texttt{map}\ f\ (\epsilon : xs) = \epsilon : \texttt{map}\ f\ xs$$

$$\texttt{map}\ f\ (x : xs) = f\ x : \texttt{map}\ f\ xs$$

Handling ϵ in this way allows us to make *map* a total function even though ϵ is not in the range of f. Using `const`, and `map` we can define the stream of the number 2 as:

$$twos = \texttt{map}\ (\lambda x.x + 1)\ (\texttt{const}\ 1)$$

Finally, `zip` allows us to combine two streams into a stream of pairs of values. Informally, `zip` is defined as:

$$\texttt{zip}(\epsilon : xs, \epsilon : ys) = \epsilon : \texttt{zip}(xs, ys)$$

$$\texttt{zip}(x : xs, y : ys) = (x, y) : \texttt{zip}(xs, ys)$$

Note, that `zip` is only defined if either both streams are empty values or both streams are non-empty values. Unlike `map`, `zip` is not a total function. To ensure that uses of `zip` are well defined we will impose a synchronous semantics on our terms. Using `zip` we can ensure that the arguments to a function are all available at the same time. For example, we can define addition on streams as:

$$sadd = \texttt{map}\ (\lambda(x, y).\texttt{add}\ x\ y)\ \texttt{zip}(x, y)$$

where `add` is the function defined in Chapter 3 for adding integers. Note, that *sadd* will only be defined for streams x and y that produce values at the same frequency. Put another way, if x can produce an empty value when y does not, then the function *sadd* is undefined for those two streams.

Using the terms we have so far, we can give a complete example of a program in SVG. Consider the function `f` defined below.

```
int f() {
  return 1 + 2;
}
```

The SVG representation of this function is[1]:

$$f = \left(z, \left\{ \begin{array}{l} xys \mapsto \texttt{zip}(\texttt{const}\ 1, \texttt{const}\ 2) \\ \\ \quad z \mapsto \texttt{map}\ (\lambda(x, y).\texttt{add}\ x\ y)\ xys \end{array} \right\} \right)$$

The function is represented as the final variable z, and a set of definitions; in this case a definition for z and a definition for xys. The result, z, is computed by mapping the addition function over a stream of pairs of the form $(1, 2)$. The result is a constant stream of the value 3. By using `zip` we have ensured that the input streams are "in sync", and a finite symbolic value can be computed.

Gating. In order to make the control-flow explicit in our terms, we equip the usual ϕ-nodes with a condition that tells us which of the two expressions will be used. Recall, in SSA form, when the control flow is split by a conditional, the join point must contain a ϕ-node which recombines the definitions from the two branches. For example, if we have:

[1]Here we are ignoring the memory state, but we could recover this by using the monadic version of `add` from the previous chapter.

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

Then, the corresponding SSA assembly language is:

```
entry:
  cbr c l1 l2
l1:
  x1 = 1
  br join
l2:
  x2 = 2
  br join
join:
  x = phi(l1,x1; l2,x2)
```

In SVG, we include the condition c in the ϕ-node. We can then remove the control flow, which gives us:

$$x1 \mapsto 1$$

$$x2 \mapsto 2$$

$$x \mapsto \phi(c, x1, x2)$$

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**. The value of x is captured by the SVG equations even though we have removed the labels, blocks, and control-flow instructions. Informally, we can define ϕ using the following rules:

$$\phi(\epsilon : cs, \epsilon : xs, \epsilon : ys) = \epsilon : \phi(cs, xs, ys)$$

$$\phi(\textbf{true} : cs, x : xs, y : ys) = x : \phi(cs, xs, ys)$$

$$\phi(\textbf{false} : cs, x : xs, y : ys) = y : \phi(cs, xs, ys)$$

Note, like `zip`, all of the input streams must produce either empty or non-empty values for ϕ to be defined.

Loops. SVG uses two constructs to represent simple loops. The first, μ, is used to define variables that are modified within a loop. Each μ is placed at a loop header and holds the initial value of the variable on entry to the loop and a value for successive iterations. The μ-term is equivalent to a non-gated ϕ node from classical SSA. The second, η, is used to refer to loop-defined variables from outside their defining loops. The η-term carries the variable being referred to along with the condition required to reach the η from the variable definition. For example, consider the following simple loop:

```
int i = 0;
while (i < 3) i++;
```

The SVG for of this loop is shown below:

$$
\begin{aligned}
i_0 &\mapsto 0 \\
i_n &\mapsto \mu(i_0, i_{n+1}) \\
i_{n+1} &\mapsto i_n + 1 \\
c_n &\mapsto i_n < 3 \\
i &\mapsto \eta(c_n, i_n)
\end{aligned}
$$

The variable i is modified within the loop. Therefore, within the loop it is defined using μ. The initial value of i in the loop is i_0, and each successive value is defined by i_{n+1}. The final value of i is defined using η. The η-term will evaluate to i_n only with the condition c_n is false—the loop has finished. Informally, we can define μ and

η as:

$$\mu(\epsilon : xs, y : ys) = \epsilon : \mu(xs, ys)$$

$$\mu(x : xs, y : ys) = x : ys$$

$$\eta(\epsilon : cs, \epsilon : xs) = \epsilon : \eta(cs, xs)$$

$$\eta(\textbf{true} : cs, x : xs) = \epsilon : \eta(cs, xs)$$

$$\eta(\textbf{false} : cs, x : xs) = \texttt{const}\ x$$

Using these definition for μ and η we can track the evolution of the streams defined in the SVG representation of our loop. The table below shows the value of each of the streams as time progresses.

	t_1	t_2	t_3	t_4
i_0	0	0	0	0
i_n	0	1	2	3
i_{n+1}	1	2	3	4
c_n	t	t	t	f
i	ϵ	ϵ	ϵ	3

As we enter the loop at time-step t_1, the value of i_n is equal to i_0; i_{n+1} is equal to $i_0 + 1$; our condition c_n is true, and the final value i is undefined.

On the second iteration of the loop, i_n becomes i_{n+1} which had a value of 1 on the previous iteration. The μ term tells us that this is where we should break the recursions and use the previous value of i_{n+1}. Finally, on the fourth iteration, the condition c_n becomes false, and the final value i becomes i_n which is 3.

The above description is not entirely accurate. Notice, even though the variable i_0 is defined outside of the loop, we have consumed four zeros from the stream i_0

while processing the loop variables. This is not a problem in this case because i_0 is a constant stream. However, the stream i_0 and i should produce values at the same rate (since they are both defined outside the loop). If we tried to zip up i_0 and i this would fail.

If we were to `zip` i_0 and i, then in data-flow terminology we would have a *non-synchronous* program, meaning that the different streams are not properly synchronized. Non-synchronous programs cannot be efficiently compiled, and may require an infinite amount of storage to evaluate. In our case, such programs may produce infinitely large symbolic values and must be avoided. More urgently, if the variable i_0 was not constant we would have the wrong semantics! This can happen in the case of nested loops, which we consider next.

Nested Loops. Consider the slightly more complex example involving nested loops show below:

```
int x,i = 0;
while (c) {
  while(d) x=i;
  i++;
}
```

Let us assume that the conditions c and d each allow for two iterations: The outer loop will iterate twice, and the inner loop will iterate twice on each outer iteration for a total of four iterations. More concretely, assume that both c and d are the stream $(t : t : \texttt{const}\ f)$. Within the outer loop, there are two variables that are changing i and x. As before, i is a μ node with initial value 0 and is incremented on each iteration. The variable x is defined within the inner loop, but it also has a value in

the outer loop. In the outer loop, x starts with value 0 and on each iteration of the outer loop is an η node with condition d and the value of x within the inner loop: in this case i_n. These definitions are shown below.

$$i_n \mapsto \mu(0, i_n + 1)$$

$$x_n \mapsto \mu(0, \eta(d, i_n))$$

If we follow the values for i and x, we can immediately see these definitions are incorrect. The table below shows the error.

	t_1	t_2	t_3	t_4	t_5	t_6
d	t	t	f	t	t	f
i_n	0	1	2	3	4	5
x_n	0	ϵ	2	ϵ	ϵ	5

As we step through the iterations of the inner loop, the stream i_n is consumed and the final value for x is 5, when it should be 2. The problem is that the stream i_n is being consumed by the inner loop but it is not properly synchronized: i_n is being consumed "too fast".

To fix this problem, we introduce our final term σ. The σ-term allows us to "speed up" a stream by duplicating values within the stream according to some condition. Informally, we can define σ as:

$$\sigma(v, \epsilon : cs, \epsilon : xs) = \epsilon : \sigma(v, cs, xs)$$

$$\sigma(v, \mathbf{true} : cs, x : xs) = v : \sigma(v, cs, xs)$$

$$\sigma(v, \mathbf{false} : cs, x : xs) = x : \sigma(x, cs, xs)$$

The σ term takes a buffered value, a stream of conditions, and a stream of values. When the condition is true, the buffered value is copied to the output. When the condition if false, the current value from the value stream is copied to the output. Using σ we can fix our example as follows:

$$i_n \mapsto \mu(0, \sigma(0, d, i_n + 1))$$

$$x_n \mapsto \mu(0, \eta(d, i_n))$$

With this new definition we now have the expected result:

	t_1	t_2	t_3	t_4	t_5	t_6
d	t	t	f	t	t	f
i_n	0	0	1	1	1	2
x_n	0	ϵ	1	ϵ	ϵ	2

As we step through the iterations of the inner loop, the values of i_n are duplicated. Each time the inner loop terminates we buffer a new value from i_n.

Intuitively a σ-term must be placed on any non-constant term that is being used with a loop where it is not defined. In other presentations of Gated SSA form, this issue is dealt with by indexing all of the μ and η nodes. While this is equivalent, we prefer our formulation as σ is a natural dual to η: η allows us to refer to variables defined within a loop from outside by "slowing down" the stream, and σ allows us to refer to variables defined outside of a loop from inside by "speeding up" the stream. Also, our formulation more closely matches existing data-flow languages and provides a cleaner semantic framework. We now turn our attention to the formal semantics of SVG.

4.1 Categorical Semantics

In this section we present a semantics for Synchronous Value Graphs. Our semantics is denotational and makes use of basic category theory. We prefer this denotational model because it provides us with a number of algebraic laws that we can use to transform our language during the normalization process. While an operational semantics may provide more intuition, proving that our normalizations are sound is more difficult in an operational setting. However, our specific representation is inspired by the operational semantics of data-flow languages. Because of this, we believe that our denotational model is relatively intuitive in addition to being easier to work with formally.

In the remainder of this chapter we will present our denotational model. First, we will motivate our model and provide some mathematical intuition. Then, we will review some basic categorical constructions and give a precise definition of the category of objects in our model. We will then give a specific representation of our language and show that it forms such a category.

4.1.1 Motivation

Perhaps the most used mathematical object in programming language semantics is the monoid. A monoid is a set together with an identity element and an associative composition operation. Using a Haskell-like notation, we can write the definition of a monoid as:

```
class Monoid a where
    zero :: a
```

```
(⊕) :: a → a → a
```

Here, we are defining the notion of a monoid in which a is a class of objects, *zero* is the identity element, and $\oplus$ is our composition operation. In addition, each instance of our `Monoid` class must satisfy the following laws:

$\forall x, y, z \in a,$

$x \oplus y \in a$	Closure
$zero \oplus x = x \oplus zero = x$	Identity
$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	Associativity

We can specify that a class of objects forms a monoid by giving specific definitions for *zero* and $\oplus$. For instance, the singleton set is a monoid, which can be specified with the following definitions:

```
data Singleton = X
instance Monoid Singleton where
  zero   = X
  X ⊕ X = X
```

It is easy to verify the monoid laws are satisfied for this case.

A more compelling example of monoids is sequencing program actions. Consider a set of program actions P, a sequencing operation ";", and the "identity action" *skip*. This forms a monoid $(P, skip, ;)$, which we can specify as:

```
data Action = Action P | Skip | Seq Action Action
instance Monoid Action where
  zero = Skip
  (⊕) = Seq
```

We can use this definition and the monoid laws to reason about sequences of program actions. For instance, using the monoid laws, we can conclude that:

$$\forall x, y, z \in P, (x; skip); (y; z) = (x; y); z = x; (y; z) \qquad .$$

While this is useful, the monoid does not give us a way to reason about side-effects, or non-total actions (e.g. actions that do not terminate). In order to model side-effects and non-totality, we need a more general structure. The essential motivation behind our use of category theory in this work is to extend the monoid structure to non-total computations with side-effects.

Categories. In order to motivate our more general structure, we will first reformulate our monoid in the language of categories. A category is a collection of "objects" and "arrows" between the objects. Informally, we can think of a category as a kind of graph where the elements and edges of the graph can be given different meanings in different contexts. Category theory studies the properties of these "graphs" where the specific meanings of the elements and edges are left abstract. This is convenient for computer science since we often deal with graph-like structures and we would like to be able to reason about the general properties of these structures.

More formally, a category is a class of objects, a class of morphisms between objects (the "arrows"), and an associative composition operation on the morphisms. A category is "small" if we have sets of objects and morphisms rather than proper classes. In our Haskell-like notation we can specify small categories by considering our objects to be types:

```
class Category (cat :: ∗ → ∗ → ∗) where
```

```
idC   :: cat a a
( ∘ ) :: cat b c → cat a b → cat a c
```

Here, our objects are the elements of the kind $*$, a.k.a. the types. Morphisms for a particular category are objects of type (`cat a b`) where `a` and `b` are types. We require that identity morphisms exists `idC`, and that we have a composition operator $\circ$. In addition, each instance must satisfy the following laws:

$$\forall \tau \in *, id_\tau \in cat\ \tau\ \tau \qquad \text{Identity}$$

$$(x \circ y) \circ z = x \circ (y \circ z) \qquad \text{Associativity}$$

The first law says that for every object in the category we must have a morphism from the object to itself. Our definition satisfies the first law by (brutally) requiring that we have a term of type $\forall a.cat\ a\ a$. The second law requires that our composition operator is associative.

We can recast our monoid as a category by noting that every monoid is equivalent to a category with only one element. That is, given a monoid $(M, \oplus)$, we can construct a category with a single object, and with a morphism for each element in M. Composition in the category is given by the monoid operation $\oplus$. For our category of program actions, we might think of the single object as the type of machine configurations. Then, each program action is a function between configurations, and composition is function composition.

Returning to the problem of side-effects and non-termination: how can we model actions which, for instance, produce an error? One solution, originally proposed by Moggi (1989), is to enrich the monoid category with more objects. To model errors, we can introduce a new object $\perp$ which represents that an error has occurred. We

	Actions	Actions with Errors
Objects	P	$P, \perp$
Morphisms	$P \to P$	$P \to (P + \perp)$
Composition	Function Composition	`comp`

Table 4.1: Difference between monoid and monad categories.

want our program actions to be functions from the "normal" machine configurations to machine configurations or $\perp$. The morphisms of the category now include the usual program actions from our original object to itself, and morphisms from the original object to $\perp$. These new morphisms are program actions that produce an error. Our composition operator is the same as function composition, but we must check for errors. We give a definition for the new composition as the function `comp` below.

```
bind x f = if x == ⊥ then ⊥ else f x
comp f g = λx. bind (f x) g
```

For convenience we have defined `comp` in terms of `bind`, which will be useful later.

The transformation of our monoid category to handle errors is summarized in the Table 4.1. This new category with two elements has all of the nice properties of our monoid, but has been extended to programs which can fail[2]. Note that we are able to do this without knowing anything particular about the set of program actions P. We could repeat this process for side-effects, non-termination, and other kinds of non-totality. If we were to do this a pattern would arise. For each kind of side-effect we want to model, we would first augment our category with new objects capable of representing the effect. Then, we provide a transformation from the

[2]Technically, we need to add one more morphism $\perp \to \perp$ to complete the definition.

original objects and morphisms to the augmented ones. Finally, we have to define a composition operation which preserves the properties of the original structure. Doing this systematically results in a class of *monads*, which are monoids over such transformations. In fact, by systematically adding in support for effects and non-totality, we have uncovered a class of *higher-order* monoids.

Monads. In the previous example we modeled errors by transforming the category enriched with one extra object. We can generalize this technique by considering all possible transformations on an initial category. A transformation on a category (and its morphisms) is called a *functor*. Formally, a functor is a transformation on the objects, and a transformation on the morphisms. For our small category of types we can represent this as:

```
class Functor (t :: ∗ → ∗) where
  map :: (a → b) → (t a → t b)
```

The type constructor `t` transforms objects, and the function `map` transforms morphisms. In addition, a functor must satisfy the following algebraic laws which preserve the structure of the category[3]. For a functor (t, map) between categories C and D, and for all $x, y, f, g \in C$:

$$t\ x \in D, \qquad map\ f \in D$$

$$map\ (id_\tau) = id_{(t\ \tau)}$$

$$map(f \circ g) = map\ f \circ map\ g$$

[3]Technically, a functor is required to be a homomorphism between categories.

If the categories C and D are the same (as in our case), then the functor is called an *endofunctor*. We would like to impose our convenient monoid structure on these functors. Doing so gives us monads: a monad is a monoid over the category of endofunctors.

More formally, a monad is a functor together with two transformations for each object in the category that gives rise to the properties we desire. These transformations on the objects will give us a way to construct the composition operation for our new category transformed by the functor. The two transformations are typically called `unit`, and `join`, with the following types:

```
unit :: a → t a
join :: t (t a) → t a
```

where `t` is our functor, and `a` is the specific object in question. The `unit` and `join` transformations must obey the associativity and identity laws below:

$$join \circ map\ join = join \circ join \qquad \text{Associativity}$$

$$join \circ unit = join \circ map\ unit \qquad \text{Identity} \quad .$$

While this gives us a higher-order analog of a monoid, it is not easy to see from the `join` and `unit` transformations. The monoid-like structure is easier to see if we introduce a derived transformation `bind` with the following type and definition:

```
bind :: t a → (a → t b) → t b
bind x f = join (map f x)
```

In Haskell, `bind` is written as the infix operator `>>=`. Using the infix vesion of `bind` we can rewrite the monad laws in a more intuitive way:

$$unit\ x \gg\!= f = f\ x \qquad \text{Left Unit}$$

$$m \gg\!= unit = m \qquad \text{Right Unit}$$

$$a \gg\!= (\lambda x.b\ x \gg\!= c) = (a \gg\!= \lambda x.b) \gg\!= c \qquad \text{Associtivity}$$

From these laws we can see that monads indeed give us a higher-order analog to our simple monoid: the monad generalizes the monoid to arbitrary functors. We can represent monads in our small category of types as:

```
class Functor m ⇒ Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

This definition says that if `m` is a functor, then `m` is a monad given the two operations `return` and `(>>=)`. The `return` function is a polymorphic version of `unit`, and `(>>=)` is a polymorphic version of `bind`. This implies that we require each unit-bind pair be structurally equivalent to every other unit-bind pair.

Strength. It is common to also require that monads used to represent programming languages are *strong*. A strong monad is a monad over a category with an associative tensor product $\otimes$, and a transformation

$$st : a \otimes m\ b \to m(a \otimes b) \qquad .$$

The *st* transformation must satisfy a number of laws that preserve the associativity of the tensor product.

Strength allows us to model computations with an environment. We can transform any computation from x to y into a computation from $x \otimes e$ to $y \otimes e$ using strength. Strength can also be thought of as making our monad compatible with $\otimes$ by allowing us to form $f \otimes x$ for morphisms that produce values in the monad. For our small category of types, strength is implicitly provided by the base category. That is, we can define the transformation `st` by making use of the operations on pairs from the base category:

```
st (e,m) = m >>= λx → return (x,e)
```

Because we are using pairs from the base category, our definition of strength automatically satisfies the relevant associativity laws.

Note, that our tensor product is associative, but not commutative. This is a good thing since otherwise we could not distinguish between potentially different computations. For instance, suppose we model state by s, and we have two stateful computations:

$$A : x \otimes s \to x' \otimes s$$

$$B : y \otimes s \to y' \otimes s$$

The composition of these two computations can be represented as a function with type:

$$AB : x \otimes y \otimes s \to x' \otimes y' \otimes s$$

However, there are two possible computations: one in which x occurs first and one in which y occurs first. This shows a subtle difference between the tensor product in the base category and strength of the monad. We will return to this in the next section.

Kleisli Category. Just because we can give a definition of monads doesn't mean there are any meaningful solutions. Happily, there are two solutions to the monad equations. These solutions correspond to the algebras over the functor. That is, the solutions correspond to the induction principle derived from the functor. The more commonly used solution considers only the free algebras, which correspond to our usual notion of algebraic data types in functional languages.

The free algebras also have the nice property that they can be represented as the category of morphisms of type: $a \to m\ a$ where m is a monad. These morphisms, referred to as the Kleisli arrows, are exactly the functions we used to represent errors previously.

For any category C and monad m we can construct the *Kleisli* category of m. For example, assume C is a small category of types and pure functions as we would have with the simply-typed lambda calculus. In the Kleisli category each identity morphism is given by `return` applied the identity morphisms of C, and each morphism in the Kleisli category corresponds to a Kleisli arrow in C. Therefore, the Kleisli category gives us a model for computations with side-effects, non-terminations, etc. Unfortunately, these categories are not flexible enough to represent the semantics of SVG.

4.2 Generalizing Monads

Monads (and the associated Kleisli categories) have been used successfully to model different types of side-effects and non-total functions (Claessen, 1999; Jones, 2002; Nanevski et al., 2006). However, not all types of partial functions can be rep-

resented as monads. Perhaps the first example of the inability to structure programs using monads was noted by Swierstra and Duponcheel (1996) when constructing a parsing library. Other examples include reactive programming (Hudak et al., 2003; Nilsson et al., 2002), user interfaces (Courtney and Elliott, 2001), dynamic optimization involving generalized algebraic data types (Nilsson, 2005), and data-flow or stream processing languages (T. Uustalu, 2005).

Data-flow languages are particular relevant for our purposes since Synchronous Value Graphs can be understood as stream processing constructs that are "wired" together to produce a complete program. Data-flow programming has a long history dating back to the seventies. It was at this time that Kahn discovered the relationship between reactive networks and stream processing languages (Kahn, 1974). At the same time, the programming language Lucid was being developed, and the design included a number of stream processing primitives (Wadge and Ashcroft, 1985).

In the previous section we have seen that a monad helps us represent the internal state of a term. A simple stream processing function also has state. However, this state is not internal to the stream function, but is in the context in which the stream function is placed. The value of a stream function at any time depends on the values of its inputs. Contrast this to a reference cell which depends on the internal state of the reference cell at a given time. The reference cell can be represented with a monad, the stream function requires a co-monad (Uustalu and Vene, 2005).

A co-monad is the dual of a monad, and can be defined by the duals of `return` and `bind`:

```
class CoMonad m where
```

```
coreturn :: m a → a
cobind   :: m a → (m a → b) → m b
```

The `coreturn` and `cobind` functions must satisfy certain laws. If the laws are satisfied, we can use a co-monad to reason about stream functions. For our purposes here, we can dispense with this development and rather use the co-monad concept to gain a little bit of mathematical intuition.

In previous sections we have seen that a monad gives rise to a category of morphisms of the form: $a \rightarrow m\ a$, where m is a monad. Similarly, a co-monad gives rise to a category of morphisms of the form: $\overline{m}\ a \rightarrow a$, where $\overline{m}$ is a co-monad. For monads, we can think of $m\ a$ as containing some internal information, such as the state of the current value of a reference cell. For co-monads, we can think of $\overline{m}$ as containing contextual information, such as the current values of a stream function's inputs. Co-monads can be used as a model for simple stream processing languages or Kahn networks.

One immediate difficulty with Kahn networks (and the Lucid stream processing functions) is that they are difficult to implement efficiently. However, it was realized that certain kinds of networks, networks with a synchronous clock, could be implemented efficiently. The implementation technique involved eliminating intermediate data structures used to store values moving from one process to another. This implementation technique is essentially a form a "list-less" programming (Wadler, 1984), or the more general technique of deforestation (Wadler, 1990) found in many functional language compilers. A number of languages have been developed on the synchronous Kahn network model, including Lustre (Halbwachs et al., 1991) and Esterel (Berry and Gonthier, 1992).

In Synchronous Value Graphs, we also have a synchronous requirement which is informally described by the ϵ values appearing in our definitions: each term must either consume ϵ values or non-ϵ values, but not a mixture of the two. This synchronous requirement can be modeled by considering that each term is influenced by *both* its context and some internal state (Power and Robinson, 1997). When looked at in this way, a synchronous data-flow language is a combination of monads and co-monads. This combination can be realized by morphisms of the form: $\overline{m}\ a \to m\ a$, where $\overline{m}$ is a co-monad, and m is a monad (Uustalu and Vene, 2005).

Giving a semantic model for these morphisms is slightly complex. Remember, monads are derived by establishing a monoidal structure over functors on a category. If our category is C, then our monads are a monoidal structure on the functors from the category C. For co-monads, we can consider functors from the dual category C^{op} (Blute et al., 1997). To combine these two, we can consider functors over the product of C and C^{op}; these are functors of the form:

$$C^{op} \times C \to Set \qquad .$$

In fact, these functors are more general than monad/co-monad combinations[4]. In the next section we will develop a semantic structure analogous to Kleisli categories for functors of this form. This will form the basis of our semantic model for Synchronous Value Graphs.

Another difficulty facing synchronous data-flow languages is the inability to define abstraction, application and recursion within the language. In the data-flow languages discussed so far, each stream processor is a block box. The processors cannot be

[4]For example, they allow for non-deterministic data-flow languages.

abstracted or applied, and it is not possible to use a general fix point operator in their definition. However, more recently, it has been shown that abstraction, application and recursion can be defined in a synchronous language (Caspi and Pouzet, 1996). These developments have been incorporated into modern data-flow languages such as recent versions of Lustre and Lucid Synchrone (Caspi and Pouzet, 2002).

The development of general recursion in data-flow languages, in many ways, parallels the development of general recursion for monads (Erkök, 2002; Erkök and Launchbury, 2002). In the next section we will present our semantic model and show how recursion operators can be defined.

4.3 Arrows

In order to define our semantics, we will start with a base category that models our expression language. Our expression language is a variant of the simply-typed lambda calculus, and can be modeled as a Cartesian closed category (Jung et al., 1988). In such a category, we have closure over products and exponentials. That is, for any two objects x and y in the category, $x \times y$ and $x \to y$ are also in the category. Furthermore, there is a bijection between $x \times y \to z$ and $x \to y \to z$ which allows us to define currying and uncurrying.

Given our base category C, the model for Synchronous Value Graphs is a category of functors of the form

$$C^{op} \times C \to Set \qquad .$$

We can think of C^{op} as representing the context, and C as representing side-effects. Technically, Set is stronger than we need (Power and Thielecke, 1999), but it is fine

for our purposes. Following Paterson (2003) we will call the objects of our category "arrows".

The objects of our category, arrows, are functors. If a is an arrow, then $(a\ x\ y)$ is a set if x and y are objects of the base category and a is contravariant in x and covariant in y. The elements of the set $(a\ x\ y)$ are the computations from x to y. We will provide a transformation for morphisms called arr. If $(f : x \to y)$ is a morphism in the base category from x to y, then $(arr\ f)$ is the set of computations $(a\ x\ y)$. We also provide a way to compose arrows together. The wiring operator, $(\ggg)$ is similar to composition, but the arguments are reversed to give a left-to-right style (arguments come from the left and results go to the right):

$$(\ggg) : a\ x\ y \to a\ y\ z \to a\ x\ z \qquad .$$

Finally, we need to be able to lift products from the base category. For monads, this is done via strength which does not conflate computations which are in different orders. Since our arrows are a generalization of moands, we must allow for the same structure. We do this by providing a half product called $first$ (Power and Robinson, 1997). The half product has type (in our category):

$$first_X : (a \to b) \to a \otimes X \to b \otimes X \qquad .$$

This allows us to build a product from a set of computations by inserting them into the first element of a product. The dual operation *second* can be defined in terms of $first$, and the two together give us a way to build products in our category. Note, we are forced to be explicit about which of the two halves of the product is first in a composition.

We can express our definition of arrows using Haskell-like notation. These definitions follow the structure found in the Haskell Standard Libraries (Paterson, 2003).

```
class Category a ⇒ Arrow (a :: ∗ → ∗ → ∗) where
   arr   :: (b → c) → a b c
   (⋙) :: a b c → a c d → a b d
   (⋙) = λf g. g ∘ f
   first :: a b c → a (b,d) (c,d)
```

Starting from our small category of types and pure functions, an arrow, a, is a constructor of two arguments corresponding to our functor. The `arr` function builds an arrow from a function in the base category. The infix operator, $\ggg$, composes two arrows to form a third. We can give a default implementation of $\ggg$ in terms of categorical composition by reversing the arguments. Finally, the function `first` provides our compatibility with products from the base category.

We require that our arrows form a category. We will derive our identity morphisms using *arr* applied to the identity morphisms from the base category. That is, we define our identity morphisms by:

$$id_{a\ \tau\ \tau} = arr(id_{\tau \to \tau}) \qquad .$$

Then, we must require our composition operator is associative, and that it preserves the structure of the base category:

$$arr\ id \ggg f = f \ggg arr\ id = f \tag{4.1}$$

$$(f \ggg g) \ggg h = f \ggg (g \ggg h) \tag{4.2}$$

$$arr(f \circ g) = arr\ g \ggg arr\ f \tag{4.3}$$

Equations (4.1) and (4.2) ensure that our arrows form a category. Equation (4.3) preserves the structure of the base category with respect to composition.

Although we have defined arrows by fiat, we could just as easily derive equations (4.1)-(4.3) by forming a monoid on our category of functors. To form a monoid, we define the monoidal product of two functors $A, B : C^{op} \times C \to Set$, to be the smallest set containing all functors such that

$$a\ x\ y \times a\ y\ z \xrightarrow{\ggg} a\ x\ z$$

is parametric in y. This monoid requirement is equivalent to equations (4.1)-(4.3). The proof of this fact is beyond the scope of this work. However, a complete account can be found in Borceux (2004).

Our half-product and composition must also be compatible. Recall, in the case of monads this is achieved through a property called "strength." Our arrows have an analogous property called *internal strength* which is established by the following equations (Paterson, 2003; Power and Thielecke, 1999):

$$first(f \ggg g) = first\ f \ggg first\ g \tag{4.4}$$

$$first(arr\ f) = arr(f \times id) \tag{4.5}$$

$$first\ f \ggg arr(id \times g) = arr(id \times g) \ggg first\ f \tag{4.6}$$

$$first\ f \ggg arr\ \pi_1 = arr\ \pi_1 \ggg f \tag{4.7}$$

$$first(first\ f) \ggg arr\ \alpha = arr\ \alpha \ggg first\ f \tag{4.8}$$

$$\text{with} \quad \alpha = \lambda((a, b), c) \to (a, (b, c))$$

Examples. The arrow definition can best be understood through examples. Throughout this section we will use two basic examples: pure functions, and stream functions, both of which are instances of arrows. Pure functions, with constructor $(\to)$, form an arrow. Below is the definition of the arrow instance for pure functions.

```
instance Arrow (→) where
   arr f    =  f
   f ⋙ g    =  g ∘ f
   first f  =  λ(a, b) → (f a, b)
```

It is easy to verify equations (4.1)-(4.8). For example, for equation (4.3) we have:

$$arr(f \ggg g) = g \circ f = arr\ g \ggg arr\ f \qquad .$$

More importantly, our second example of stream functions are also form an arrow. The type of a stream functions is $[a] \to [b]$, where $[a]$ is the type of an infinite stream of values of type a. We will write this type as `(SF a b)`. The type `SF` of stream functions forms an arrow according to the definition below.

```
newtype SF a b = SF ([a] → [b])
instance Arrow SF where
   arr f          = SF (map f)
   SF f ⋙ SF g    = SF (g ∘ f)
   first (SF f)   = SF (λl → let (bs,ds) = unzip l in zip (f bs) ds)
```

Where unzip and zip have types:

```
zip   :: [a] → [b] → [(a,b)]
unzip :: [(a,b)] → ([a],[b])
```

Derived Functions. Having a definition of the arrow class, we can now define a couple of essential combinators in terms of the arrow functions. The function `second` is the dual to `first`, and can be defined as:

```
second :: Arrow a ⇒ a b c → a (d,b) (d,c)
second f = arr swap ⋙ first f ⋙ arr swap
   where swap (x,y) = (y,x)
```

Using `first` and `second` we can define the split combinator. The split combinator, written infix as `(***)`, splits an input pair between two arrows and returns the combined result.

```
(***) :: Arrow a ⇒ a b c → a b' c' → a (b,b') (c,c')
f *** g = first f ⋙ second g
```

Finally, the fan-out function, written infix as `(&&&)`, duplicates its input giving a copy to each of the two argument arrows. The result of the two arrows is combined and returned.

```
(&&&) :: Arrow a ⇒ a b c → a b c' → a b (c,c')
f &&& g = arr (λb → (b,b)) ⋙ (f *** g)
```

4.3.1 Relationship to Monads

As we have noted, arrows are a generalization of monads. Now that we have a concrete definition of arrows, we can make the relationship between the two explicit. In this section we will explore the relationship between monads and arrows and clarify our earlier claim that stream functions do not form a monad.

The relationship between monads and arrows is captured by a single property: the ability to define application. All arrows for which application can be defined form a monad and vice versa. Recall that arrows represent computations and not values. Therefore, in general, the arrow combinators are point-free with respect to values: intermediate values are never named, only computations are named. However, suppose that we did have a named value, then in order to make any use of it, we would have to apply an arrow to it. That is, if we have a value of type α, and an arrow

from α to β, then, to create a monad, we must be able to apply the value to the arrow. This can be seen as a kind of uncurrying, and is captured by the definition `ArrowApply` below.

```
class Arrow a ⇒ ArrowApply a where
   app :: a (a b c, b) c
```

The normal function type supports apply trivially.

```
instance ArrowApply (→) where
   app (f,x) = f x
```

The ability to apply a computation to a value is the key property of a monad. If an arrow supports apply, then each arrow can be put into correspondence with a monadic function (a pure function with a monadic result). These functions are the Kleisli arrows of the monad. Therefore, for any arrow that supports application, we can make the following identification:

$$A\ \alpha\ \beta\ \equiv\ \alpha \rightarrow A\ ()\ \beta\ \equiv\ \alpha \rightarrow M\ \beta$$

where M is a monad. This also corresponds to the notion that a monad is a special case of our class of functors in which the contra-variant argument is not required. Using this identification, we can now define a monad for any arrow that supports application.

```
newtype ArrowApply a ⇒ ArrowMonad a b = ArrowMonad (a () b)
instance ArrowApply a ⇒ Monad (ArrowMonad a) where
   return x            = ArrowMonad (arr (λz → x))
   ArrowMonad m >>= f  = ArrowMonad (m ⋙ arr f' ⋙ app)
       where f' x = let ArrowMonad h = f x in (h, ())
```

Although the above definition of monad is sufficient, is it illuminating to see a more direct definition of the arrow instance associated with a monad. We can provide a more direct definition that is directly related to the computation functions (or Kleisli arrows) of the monad. For any monad, we can define the set of Kleisli arrows associated with that monad; these are all of the functions with co-domain in the monad. The set of Kleisli arrows for a given monad form an arrow according to the definition below.

```
newtype Kleisli m a b = Kleisli (a → m b)
instance Monad m ⇒ Arrow (Kleisli m) where
   arr f                    = Kleisli (return ∘ f)
   Kleisli f ⋙ Kleisli g    = Kleisli (λb → f b >>= g)
   first (Kleisli f)        = Kleisli (λ(b,d) → do c ← f b
                                                   return (c,d))
```

Of course, all arrows formed from monads also support application.

```
instance Monad m ⇒ ArrowApply (Kleisli m) where
   app = Kleisli (λ(Kleisli f, x) → f x)
```

The ability to define application is a necessary and sufficient condition for an arrow to form a monad. This ability is closely related to the fact that the contra-variant argument of the functor plays no part: the context of the computation is not relevant. Note, that there is no reasonable way of defining `app` for stream functions.

Like stream functions, Synchronous Value Graphs do not support application. Therefore, we will use arrows without application to structure our language. Even though we lack application, there are other properties Synchronous Value Graphs do have. We will look at these properties now.

4.3.2 Choice

Even though many arrows do not support application (and thus are not monads), more arrows do support choice. Choice is analogous to the `first` and `second` functions defined in the `Arrow` class. Just as the `first` and `second` functions allow us to define arrows over a product type, the choice functions, `left` and `right`, allow us to define arrows over a sum type. Similar to products, a complete definition of choice only requires the function `left`.

```
class Arrow a ⇒ ArrowChoice a where
   left :: a b c → a (Either b d) (Either c d)
```

The function `left` must satisfy the following laws:

$$left(arr\ f) = arr(f \oplus id) \tag{4.9}$$

$$left(f \ggg g) = left\ f \ggg left\ g \tag{4.10}$$

$$left\ f \ggg arr(id \oplus g) = arr(id \oplus g) \ggg left\ f \tag{4.11}$$

$$arr(Left) \ggg left\ f = f \ggg arr\ Left \tag{4.12}$$

$$left(left\ f) \ggg arraccossum = arr\ assocsum \ggg left\ f \tag{4.13}$$

where

```
assocsum :: Either (Either a b) c → Either a (Either b c)
assocsum (Left (Left a))  = Left a
assocsum (Left (Right a)) = Right (Left a)
assocsum (Right a)        = Right (Right a)
```

Our definition of `left` and the associated laws make use of a sum type, `Either`, from the base category. The function `right` can be defined in terms of `left`, also relying on the base category.

```
right :: ArrowChoice a ⇒ a b c → a (Either d b) (Either d c)
right f = arr mirror ⋙ left f ⋙ arr mirror
   where mirror (Left x) = Right x
         mirror (Right y) = Left y
```

We can construct two operators for building arrows over sum types. The merge function, `(+++)`, combines two arrows to produce a new arrow over their combined argument and result types.

```
(+++) :: ArrowChoice a ⇒ a b c → a b' c' → a (Either b b') (Either c c')
f +++ g = left f ⋙ right g
```

The `(|||)` operator combines two arrows with the same result type into a new arrow with a argument sum type.

```
(|||) :: ArrowChoice a ⇒ a b d → a c d → a (Either b c) d
f ||| g = f +++ g ⋙ arr untag
   where untag (Left x) = x
         untag (Right y) = y
```

These combinators can be used to "wire-up" networks of arrows.

Choice Examples. Both normal functions and Kleisli arrows support choice with the following implementations.

```
instance ArrowChoice (→) where
   left f = either (Left ∘ f) (Right ∘ id)

instance Monad m ⇒ ArrowChoice (Kleisli m) where
   left m = let Kleisli f = m ⋙ arr Left
                Kleisli g = arr Right
            in Kleisli (either f g)
```

More importantly, stream functions also support choice.

```
instance ArrowChoice SF where
  left (SF f) = SF ((arr getLeft ⋙ f) &&& arr id ⋙ arr replace)
    where getLeft (Left x : xs) = x : getLeft xs
          getLeft (Right _: xs) = getLeft xs
          replace (x:xs, Left  _:ys) = Left x : replace (xs,ys)
          replace (xs,   Right y:ys) = Right y: replace (xs,ys)
```

This last definition of choice for stream function shows the use of the combinators defined in the section. Hopefully, these definitions also give some intuition as to how complex programs can be constructed.

4.3.3 Loops

The last extension to arrows that we will look at is the ability to define a form of recursion on arrows. Our recursion operator is called `loop`, and can be thought of as a form of value recursion. Arrows that support `loop` are members of the `ArrowLoop` class:

```
class Arrow a ⇒ ArrowLoop a where
   loop :: a (b,d) (c,d) → a b c
```

The second component of the arrow result (d) is fed back into the second component of the argument forming a loop. The `loop` function must satisfy a number of algebraic

laws which are shown below.

$$loop(arr\ f) = arr(trace\ f) \tag{4.14}$$

$$loop(first\ h \ggg f) = h \ggg loop\ f \tag{4.15}$$

$$loop(f \ggg first\ h) = loop\ f \ggg h \tag{4.16}$$

$$loop(f \ggg arr(id \times k)) = loop(arr(id \times k) \ggg f) \tag{4.17}$$

$$loop(loop\ f) = loop(arr\ unassoc \ggg f \ggg arr\ assoc) \tag{4.18}$$

$$second(loop\ f) = loop(arr\ assoc \ggg second\ f \ggg arr\ unassoc) \tag{4.19}$$

where

```
trace :: ((b,d) → (g,d)) → b → g
trace f b = let (c,d) = f (b,d) in c
```

We will look closely at these laws in Chapter 5 when we present a mechanized semantics for Synchronous Value Graphs.

Loop Examples. For normal functions, the definition of `loop` relies on value recursion at the term level. In a language like Haskell with lazy evaluation, we can define `loop` for function as:

```
instance ArrowLoop (→) where
   loop f b = let (c,d) = f (b,d) in c
```

Notice, the second element of the result of `f`, the variable `d`, is also an argument to the function `f`.

The `loop` function for arrows is similar to the `mfix` function that can be defined for some monads (Erkök, 2002; Erkök and Launchbury, 2002). In fact, any monad that supports `mfix` has an associated arrow that supports `loop`.

```
instance MonadFix m ⇒ ArrowLoop (Kleisli m) where
   loop (Kleisli f) = Kleisli (liftM fst ∘ mfix ∘ f')
       where f' x y = f (x, snd y)
```

In this case, the `mfix` operation is providing the equivalent of value-recursion within the monad.

Stream functions also support `loop`; again, the definition relies on value recursion at the term level.

```
instance ArrowLoop SF where
   loop (SF f) = SF (λbs → let ds = snd (unzip cd)
                               cd = f (zip bs ds)
                           in fst (unzip cd))
```

4.3.4 Denotational Model

In the previous sections we presented a general semantic framework that is capable of modeling our intermediate representation. We have presented the framework with an algebraic specification. This specification has been shown to correspond to traced premonoidal categories (Power and Thielecke, 1999), which gives us a rich language for describing our terms. However, we must now give a specific instance of this framework and show it satisfies the algebraic specification.

In the remainder of this chapter we will give our denotational model. First, we will provide some intuition by looking at an operational interpretation for synchronous data-flow languages. From this operational interpretation we can construct a mathematical model which mirrors our intuitive understanding. We will then show this mathematical model forms one of the categories described in the previous sections.

Data-flow Semantics

To motivate our categorical semantics, we will first look at how synchronous data-flow languages are modeled operationally. Recall from section 4.0.3 that our informal definition of Synchronous Value Graphs was specified as stream processing primitives. The informal definition of `map` is repeated below.

$$\texttt{map}\ f\ (\epsilon : xs) = \epsilon : \texttt{map}\ f\ xs$$

$$\texttt{map}\ f\ (x : xs) = f\ x : \texttt{map}\ f\ xs$$

This style of semantics has been formalized by Caspi and Pouzet (1996) and others. To formalize this semantics, we give a judgement of the form: $t \xrightarrow{v} t'$. This is read as t steps to t' and produces the value v on its output stream. The operational rules for `map` are:

$$\frac{x \xrightarrow{\epsilon} x'}{\texttt{map}\ f\ x \xrightarrow{\epsilon} \texttt{map}\ f\ x'}\ (map_0) \qquad \frac{x \xrightarrow{v} x' \qquad f\ v \rightarrow^* w}{\texttt{map}\ f\ x \xrightarrow{w} \texttt{map}\ f\ x'}\ (map_n)$$

The map_n rule says that `map` f x steps to `map` f x' and produces the value w if x steps to x' and produces the value v, and f v evaluates to w in some number of steps. This last antecedent uses the operational semantics of our simply-typed expression language. This demonstrates several essential features of this style of semantics. First, the semantics is defined in terms of the semantics for expressions. Second, the terms evolve as they consume inputs. This last feature is more apparent in the case of η. Recall the informal specification of η has as one of its rules

$$\eta(\textbf{false} : cs, x : xs) = \texttt{const}\ x$$

with the corresponding operational rule:

$$\frac{(\eta_f)}{\dfrac{x \xrightarrow{\texttt{false}} x' \qquad c \xrightarrow{w} c'}{\eta(x,c) \xrightarrow{w} \texttt{const}\, w}} \quad .$$

We can see that an η term transforms into a `const` term when the input becomes `false`. Finally, the terms *only* evolve and produce output when inputs are available, and only one input is processed at a time. A complete operational semantics for Synchronous Value Graphs can be found in Appendix A.

Model

Our model is designed after the key features of dataflow semantics. Whenever inputs are ready, the term can evolve into another term according to the semantics of the expression language. The terms can only evolve after an input is read, otherwise the term is stalled waiting for an input. Finally, each term can only read an input after it has evolved to a new term. Specifically, we model a Synchronous Value Graphs as being in one of two states: waiting for input, or producing output. These two states are represented by the data type shown below.

```
data Term a b
   = In (a → Term a b)
   | Out b (Term a b)
```

When a term is waiting for input, it is represented as `In f` where f is a function in our expressions language that when applied to the eventual input will produce a new term. When a term is producing output, it is represented by `Out x t` where x is the output, and t is a new term that will replace the current one when the output is

consumed.

The type parameters a and b, the functions on input terms, and the x in the output term are all objects in our base category: i.e. expressions and types of expressions. We enrich the category via the two-place functor Term. We will now show that the Term functor is an arrow by defining all of the necessary transformations. Formal proofs of the algebraic laws are given in Chapter 5.

Terms form a category. The first definition shows how our SVG terms form a category. We must provide an identity element for each object in the category, and a composition operator. The identity elements are all constructed from a single polymorphic term called `id`. An identity element takes an input, `x` and transforms to an output term which produces `x`. After output, the identity term becomes the identity term again.

```
instance Category Term where
  id    = In (λx → Out x id)
  f ∘ g = case f of
            Out x f' → Out x (f' ∘ g)
            In f'    → case g of
                         Out y g' → f' y ∘ g'
                         In g'    → In (λx → f ∘ g' x)
```

Composition is a bit more involved. To compose SVG terms, we look at the first term to see if it is an input or an output term. If it is an output term, then we can use the base category composition operator to construct a composed output term. Otherwise, if we have an input term, then we have to also look at the form of the second term. An input composed with an output is the composition of the applied input function to the output value with the output term. An input composed with

another input is an input term with the two input functions composed with the base category composition operator.

Terms are arrows. Our SVG terms also form an arrow. The arrow built from definitions of `arr` and `first`. The `arr` function is similar to the category identity function, but we apply a base category function to each value in the output sequence.

```
instance Arrow Term where
  arr   f = In (λx → Out (f x) (arr f))
  first f = bypass epsilon f

epsilon = Nothing
bypass Nothing  (In f)     = In (λ(b,d) → bypass (Just d) (f b))
bypass (Just d) (Out c sp) = Out (c,d) (bypass epsilon sp)
bypass Nothing  sp         = In (λ(b,d) → bypass (Just d) sp)
```

To implement `first`, we transform the argument term using the function `bypass` with an initial first argument of `epsilon`. The function of `epsilon` is to wrap the argument term in a term which inputs/outputs pairs of values. The first value of the pair is fed into the argument term and the second value of the pair is passed through the term.

Terms support choice. To support choice, we must give an implementation of `left` which operates similar to `first`.

```
instance ArrowChoice Term where
  left (Out x f) = Out (Left x) (left f)
  left (In f)    = In (λx → case x of
                              Left  a → left (f a)
                              Right a → Out (Right a) (left (In f)))
```

For an output term, `left` simply wraps the output values in the base category term `Left`. If we have an input term, then we have to check the value. If the value is a `Left`, then we apply our function. Otherwise, we skip the `Right` value letting it pass through the term.

Terms support recursion. Finally, we need our Synchronous Value Graphs to support recursion. The `loop` function defines the meaning of recursion in our graphs.

```
instance ArrowLoop Term where
  loop = loop' epsilon

loop' :: Maybe c → Term (a,c) (b,c) → Term a b
loop' x        (Out (a,b) f) = Out a (loop' (Just b) f)
loop' (Just x) (In f)        = In (λb → loop' Nothing (f (b,x)))
```

The operation of `loop` can be thought of as connecting wires between the input and output terms. The looping term has an "internal" wire which is connected back to itself. This extra wire is carried as one component of a pair which is not part of the term's visible type. For an output term, we copy the internal wires value to our sub-terms. For an input term, we apply the base category function to the normal input and our internal value. Having used the internal value, we do not pass it along to the sub-terms.

At this point we have a semantics framework, and a concrete instance of the framework we can use to model our language. In the next chapter we will formalize the model further by providing mechanized proofs of the categorical properties. In Chapter 7 we will look at how these algebraic rules are used in practice.

Chapter 5

Mechanized Semantics and Rewrite Rules

In this chapter we present a mechanized version of the semantics described in Chapter 4. We have formalized our semantics in the theorem prover Coq. The formal definition of the model and the categorical properties justify the rewrite rules based on those laws. Also, this gives us a framework for formally proving additional rewrite laws.

5.1 SVG Category

The first step in mechanizing our definition of Synchronous Value Graphs, is defining our basic model and showing it forms a category. Recall, our model is based on the operational semantics of data-flow languages, and consists of terms either waiting for input or producing output. An informal definition of these term is given in Section 4.3.4.

In our model, each term represents an infinite stream of values. The output of one term can be "wired" up to the input of another forming larger terms. In order to formalize these definitions, we have to be able to define infinite streams. Our proof development system, Coq, provides a mechanism for defining infinite object through co-inductive definitions (Coq development team, 1989–2008). The co-inductive definition of our terms is shown below.

```
CoInductive Term (a b : Set) :=
| In   : (a -> Term a b) -> Term a b
| Out  : b -> Term a b -> Term a b
| Wrap : Term a b -> Term a b.

Implicit Arguments In [a b].
Implicit Arguments Out [a b].
Implicit Arguments Wrap [a b].
```

There are several things to note about this definition. First, each term is parametrized by two values of type `Set`. These values represent the core-language types for the input and output of the term. We see the familiar `In` and `Out` terms. `In` takes a core-language function from the input type to a new term, and `Out` takes an output value and a term.

Notice that as our "core language" we are using the built-in function space of Coq itself. This core language is strictly more expressive than our core language of simply types lambda terms. However, using Coq's built-in function makes proofs easier to develop and does not create any difficulties. In fact, we are free to change our model to use Coq's language as our core language if we like, however equality would be more complex.

Finally, we have a third term constructor `Wrap` that we did not present in the informal semantics. This term is used to build co-inductive definitions, but has no effect on the semantics. Because Coq relies on the Curry-Howard correspondence (Curry, 1934), we must show that our co-inductive definitions make progress. The easiest way to do this is to make sure all recursive calls are under a constructor. The `Wrap` constructor is used in cases where our recursive calls would not otherwise be under an `In` or `Out` constructor. We will see an example of this when we define composition below.

Forming a category. To show that our terms form a category, we must define the identity morphisms, and an associative composition operator. Both of these definitions follow exactly those given in Section 4.3.4. The identity morphisms are defined as the set of terms which can be instantiated from the definition `ident` below.

```
CoFixpoint ident {a : Set} : Term a a :=
  In (fun x => Out x ident).
```

Note that, like our terms, `ident` is a co-inductive definition: it forms an infinite term due to the recursion under the `Out` constructor. Generally, all of the definitions which

manipulate terms will be co-inductive. The definition of composition is shown below.

```
CoFixpoint comp {a b c:Set} (f:Term b c) (g:Term a b) : Term a c :=
match f with
| Out x f' => Out x (comp f' g)
| In f' =>
   match g with
   | In g'    => In (fun x => comp f (g' x))
   | Out y g' => Wrap (comp (f' y) g')
   | Wrap g'  => Wrap (comp f g')
   end
| Wrap p => Wrap (comp p g)
end.
```

We have already seen composition defined in the informal semantics. This definition has an additional case for the `Wrap` constructor, and we see the need for `Wrap` in the case of composing an input with an output. When composing an input with an output, we call `comp` recursively with `(f' y)` and `g'`. However, in order to establish that our co-inductive definition is making progress we enclose this call in `Wrap`.

Equivalence The primary purpose of mechanizing our semantics is to prove that our rewrite laws are sound. This involves proving that terms are equivalent. We say two terms are equivalent if they produce the same infinite stream of values when given the same infinite stream of inputs. Often this notion of equivalence is called *bisimilarity.*

We define equivalence using a set of proof rules. Hence, two terms are equivalent if and only if we can prove so using our rules. The rules for deciding equivalence are

shown below.

In	Out	WrapL	WrapR
$\dfrac{\forall x, f\ x \sim g\ x}{\texttt{In}\ f \sim \texttt{In}\ g}$	$\dfrac{x \equiv y \qquad t \sim u}{\texttt{Out}\ x\ t \sim \texttt{Out}\ y\ u}$	$\dfrac{t \sim u}{\texttt{Wrap}\ t \sim u}$	$\dfrac{t \sim u}{t \sim \texttt{Wrap}\ u}$

There are four rules: one for input term, one for output terms, and two rules for our `Wrap` constructor. The first rule says that two input terms are equivalent if their functions always produce equivalent terms when applied to the same input. The second rule says that two output terms are equivalent if they output the same value, and continue to equivalent terms. Note that we are using the core-language equivalence for values ($\equiv$). The third and fourth rules say that if we encounter a `Wrap` term we can discard it. It is in exactly this sense that we mean the `Wrap` terms do not have any semantic meaning.

Our equivalence proof rules can be easily encoded using a co-inductive relation on terms. We use the name `bisim` for the relation, which is shown below.

```
CoInductive bisim {a b : Set} : Term a b -> Term a b -> Prop :=
| biIn    : forall f g,
            (forall x, bisim (f x) (g x)) ->
            bisim (In f) (In g)
| biOut   : forall x y t u,
            x = y -> bisim t u ->
            bisim (Out x t) (Out y u)
| biWrapL : forall t u,
            bisim t u ->
            bisim (Wrap t) u
| biWrapR : forall t u,
            bisim t u ->
            bisim t (Wrap u).

Notation "a ~ b" := (bisim a b) (at level 80).
```

Each of the four proof rules is represented by a constructor in our definition. Reading from top to bottom we can see the antecedents and consequent of each rule. For convenience, we have defined a notation which allows us to write $a \sim b$ for `(bisim a b)` in the remainder of the development.

5.1.1 Co-inductive Proof Techniques

One of the difficulties of working with co-inductive definitions, is that we lose the basic proof technique of *decomposition*. Normally, with finite terms, we can reason about properties by looking at the shape of a term. We can replace a term by its definition and reason case by case on the different possibilities. With infinite terms, this is not viable in general because the process of unrolling the definition may go on forever.

Nevertheless, we can still achieve a limited form of decomposition by proving special cases. We will do this using a helper function. The helper function is the identity function specialized to our `Term` type.

```
Definition Term_decompose {a b:Set} (t:Term a b) : Term a b :=
match t with
| In f => In f
| Out x f => Out x f
| Wrap p => Wrap p
end.
```

At first this function does not seem very interesting. However, note that `Term_decompose` is not co-inductive; in fact, it is not even recursive. Using `Term_decompose` we can prove a small lemma which says that every term `t` is equal to the application of `Term_decompose` to itself.

```
Lemma Term_decompose_lemma :
  forall a b (t : Term a b),
    t = Term_decompose t.
Proof.
  intros a b t; case t; auto.
Qed.
Implicit Arguments Term_decompose_lemma [a b].
```

The proof of this lemma is a straight-forward case analysis on the term. Because `Term_decompose` is not recursive, after we perform case analysis on `t` we can inline `Term_decompose t` and easily show the (intensional) equality.

The lemma just proved gives us a limited form of decomposition. For convenience, we will define a small proof tactic which tries to apply this lemma called `dc`.

```
Ltac dc := intros; apply Term_decompose_lemma.
```

To use this lemma, we simply apply it in any context in which we need to prove that two identical terms are equal. The lemma allows the proof system to unroll the definitions one time to search for equalities. For example, suppose we want to prove that `ident` is equal to its definition. We can do this simply by applying our lemma.

```
Lemma unfold_ident :
  forall a,
    @ident a = (In (fun x => Out x ident)).
Proof.
 dc.
Defined.
```

The same procedure we carried out for `ident` can be carried out for any function on `Term`s. In order to prove our category properties hold, we will need to prove

similar decomposition lemmas for `comp`. These lemmas are shown in Figure 5.1. We have one lemma for each of the possible result shapes the function can produce. The lemmas are organized into a sections which define the common input parameters for the lemmas. The `cOut` section contains the lemma needed to handle cases which result from the first input parameter being an output term. Similar constructions are defined in sections `cWrap` and `cIn`.

The hints given in Figure 5.1 allow us to use these lemmas to automatically rewrite terms when needed. The hints are put into two groups. The first group `TermF` allows us to unroll `comp` using its definition one time. The second group `TermB` allows us to replace a term with an equivalent call to `comp`.

To see how we can use the rewrite rules, consider the tactics below

```
Ltac biSearch :=
 (apply biIn ||
  apply biOut ||
  (apply biWrapL; try apply biWrapR) ||
  apply biWrapR);
 intros; auto.

Ltac rewrite_bisim :=
  repeat (autorewrite with TermF; biSearch);
  autorewrite with TermB; auto.

Ltac rewrite_ident :=
  rewrite unfold_ident;
  repeat (autorewrite with TermF; biSearch);
  rewrite <- unfold_ident; auto.
```

The first tactic `biSearch` tries to apply one of our equivalence proof rules, and if successful calls the `auto` tactic to reduce the proof further. The second tactic,

```
Section rewriteLemmas.
  Variables a b c : Set.

Section cOut.
  Variable x : c.
  Variable f : Term b c.
  Variable g : Term a b.

  Lemma cOut :
  comp (Out x f) g = Out x (comp f g). dc. Defined.
End cOut.

Section cWrap.
  Variable f : Term b c.
  Variable g : Term a b.

  Lemma cWrap :
  comp (Wrap f) g = Wrap (comp f g). dc. Defined.
End cWrap.

Section cIn.
  Variable f : b -> Term b c.

  Lemma cInIn : forall (g : a -> Term a b),
  comp (In f) (In g) = In (fun x => comp (In f) (g x)). dc. Defined.

  Lemma cInOut : forall (y:b) (g : Term a b),
  comp (In f) (Out y g) = Wrap (comp (f y) g). dc. Defined.

  Lemma cInWrap : forall (g : Term a b),
  comp (In f) (Wrap g) = Wrap (comp (In f) g). dc. Defined.
End cIn.
End rewriteLemmas.

Hint Rewrite cOut cWrap : TermF.
Hint Rewrite <- cOut cWrap : TermB.
Hint Rewrite  cInIn cInOut cInWrap : TermF.
Hint Rewrite <-  cInIn cInOut cInWrap : TermB.
```

Figure 5.1: Rewrite lemmas corresponding to composition.

`rewrite_bisim`, uses our decomposition lemmas. First, the tactic applies the unrolling lemmas as many times as possible. Then, it tries to apply one of our equality proof rules. If successful, then it will try to undo any left-over unrolling and reduce the proof with `auto`. The third tactic is similar to the second except it uses `unfold_ident`. As we will see in the next section, this simple proof strategy is powerful enough to handle many of our proofs.

5.1.2 Category Lemmas

The first two lemmas we have to prove show that our identity element is both a left and right unit for composition. Specifically, for the left unit we have:

```
Lemma left_unit :
  forall a b (f : Term a b),
    bisim (comp ident f) f.
Proof.
 cofix; intros.
 case f; intros;
   rewrite_ident.
Qed.
```

The proof of `left_unit` is a case analysis on the term, f, followed by our decomposition and search tactic `rewrite_ident`. The statement and proof for the right unit is similar: we omit the proof.

```
Lemma right_unit :
  forall a b (f : Term a b),
    bisim (comp f ident) f.
```

The last lemma shows that our composition function is associative. The statement and proof are shown below.

```
Lemma assoc :
  forall (a b c d:Set) (f: Term a b) (g:Term c a) (h : Term d c),
    bisim (comp f (comp g h)) (comp (comp f g) h).
Proof.
  cofix; intros;
  case f; case g; case h; intros;
    rewrite_bisim.
Qed.
```

The proof of `assoc` performs case analysis on each of the input functions. Then, our decomposition tactic is able to complete all 27 sub-goals automatically.

5.2 SVG Arrow

To go from category to arrow, we must define the lifting function `arr`, the wiring combinator `(>>>)`, and the `first` function for creating streams of pairs. The first two definitions are shown below.

```
CoFixpoint arr {a b:Set} (f : a -> b) : Term a b :=
  In (fun x => Out (f x) (arr f)).

Notation "a >>> b" := (comp b a) (at level 60).
```

The `arr` function is the same as we saw in Section 4.3.4. The wiring combinator is defined as a notation for the composition function with the arguments reversed.

Following the informal semantics, we define `first` using a helper function `bypass`. The `bypass` function takes a buffered term which is used to provide the output value when needed. The initial value of the buffered term (`Nothing`) is supplied by `first`.

```
CoFixpoint bypass (a b c:Set)
  (v:Maybe c) (t:Term a b) : Term (a*c) (b*c) :=
match v with
| Nothing =>
   match t with
   | In f => In (fun x:a*c => let (xa,xc) := x in
                              bypass a b c (Just c xc) (f xa))
   | sp   => In (fun x:a*c => let (xa,xc) := x in
                              bypass a b c (Just c xc) sp)
   end
| Just d  =>
   match t with
   | Out xc sp => Out (xc,d) (bypass a b c (Nothing c) sp)
   | sp        => In (fun x:a*c => let (xa,xc) := x in
                                   bypass a b c (Just c xc) sp)
   end
end.

Definition first {a b c:Set} (f: Term a b) : Term (a*c) (b*c) :=
  bypass a b c (Nothing c) f.
```

The arrow function must satisfy the properties given in Equations 4.4-4.8. These properties are formalized in Figure 5.2. The formalization of Equations 4.4-4.8 use three helper function. The functions `cross_id` and `id_cross` encode the operations $f \times id$ and $id \times f$ respectively. The `alpha` function represents α from Equation 4.8.

The proof techniques used for the arrow lemmas are similar to those used to prove the category properties. First, we must define and prove a number of decomposition lemmas related to `first`. Then we add these decomposition lemmas to our rewrite

```
Definition cross_id {a b:Set} (c:Set) (f:a->b) : (a*c) -> (b*c):=
  fun p => let (x,y) := p in (f x, y).

Definition id_cross {a b:Set} (c:Set) (f:a->b) : (c*a) -> (c*b):=
  fun p => let (x,y) := p in (x, f y).

Definition alpha {a b c:Set} : ((a*b)*c) -> (a*(b*c)) :=
  fun p => let (xy,z) := p in
           let (x,y)  := xy in
           (x,(y,z)).

Section FirstLemmas.
  Variable a b c : Set.
  Variable f : Term a b.
  Variable g : Term b c.

  Lemma first1 :
    @first a c b (f >>> g) ~ (first f >>> first g).

  Lemma first2 : forall (f:a->b),
    first (arr f) ~ arr (cross_id c f).

  Lemma first3 : forall (g:b->c),
    first f >>> arr (id_cross _ g) = arr (id_cross _ g) >>> first f.

  Lemma first4:
    first f >>> arr (@fst b c) ~ arr (@fst a c) >>> f.

  Lemma first5:
    first (first f) >>> arr (@alpha b a c) ~
      arr (@alpha a a c) >>> first f.
End FirstLemmas.
```

Figure 5.2: Lemmas for `first` properties.

database and prove the lemmas using the same basic pattern as used for the category lemmas.

5.2.1 Choice

Choice is implemented similarly to the `first` (and `second`) function(s). Just as `first` allows us to define arrows over a product type, the choice function, `left` allows us to define arrows over a sum type. The definition of `left` is shown below.

```
CoFixpoint left {a b c : Set} (t : Term a b) : Term (a+c) (b+c) :=
match t with
| Wrap u  => Wrap (left u)
| Out x u => Out (inl _ x) (left u)
| In f    => In (fun x => match x with
                          | inl y => left (f y)
                          | inr y => Out (inr _ y) (left (In f))
                          end)
end.
```

Our implementation of `left` must satisfy Equations 4.9-4.13. These properties are formalized in Figure 5.3. The formalization of Equations 4.9-4.13 use three helper function. The functions `plus_id` and `id_plus` encode the operations $f \oplus id$ and $id \oplus f$ respectively. The `assocsum` function represents the same function appearing in Equation 4.13.

Again, the proof techniques used for the above lemmas are similar to those used to prove the category properties. First, we must define and prove a number of decomposition lemmas related to `left`. Then we add these decomposition lemmas to our rewrite database and prove the lemmas using the same basic pattern as used for the category lemmas.

```
Definition plus_id {a b:Set} (c:Set) (f:a->b) : a+c -> b+c :=
  fun e => match e with
           | inl x => inl c (f x)
           | inr y => inr b y
           end.

Definition id_plus {a b c:Set} (f:a->b) : c+a -> c+b :=
  fun e => match e with
          | inl x => inl b x
          | inr y => inr c (f y)
          end.

Definition assocsum {a b c:Set} (x : (a + b) + c) : a + (b + c) :=
match x with
| inl s => match s with
           | inl x => inl (b+c) x
           | inr x => inr a (inl c x)
           end
| inr x => inr a (inr b x)
end.

Section LeftLemmas.
  Variable a b c : Set.
  Variable f : Term a b.
  Variable g : Term b c.

  Lemma left1 : forall (f : a -> b),
    left (arr f) ~ arr (plus_id c f).

  Lemma left2 :
    @left a c b (f >>> g) ~ left f >>> left g.

  Lemma left3 : forall (g : a -> b),
    left f >>> arr (id_plus g) ~ arr (id_plus g) >>> left f.

  Lemma left4 :
    arr (inl c) >>> left f ~ f >>> arr (inl c).

  Lemma left5 :
    left (left f) >>> arr (@assocsum b c a) ~ arr assocsum >>> left f.
End LeftLemmas.
```

Figure 5.3: Lemmas for `left` properties.

5.2.2 Loops

The last aspect of the arrow category we will formalize is loops. Cyclic graphs are constructed with the `loop` function which is shown below. Just as in our informal semantics, `loop` is defined using helper function `loop'`.

```
CoFixpoint loop'
  (a b c:Set) (mc:Maybe c) (t:Term (a*c) (b*c)) : Term a b :=
match mc with
| Just x  =>
  match t with
  | In f    => In (fun av => loop' a b c (Nothing c) (f (av,x)))
  | Out v f => Out (fst v) (loop' a b c mc f)
  | Wrap t  => Wrap (loop' a b c mc t)
  end
| Nothing =>
  match t with
  | In f    => In (fun av => loop' a b c (Nothing c) (In f))
  | Out v f => let (x,y) := v in
               Out x (loop' a b c (Just c y) f)
  | Wrap t  => Wrap (loop' a b c mc t)
  end
end.

Definition loop {a b c: Set} := loop' a b c (Nothing c).
```

The loop properties are were given in Equations 4.14-4.19. The formalization of these properties is shown in Figure 5.4.

```
Variable trace : forall (a b c: Set) (f: a*c -> b*c), a -> b.
Implicit Arguments trace[a b c].

Definition assocL {a b c:Set} : ((a*b)*c) -> (a*(b*c)) :=
  fun p1 => let (xy,z) := p1 in
            let (x,y)  := xy in
            (x,(y,z)).

Definition assocR {a b c:Set} : (a*(b*c)) -> ((a*b)*c) :=
  fun p1 => let (x,yz) := p1 in
            let (y,z)  := yz in
            ((x,y),z).

Definition swap {a b:Set} (p:a*b) : b*a := let (x,y) := p in (y,x).

Variable trace : forall (a b c: Set) (f: a*c -> b*c), a -> b.
Implicit Arguments trace[a b c].

Definition assocL {a b c:Set} : ((a*b)*c) -> (a*(b*c)) :=
  fun p1 => let (xy,z) := p1 in
            let (x,y)  := xy in
            (x,(y,z)).

Definition assocR {a b c:Set} : (a*(b*c)) -> ((a*b)*c) :=
  fun p1 => let (x,yz) := p1 in
            let (y,z)  := yz in
            ((x,y),z).

Definition swap {a b:Set} (p:a*b) : b*a := let (x,y) := p in (y,x).
```

Figure 5.4: Lemmas for `left` properties.

Chapter 6

Implementation

In previous chapters we have described our intermediate representation of Synchronous Value Graphs (SVG) and shown how it can be used to validate transformations. In this chapter we will describe the main compilation algorithms we use to compute SVG from the input assembly language. The intermediate representation is computed in two steps: first programs are converted to Gated SSA form. Then, the Gated SSA form is evaluated using our denotational model to produce SVG as described in Chapter 3. The core of the compilation process is the Gated SSA transformation, and much of this chapter is devoted to developing an efficient algorithm for computing Gated SSA for arbitrary input programs.

6.1 Introduction

In Chapter 3 we described the semantics of our input assembly language, and showed how we can elaborate this language using monadic macros into a simple functional language. In Chapter 4 we presented our intermediate language, Synchronous Value Graphs (SVG), which contains this simple functional language as a subset. In those chapters we hinted at how a full transformation into SVG is done, but we did not fully describe how to transform the control-flow of the functional language into the constructs of SVG. In this chapter we will describe this missing and crucial piece of the compilation process.

The compilation process starts by computing Gated Single Static Assignment Form (GSA) (Ottenstein et al., 1990a), from the input assembly language. The GSA form is then elaborated as described in Chapter 3 into SVG. After elaboration, we have SVG terms where the additional GSA constructs (ϕ, η, and σ) are uninterpreted constants. From this point it is trivial to convert the elaborated GSA constructs into SVG constructs to arrive at our final intermediate form. Hence, it is the transformation to GSA form which is ultimately responsible for transforming the control-flow of the functional language into the constructs of SVG. It is this transformation to GSA that we describe now.

6.1.1 Gated SSA Form

Gated Single Static Assignment Form is a subset of Synchronous Value Graphs and is an extension of Single Static Assignment Form (SSA) (Cytron et al., 1991). Recall, in SSA form each variable has exactly one definition. Therefore, in SSA

form, use-def chains are explicit and each chain contains exactly one element. This simplifies many compiler optimizations, and is the original motivation for SSA form. For our purposes, SSA form is closer to a denotational language because, like in mathematics, each variable has exactly one definition. GSA form extends SSA form by adding *gating functions* which capture information about the control-flow of the program. The gating functions are placed within the GSA terms ϕ, η, and σ, which correspond to the ϕ, η, and σ nodes from SVG. Because the control-flow is also accounted for in the terms, GSA is even better suited as a denotational language since definitions are referentially transparent with respect to the control flow.

The syntax of GSA programs is an extension of our assembly language syntax from Chapter 3. GSA extends the language of assembly instructions with terms for carrying gating functions. The new instruction syntax is shown in Figure 6.1. The

i	::=	**getelemptr** $t\ v\ \overline{v}$	Pointer Arithmetic
	|	**alloc** $t\ v$	Stack Allocation
	|	**load** $t\ v$	Load from Memory
	|	**store** $t\ v\ v$	Store to Memory
	|	**binop** $t\ v\ v$	Binary Operator
	|	**conv** $t\ v\ t$	Type Conversion
	|	**select** $t\ v\ v\ v$	Select on Condition
	|	**call** name$(\overline{v})$	Function Call
	|	$\phi(\overline{GF \Rightarrow v})$	Phi Node
	|	$\eta(GF, i)$	Eta Node
	|	$\sigma(GF, i)$	Sigma Node

Figure 6.1: Gated Assembly Language Instructions

new instructions are: ϕ, η and σ. The ϕ-node is used at join points to describe the conditions under which different values will be defined. The η- and σ-nodes are used to describe the conditions under which a loop defined variable is complete. The semantics of these instructions were described in Chapter 4 as part of SVG.

The gating functions are *essentially* boolean expressions over values. We will give a precise definition for the gating functions in later sections when we have developed a little more background. However, it is safe to think of the gating functions as simple boolean expressions built using values, logical AND, and logical OR. Note that multiple gating functions can appear within a ϕ-node. Each gating function indicates when the associated value will be defined. As such, the set of gating functions within a ϕ-node must be mutually exclusive, and every assignment of values must satisfy exactly one of the gating functions in the set. The gating function for the entire ϕ-node is a disjunction of the individual gating functions, and must be satisfiable.

A simple example will help to clarify the new GSA instructions. Recall Figure 3.6 from Chapter 3, which is reproduced here in Figure 6.2. The left-hand side of the figure shows a function written in C that defines a simple loop. The right-hand side shows the corresponding assembly language. The assembly language code contains four blocks. The `header` and `body` blocks form a loop. The function begins execution at the `entry` block, which branches to the loop header. When we first enter the `header` block the value of i is 0, so c is `true` and we branch to `body`. The `body` block updates the t and i variables and continues back to the `entry` block. After several iterations, c becomes `false` and we branch to `end` which returns the value t (which is 45).

```
int loop() {
 int t = 1;
 int i;
 for (i = 0; i < 10; i++) {
   t += i;
 }
 return t;
}
```

C code with simple loop

```
int32 loop() {
entry:
 t = int32 1
 i = int32 0
 br header
header:
 c = slt int32 i, 10
 cbr c, body, end
body:
 t = add int32 t, i
 i = add int32 i, 1
 br header
end:
 ret int32 t
}
```

Corresponding Assembly Code

Figure 6.2: Loop Example

Figure 6.3 shows the SSA and GSA form for the assembly program in Figure 6.2. The left-hand side of the figure shows the SSA form assembly. First, note that in the SSA form, variables have been renamed so that each variable has only one definition. At the join point in block `header` we have inserted two ϕ-nodes for the two variables `t` and `i` which each have several definitions at this point. The ϕ-node for `t` says that if the control-flow is coming from block `entry`, then the value of `t2` is `t1`; otherwise, if the control-flow is coming from block `body`, then the value of `t2` is `t3`. The return value is `t2` since this is the final definition of the original variable `t` before the `end` block is reached.

The right-hand side of Figure 6.3 shows the GSA form for the assembly program in Figure 6.2. Like the SSA form, variables have been renamed so that each variable has

```
int32 loop() {
entry:
 t1 = int32 1
 i1 = int32 0
 br header
header:
 t2 = phi(entry -> t1, body -> t3)
 i2 = phi(entry -> i1, body -> i3)
 c = slt int32 i2, 10
 cbr c, body, end
body:
 t3 = add int32 t2, i2
 i3 = add int32 i2, 1
 br header
end:

 ret int32 t2
}
```

SSA for simple loop

```
int32 loop() {
entry:
 t1 = int32 1
 i1 = int32 0
 br header
header:
 t2 = mu(t1, t3)
 i2 = mu(i1, i3)
 c = slt int32 i2, 10
 cbr c, body, end
body:
 t3 = add int32 t2, i2
 i3 = add int32 i2, 1
 br header
end:
 t4 = eta(c, t2)
 ret int32 t4
}
```

GSA for simple loop

Figure 6.3: SSA and GSA for Loop Example

only one definition. At the join point in block `header` we have inserted two μ-nodes for the two variables `t` and `i` which each have several definitions at this point. We use a μ-node because these are loop-defined variables, and the gating function will be defined outside of the loop. The μ-node for `t` says that the initial value of `t2` is `t1`, and subsequent values are computed from `t3`. When we exit the loop at block `end`, the gating function for the loop-defined variable `t2` is `c`, and this is used to build an η-node representing the final value of the original `t` variable, which is returned.

6.1.2 Computing Gated SSA

Computing Gated SSA form requires that we solve two problems: first, we must locate the basic blocks that require ϕ-, μ-, η- or σ-nodes; then, we must compute the gating functions for each of these nodes. Previous algorithms by Ottenstein et al. (1990b) and Havlak (1994) start with SSA form. With these approaches, the location of the nodes has already been determined by the SSA form. Some SSA ϕ-nodes will need to be converted to η, σ, or μ, but it is easy to determine this using the connected components of the control-flow graph. This only leaves the problem of computing the gating functions. Both Ottenstein et al. (1990b) and Havlak (1994) achieve this by traversing the paths in the control-flow graph and collecting the conditions under which each path is taken. Achieving a reasonable running time for these approaches is complex. Ultimately, the best algorithms of this type are quadratic in the size of the graph, not counting the cost of computing SSA form (and other pieces of required data) in advance.

Our algorithm starts with normal assembly language as input and computes both the location of the nodes and the gating functions in a single pass. As such, our algorithm does not require the code to be put into SSA form beforehand. The running time of our algorithm is near linear and can handle arbitrary input programs. We believe our algorithm, in addition to being the best choice for computing GSA, can be used as a replacement for the usual SSA algorithms in compilers that require SSA. In such a case, a compiler writer can use our algorithm minus the computation of the gating functions to produce normal SSA form; gating functions can be added without changing the algorithm.

Our algorithm for computing Gated SSA is a variation of Tarjan's Path Compression algorithm for computing path expressions on irreducible graphs (Tarjan, 1981). A similar algorithm was proposed by Tu and Padua (1995b,c), upon which we make several improvements. First, we extend the algorithm to programs with irreducible control-flow. Second, we simplify the path compression algorithm by introducing a more uniform algebra for gating expressions. This allows us to use a simple compression technique on our path expressions as described in Tarjan (1975, 1979).

In the remainder of this chapter we will present our algorithm for computing Gated SSA form. First, we will review some basic definitions and lemmas, and present the core theorem which enables our approach. Then, we will present a simple algorithm based on Gaussian elimination with bad complexity. We will then improve this algorithm using a technique called "decomposition by dominators". Finally, we extend our algorithm to arbitrary graphs.

6.2 Background

Our algorithm works over the control-flow graph of the input program and the graph's associated dominator tree. The control-flow graph is constructed from the basic blocks of the input functions. A formal definition is given below.

Definition 6. *A* **Control Flow Graph** *(N,E) is a directed graph whose nodes (N) are the basic blocks in a function, and whose edges (E) represent the possible flows of control between basic blocks.*

An example of a control-flow graph is shown in Figure 6.4. By convention, we number the basic blocks of each function starting with 0 ensuring that there are no gaps in

the numbering. Thus, the entry block of each function is always numbered 0, and the highest numbered block is one less than the total number of blocks in the function. We will use lower-case letters to stand for block numbers (e.g. u, v, x), and upper-case

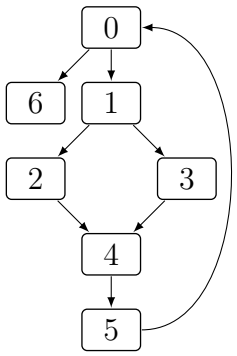


Figure 6.4: A Control-flow Graph

letters to stand for sets of blocks (e.g. X). We will indicate edges in the control-flow graph using the notation: $u \to v$ for a single edge, $u \to^+ v$ for one or more edges connecting u and v, and $u \to^* v$ zero or more edges connecting u and v. We will refer to $u \to^* v$ as a *path* between u and v. For example, in Figure 6.4, we have two paths between blocks 1 and 4 which are: $1 \to 2 \to 4$, and $1 \to 3 \to 4$. We can also partially describe a path between blocks 0 and 5 by: $0 \to^* 4 \to 5$, or $0 \to^* 3 \to^+ 5$. If there is an edge $u \to v$ in a given control-flow graph, we will refer to u as a *successor* of v, and to v as a *predecessor* of u.

We will also make use of the dominence relation between blocks which we specify using the three definitions below.

Definition 7. *A block u* **dominates** *a block v, written: $u \gtrdot v$, if every control flow path to v must pass through u.*

Definition 8. *A block u* **strictly dominates** *v, written: $u \gg v$ if $u \gtrdot v$ and $u \neq v$.*

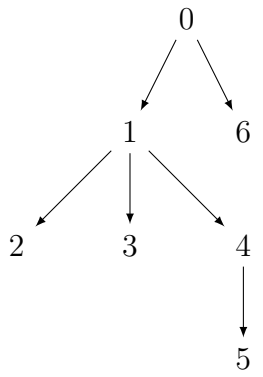


Figure 6.5: Dominator Tree for Control-flow Graph in Figure 6.4

Definition 9. *A block u is the* **immediate dominator** *of v, written: $\texttt{idom}(v)$, if $u \gg v$, and forall x, such that $x \gg v$, $x \gg u$.*

This last definition defines a unique immediate dominator for each block in the control-flow graph except for the entry block 0. Therefore, we can form a tree, called the *dominator tree*, in which each block's parent is its immediate dominator. For example, the dominator tree for the control-flow graph shown in Figure 6.4 is shown in Figure 6.5. The tree is always rooted at 0 since this is the only block with no immediate dominator. If $u = \texttt{idom}(v)$, we will refer to u as the *parent* of v, and we will refer to v as a *child* of u.

Finally, the following definitions will be needed to connect our algorithm with traditional SSA algorithms.

Definition 10. *The* **dominance frontier** *of a block u, written $\texttt{DF}(u)$, is defined as the set of blocks:*

$$\texttt{DF}(u) = \{v \mid \exists x.x \to v \wedge u \ggcurly x \wedge u \not\gg v\} \qquad .$$

Definition 11. *The* **iterated dominance frontier** *of a block u, written* $\mathtt{DF}^+(u)$*, is the transitive closure of the dominance frontier.*

The locations where gating functions are needed is related to the iterated dominance frontier by the following theorem:

Theorem 1 ((Cytron et al., 1991))**.** *If X is the set of blocks in which a variable x is defined, then* $\mathtt{DF}^+(X)$ *is the minimum set of blocks that require ϕ-nodes for the variable x.*

Traditional SSA algorithms apply this theorem directly by computing the iterated dominance frontier for the set of blocks in which a variable is defined, and then inserting ϕ-nodes within those blocks for the given variable. Our problem is more complex in that we must also compute the gating functions. Our algorithm will instead use the dominator tree to compute where the ϕ-nodes (and η- and σ-nodes) need to be placed. Along the way, we will compute the information needed to generate the gating functions. To motivate our algorithm, we will first show a relationship between the dominator tree and the iterated dominance frontier.

6.2.1 Relation Between The Dominator Tree and The Dominance Frontier

The next three lemmas establish the following fact: for any control-flow graph, only blocks in sibling sub-trees of the graph's dominator tree can appear together in the dominance frontier relation. Put another way, if $u \in \mathtt{DF}^+(v)$, then there must be a block x which dominates both u and v; and in fact, $\mathtt{idom}(u)$ is this block. This fact is what allows us to use Tarjan path compression algorithms to compute the location

of ϕ-nodes and their gating functions. Using this fact, we can structure our algorithm as a traversal of the dominator tree, and at each step only consider descendants of the current block. We will return to the algorithm shortly; first we will establish this important fact. We begin with a small helper lemma.

Lemma 1. *For any path $d \rightarrow^+ v$ in a CFG, if $d \gg v$ and d occurs only once in the path, then d dominates every block in the path.*

Proof. Suppose the opposite, then there is a block u such that $d \not\gg u$, and $d \rightarrow^+ u \rightarrow^* v$. However, this implies that the path $0 \rightarrow^* u \rightarrow^* v$ does not contain d which contradicts $d \gg v$. □

We demonstrate the relationship between the immediate dominator and the dominance frontier with the following core lemma. The core lemma shows that the immediate dominator of a block must also dominate all of the blocks in its dominance frontier.

Lemma 2 (Core Lemma)**.** *For a control-flow graph $G = (N, E)$ and $v, x \in N$, if $v \in \mathit{DF}(x)$, then $\mathit{idom}(v) \gg x$.*

Proof. From the definition of $\texttt{DF}(x)$, $x \neq \texttt{idom}(v)$. Suppose the opposite: $\texttt{idom}(x) \not\gg v$, then there is a path, $p : 0 \rightarrow^* x$, from the entry node to x which does not contain $\texttt{idom}(v)$.

However, because $v \in \texttt{DF}(x)$, there exists w such that $w \rightarrow v \in E$ and $x \gtrdot w$ and $x \not\gg v$. Let $x \rightarrow^* w \rightarrow v$ be a path from x to v such that x only appears once (if x appears more than once, cycles $x \rightarrow^* x$ can be removed). The node $\texttt{idom}(v)$ must not appear in the path $x \rightarrow^* w$. If it did, then by Lemma 1 we would have $x \gg \texttt{idom}(v) \gg v$.

Taking these two observations, we can form the path $p \rightarrow^* x \rightarrow^* w \rightarrow v$. Thus, we have a path from the entry node to v that does not contain $\texttt{idom}(v)$, which is a contradiction. □

We can easily extend the core lemma to the iterated dominance frontiers.

Lemma 3 (Sibling Frontiers)**.** *If $u \in \mathit{DF}^+(x)$, then $\mathit{idom}(u) \gg x$.*

Proof. By induction on n given $\texttt{DF}^n(x)$.

Base Case. If $n = 1$, then since $u \in \texttt{DF}(x)$, $\texttt{idom}(v) \gg x$ by Lemma 2.

Inductive Case. Let $u \in \texttt{DF}^{n-1}(x)$ and $v \in \texttt{DF}(u)$, then by the induction hypothesis, $\texttt{idom}(u) \gg x$, and by Lemma 1 $\texttt{idom}(v) \gg u$. Therefore:

$$\texttt{idom}(v) \gg u \Rightarrow \texttt{idom}(v) \gtrdot \texttt{idom}(u) \gg x \Rightarrow \texttt{idom}(v) \gg x \quad .$$

□

By Lemma 3, we know that the blocks in the iterated dominance frontier for a node x are contained in the children of the immediate dominator of x. Using this we can compute the locations for ϕ-nodes related to x by only considering the children of $\texttt{idom}(x)$ in the dominator tree. However, not all of the children of $\texttt{idom}(x)$ are in the dominance frontier of x. To narrow in on the exact blocks that need ϕ-nodes, we need to look at some of the paths in the control-flow graph. We will now define these paths, which we refer to as *gating paths*.

6.2.2 Gating Paths

A gating path is a path in the control-flow graph which is relevant for computing a gating function. We will now make this definition more precise through a series

of lemmas which follows the development found in Tu and Padua (1995b). We start with a definition for gating paths and then connect this to the iterated dominance frontier and the gating functions.

Definition 12. *A* **gating path** *for a node v is a path in the Control Flow Graph from* `idom`*(v) to v that only contains nodes dominated by* `idom`*(v).*

This definition makes a connection between the dominator tree and the control-flow graph. One way to think of this definition is to imagine restricting a control-flow graph to only the blocks which appear in a sub-tree of the dominator tree. Our first result shows that every node in a control-flow graph has a gating path.

Lemma 4 (Existence of Gating Paths)**.** *Given a CFG (N,E), for every basic block $n \in N$, there is a gating path from* `idom`*(n) to n.*

Proof. Since $\texttt{idom}(n) \gg n$, there exist a path $\texttt{idom}(n) \rightarrow^* n$. If this path contains cycles of the form $\texttt{idom}(n) \rightarrow^* \texttt{idom}(n)$, they can be removed giving us a new path $\texttt{idom}(n) \rightarrow^* n$ in which $\texttt{idom}(n)$ occurs only once. Suppose there is no gating path from $\texttt{idom}(n)$ to n, then there is a node, u on the path such that $\texttt{idom}(n) \not\gg u$. Therefore, there is a path $S \rightarrow^* u \rightarrow^* n$ that does not contain $\texttt{idom}(n)$, which contradicts $\texttt{idom}(n) \gg n$. □

The next two lemmas connect the iterated dominance frontier to gating paths. Specifically, nodes in the iterated dominance frontier for a node x, must have gating paths which include x.

Lemma 5. *If $v \in$ `DF`(x), then there is a gating path from* `idom`*(v) to v which passes through x.*

Proof. By definition, $\exists w, w \to v \wedge x \ggcurly w$. Using this and Lemma 4 we can construct a gating path from x to w. Because the path $x \to^* w$ is a gating path, it contains only nodes dominated by x and therefore dominated by $\texttt{idom}(v)$.

By Lemma 2, $\texttt{idom}(v) \gg x$, hence, by Lemma 1 there is a gating path from $\texttt{idom}(v)$ to x. Therefore, the path $\texttt{idom}(v) \to^* x \to^* w \to v$ is a gating path. □

Lemma 6. *If $v \in \mathit{DF}^+(x)$, then there is a gating path from $\mathit{idom}(v)$ to v which passes through x.*

Proof. By induction on n given $\texttt{DF}^n(x)$ and Lemmas 3 and 5. □

We will now show the reverse relation: nodes which appear on a gating path are in the dominance frontier relation with the end of the path.

Lemma 7. *If there is a gating path $\mathit{idom}(v) \to^+ x \to^* v$, then $v \in \mathit{DF}^+(x)$.*

Proof. Since $\texttt{idom}(v) \neq x$, v must be a join node in the control-flow graph. Otherwise, w would be the immediate dominator of v. If there are no other join nodes, then in the path $x \to^* w \to v$, every node can have only one predecessor, and therefore, $x \ggcurly w$. Because $\texttt{idom}(v) \neq x$ and $\texttt{idom}(v) \gg x$ but $x \not\gg v$, we have $v \in \texttt{DF}(x) \subset \texttt{DF}^+(x)$.

The remainder of the proof proceeds by induction on the number of join nodes on the sub-path $x \to^* w \to v$. For additional details see Tu and Padua (1995b). □

Putting Lemmas 5 and 7 together we arrive at the following Theorem due to Tu and Padua (1995b).

Theorem 2 ((Tu and Padua, 1995b))**.** *Given a control-flow graph (N,E), and $X \subset N$, then for any $v \in N$, $v \in \bigcup_{x \in X} \mathit{DF}^+(x)$ if and only if there is a gating path from $\mathit{idom}(v)$ to v containing a node that belongs to X.*

Proof. Follows from Lemmas 6 and 7. □

Having established a correspondence between gating paths and the iterated dominance frontier, we can now compute traditional SSA form using gating paths. First, we compute the gating paths for each node v in a control-flow graph. Then, by Theorem 2, we know that for each variable defined in the blocks which appear on these paths we may need a ϕ-node. One advantage of this approach is that we can structure the algorithm around the dominator tree which will give us better complexity. More importantly, by computing the gating paths we will not only identify locations for ϕ-nodes, but we will also compute the paths leading to each ϕ-node (or η- or σ-node) which are relevant for the gating functions. Using these paths we can build the gating functions required for Gated SSA. In the next section we will describe how the gating functions are computed from the gating paths.

6.3 Gating as a single-source path-expression problem

Once we have identified the location for a ϕ-, η-, or σ-node using Theorem 2, we can construct a gating function by considering all gating paths leading to the CFG node in question. For example, in Figure 6.4, we need an η-node at CFG node 6. We can compute a gating function by considering all gating paths that lead to 6. Some

of these paths are shown below:

$$\begin{aligned} p_1 &= 0 \to 6 \\ p_2 &= 0 \to 1 \to 2 \to 4 \to 5 \to p_1 \\ p_3 &= 0 \to 1 \to 3 \to 4 \to 5 \to p_1 \\ p_4 &= 0 \to 1 \to 2 \to 4 \to 5 \to p_2 \\ p_5 &= \ldots \end{aligned}$$

There are two things to note about these paths. First, we can use the more concise language of *regular expressions* to represent sets of paths. For example, the paths above could be represented as:

$$\begin{aligned} p_1 &= 0 \to 6 \\ p_{2,3,4} &= \{0 \to 1 \cdot \{1 \to 2 \to 4, 1 \to 3 \to 4\} \cdot 4 \to 5 \to 1\}^* \to p_1 \end{aligned}$$

Where the notation $x \cdot y$ is used to represent concatenation of sets of paths, and the notation * is used to represent transitive closure with respect to concatenation. Second, not all paths need to be considered; for instance, the path p_4 does not add any information which is not contained in paths p_2 and p_3. In this case, using regular expressions is not only convenient, it allows us to easily characterize which paths are relevant. Namely, we need only consider simple, non-redundant regular expressions over paths. To make this precise we will introduce some notation for regular expressions over paths, which we refer to as *path expressions*.

Path Expressions. Formally, path expressions are defined over an alphabet of edges. For our purposes the alphabet will be the set of edges in a given control-flow graph. For reasons which will become clear, we will use the notation $\gamma_{(x,y)}$ to denote the atomic path expression for the edge $x \rightarrow\ y$.

Definition 13 (Path Expressions)**.** *A* path expression *over a set of edges E is one of:*

1. $\emptyset$ *the empty set.*
2. Λ *the empty path.*
3. $\gamma_{(x,y)}$ *an atomic path from node x to node y where $x \rightarrow\ y \in E$.*
4. $P_1 \cup P_2$ *union of two path expressions.*
5. $P_1 \cdot P_2$ *concatenation of path expressions.*
6. P^* *reflexive transitive closure under concatenation.*

We can interpret path expressions as sets of paths within a CFG. Later we will give an alternate interpretation of path expressions, however, for the time being, we will interpret path expressions as the following sets:

$$\begin{aligned}
[\![\emptyset]\!] &= \emptyset \\
[\![\Lambda]\!] &= \{\Lambda\} \\
[\![\gamma_{(x,y)}]\!] &= \{x \rightarrow y\} \\
[\![P_1 \cup P_2]\!] &= \{p \mid p \in [\![P_1]\!] \vee p \in [\![P_2]\!]\} \\
[\![P_1 \cdot P_2]\!] &= \{p_1 \rightarrow p_2 \mid p_1 \in [\![P_1]\!] \wedge p_2 \in [\![P_2]\!]\} \\
[\![P^*]\!] &= \bigcup_{n=0}^{\infty} [\![P]\!]^n \text{ where } P^0 = \{\Lambda\} \text{ and } P^n = P^{n-1} \cdot P
\end{aligned}$$

From this definition, we see that $\emptyset$ is a unit for $\cup$ and a zero for concatenation. In addition, we will arrange for Λ to be a unit for concatenation:

$$\emptyset \cup P = P \cup \emptyset = P$$

$$\emptyset \cdot P = P \cdot \emptyset = \emptyset$$

$$\Lambda \cdot P = P \cdot \Lambda = P$$

These laws are consistent with the following identities for closure, which are also satisfied by our interpretation of path expressions as sets of paths.

$$\emptyset^* = \{\Lambda\}$$

$$\Lambda^* = \{\Lambda\}$$

We will require that any interpretation of path expressions satisfy these laws. We say that two path expressions P_1 and P_2 are *equivalent* if $[\![P_1]\!] = [\![P_2]\!]$. In addition, a path expression, P, is *simple* if either $P = \emptyset$ or $\emptyset$ is not a sub-expression of P. Any path expression can be made simple by repeated application of the equalities above.

Finally, we would like our path expressions to uniquely represent the paths they describe. We accomplish this by defining *non-redundant* path expressions.

Definition 14 (Non-Redundant Path Expression)**.** *A path expression is* **non-redundant** *if:*

1. *P is atomic ($P = \emptyset$, Λ, or $\gamma_{(x,y)}$)*
2. *if P_1 and P_2 are non-redundant, then*
   (a) *$P_1 \cup P_2$ is non-redundant if $[\![P_1]\!] \cap [\![P_2]\!] = \emptyset$*

(b) $P_1 \cdot P_2$ *is non-redundant if* $\forall w \in [\![P_1 \cdot P_2]\!], \exists! w_1, w_2 \mid w = w_1 w_2 \wedge w_1 \in [\![P_1]\!] \wedge w_2 \in [\![P_2]\!]$

3. P^* *is non-redundant if* $\forall w \in [\![P^*]\!], \exists! w_1, \ldots, w_k \mid w = w_1 \ldots w_2 \wedge w_i \in [\![P]\!]$

Computing Gating Functions With these definitions for path expressions, we can now recast the problem of computing gating functions as a single-source path-expression problem. A singe-source path-expression problem is the problem of computing for a given source vertex s and for each vertex v a non-redundant path expression $P(s, v)$ such that $[\![P(s, v)]\!]$ contains all of the paths from s to v.

Given an algorithm for single-source path-expression, we can compute a gating function in the following way. First, for each node, we enumerate the paths in $[\![P(\texttt{idom}(n), n)]\!]$. These paths are exactly the gating paths described earlier. Note that we can identify the ϕ-nodes needed at block n by inspecting the elements of these paths thus giving us a way to compute normal SSA form. For each gating path, we can compute the conditions for control to flow along the edges of that path by looking at the basic blocks along the paths. The set of conditions we compute for $[\![P(\texttt{idom}(n), n)]\!]$ makes up the gating function for block n.

The algorithm described above requires us to compute the gating paths for each node in the control-flow graph, and then compute conditions along each of these paths. The complexity of this process is proportional to the number of basic blocks times the number of edges in the graph. We can improve on this by reinterpreting the path expressions as gating functions directly. This is the first step towards our final algorithm, which we describe this in the next section. Of course, we must also describe how to compute $[\![P(\texttt{idom}(n), n)]\!]$, a problem we will address in Section 6.4.

6.3.1 From Path Expressions to Gating Functions

In this section we will present the algorithm for computing gating functions from path expressions. Throughout the remainder of this chapter we will present source code for algorithms and data structures using Haskell (Jones, 1998) as our implementation language. We will begin by writing down data structures for path expressions. Path expressions are represented using the following data type definition.

```
data PathExpr a
   = Lambda
   | Empty
   | Gamma { from :: Int, to :: Int, value :: a }
   | PathExpr a :+ PathExpr a
   | PathExpr a :. PathExpr a
   | Star (PathExpr a)
     deriving Eq
```

The type `PathExpr` is parametrized over an arbitrary type `a`, which will be used to carry additional information within our path expressions. The `Lambda` and `Empty` constructors represent Λ and $\emptyset$ respectively. The `Gamma` constructor represents $\gamma(x, y)$, and is declared as a record with three fields named `from`, `to`, and `value`. The value field is given the type `a` indicating that we can store any type of additional data within the atomic paths. We use the infix constructor `(:+)` to represent union and the infix constructor `(:.)` to represent concatenation. Finally, the `Star` constructor is used to represent closure under concatenation.

As mentioned in the previous section, `Empty` is a unit for union and a zero for concatenation, `Lambda` is a unit for concatenation, and the closure of `Empty` or `Lambda` is the empty path, `Lambda`. We can capture these algebraic properties using the "smart" constructor functions below.

```
-- union smart constructor
gcup :: PathExpr a → PathExpr a → PathExpr a
gcup Empty g = g
gcup g Empty = g
gcup x y     = x :+ y

-- concatenation smart constructor
gdot :: PathExpr a → PathExpr a → PathExpr a
gdot Empty _   = Empty
gdot _ Empty   = Empty
gdot Lambda g  = g
gdot g Lambda  = g
gdot x y       = x :. y

-- closure smart constructor
gstar :: PathExpr a → PathExpr a
gstar Empty  = Lambda
gstar Lambda = Lambda
gstar g      = Star g
```

It is safe to use `gcup`, `gdot`, and `gstar` in place of the constructors `(:+)`, `(:.)`, and `Star` respectively. Doing so will eliminate unnecessary path expression structure as soon as possible. However, we will sometimes not use them, and produce longer path expressions for presentation purposes. In such cases we can apply our algebraic simplification rules using the function `simplify` below.

```
simplify :: PathExpr a → PathExpr a
simplify (Star x) = gstar (simplify x)
simplify (x :+ y) = gcup  (simplify x) (simplify y)
simplify (x :. y) = gdot  (simplify x) (simplify y)
simplify g        = g
```

Gating Functions. Our gating functions are a form of path expression. Gating functions are path expressions where each atomic path carries a condition that must

be true for the control-flow to follow that path. We represent this using the `Gate` type defined below.

```
type Gate = PathExpr [Term]
```

`Term` is an abstract type representing terms in our intermediate representation: essentially LLVM assembly statements. Each gate carries a list of terms which are interpreted as a conjunction of boolean values. An empty list is interpreted as equivalent to `true`.

To use the `Gate` type we need to be able to construct atomic path expressions. These are the paths built using the `Gamma` constructor. For a given control-flow graph with edges E, an atomic path expression has the form `Gamma f t c`, where `f` and `t` are labels corresponding to an edge $f \rightarrow t \in E$, and c is a condition which must be true when this edge is traversed. In order to compute the atomic path expressions, we examine the control flow instruction of the `from` block. This is done by the function `evalEdge`, which is shown below.

```
evalEdge :: Label → Label → Gate
evalEdge m n = pathExpr (blockOf m) (blockOf n)
```

The function `evalEdge` takes two labels and returns an atomic gating function for the edge between the two basic blocks with those labels. It does this by first looking up the basic block for each label and calling the helper function `pathExpr`. In order to look up the basic blocks for a label, we use the function `blockOf` which has type: `Label → GBlock`[1]. The `GBlock` type is a record containing information about a basic

[1]The acute reader will notice that `blockOf`, and therefore `evalEdge` must be monadic in order to track the mapping of label to blocks needed to implement `blockOf`. This is indeed true, however only complicates the presentation and has thus been omitted.

block such as its instructions, predecessors and successors, depth-ordering, etc. We will introduce accessor functions for `GBlock` when necessary.

The `pathExpr` helper function is shown below. This function is responsible for constructing atomic path expressions of type `Gate`.

```
pathExpr :: GBlock → GBlock → Gate
pathExpr bx by =
    case control bx of
      Seq z | y == z   → Gamma x y []
      MBr c ty dl arms
          | dl == y    → Gamma x y $ map (ne c ∘ fst) arms
          | otherwise  → Gamma x y [ eq c (yval arms) ]
 where
   x = label bx
   y = label by

   yval  :: [(Term,Label)] → Term
   eq,ne :: Term → Term → Term
```

To build an atomic path expression for the edge $x \rightarrow y$, we look at the control transfer instruction for the block `bx`, which has label `x`. We fetch the control transfer instruction from the block with the accessor function `control`. The control transfer instruction can be one of two things: either an unconditional branch (represented by `Seq`), or a multi-way branch (represented by `MBr`). The unconditional branch carries the label of the target block, `z`. The `pathExpr` function checks that this label is equal to `y`, the label for block `by`, and returns an atomic path expression with an empty list of conditions (which is equivalent to `true`) indicating that the edge is always traversed.

The multi-way branch constructor has four arguments. The type of this constructor is show below.

```
MBr :: Term → Type → Label → [(Term,Label)] → Control
```

The first and second parameters are the type and term that is tested by the conditional branch. The third argument is a default label which is used if no other edge is taken by the branch. The fourth argument is a list of term and target labels. If the condition term is equal to a term in the list then the corresponding label is the target of the branch. For example, a simple if statement testing a boolean condition, `c`, would be represented as:

```
MBr c (I 1) falseLabel [(true, trueLabel)]
```

Returning to `pathExpr`, in the case of a multi-way branch, we first check if the target label, `y`, is equal to the default label. If so, then the condition for taking this edge is that the tested term, c, is not equal to any of the terms in the multi-way branch arms. We construct this condition by producing a list of terms of the form $x \neq c$ for all terms x in the list of branch arms. We use the function `ne` to build the expression for testing inequality. Otherwise, if the target label, `y`, is not equal to the default label, then we use the function `yval` to find and return the term associated with the label `y` (a simple list search). The condition for taking this edge is that c is equal to this value.

Using `evalEdge`, we can construct a set of atomic edges for a control-flow graph. Then, if we compute $[\![P(\texttt{idom}(n), n)]\!]$ using these edges, we can use the result to build the gating function for block n. We accomplish this by reinterpreting the path expressions not as sets, but as boolean expressions. The alternate interpretation for

path expressions is shown below.

$$[\![\emptyset]\!]_b = False$$

$$[\![\Lambda]\!]_b = True$$

$$[\![\gamma_{(x,y,ts)}]\!]_b = \bigwedge_{t=ts} t$$

$$[\![P_1 \cup P_2]\!]_b = [\![P_1]\!]_b \vee [\![P_2]\!]_b$$

$$[\![P_1 \cdot P_2]\!]_b = [\![P_1]\!]_b \wedge [\![P_2]\!]_b$$

$$[\![P^*]\!]_b = [\![P]\!]_b$$

In this case we are dealing with `Gate`'s which have `Term`'s embedded in them. We have represented the embedded `Term`'s with the notation ts which we use as a set. This interpretation is implemented by the `gateToCondition` function shown below.

```
gateToCondition :: Gate → Term
gateToCondition g =
  case simplify g of
    Lambda       → true
    Empty        → false
    Gamma _ _ c  → conj c
    p :+ q       → disj [gateToCondition p, gateToCondition q]
    p :. q       → conj [gateToCondition p, gateToCondition q]
    Star p       → gateToCondition g
 where
   true, false :: Term
   conj :: [Term] → Term
   disj :: [Term] → Term
```

The `gateToCondition` function is designed to be applied to a path expression after-the-fact to build a gating function. However, we can dispense with path expressions altogether by introducing a set of "smart" constructors which perform the function of `gateToCondition` as the path expressions are built. Although our imple-

mentation makes this optimization, we will not pursue it further here. Instead, we will continue to work with path expressions for expository purposes.

We have now described the basic machinery of our algorithm. Our basic approach is centered around computing path expressions for a control-flow graph. The path expressions allow us to compute, as a special case, the gating paths for every basic block in the graph. These gating paths tell us where ϕ-, $\eta-$, and σ-nodes are needed, and also give us a way to easily compute the gating functions for those same blocks. We will now turn to the main problem of computing path expressions for a control-flow graph.

6.4 Path Expressions via Gaussian Elimination

In this section we describe a simple algorithm for computing the set of gating paths $[\![P(\texttt{idom}(n), n)]\!]$. We will improve this algorithm in later sections. The algorithm presented in the section has poor complexity, but is easy to understand and will give some intuition for later versions of the algorithm.

Generally speaking, our algorithm is an instance of Gaussian elimination. Recall, Gaussian elimination is an algorithm for solving systems of linear equations. Specifically, in the case of matrices, we are solving equations of the form: $Ax = b$ where A is an $n \times n$ matrix, and x and b are vectors of length n. To solve this equation, we first perform Gaussian elimination (also called *LU-decomposition* in this specific case) to put our equation in the form:

$$Ax = LUx = b$$

where L is a lower-triangular matrix, and U is an upper-triangular matrix. Then, we solve:

$$Ly = b$$

for the variable y; since L is lower-triangular, this can be done easily with forward and backward substitution. Then we are left with solving:

$$Ux = y$$

which can also be done with forward and backward substitution. One advantage of this algorithm is that, if A is held constant, we can reuse the decomposition to solve for many different values of b. Since there are more efficient algorithms for solving triangular systems than general systems, this can result is a significant speedup when solving for multiple values of b.

For our problem, the role of A is played by a matrix of size $n \times n$ where n is the number of nodes in the control-flow graph. The matrix element (u, v) is a path expression representing the paths in the control-flow graph between the blocks u and v. Our matrix is sparse because most pairs of blocks will not have a path between them. Therefore, we represent this matrix with a finite map from pairs of block labels to path expressions. The type declarations for our finite map and its related functions is shown below.

```
type PathName = (Label,Label)
data PathMap  -- abstract type
emptyPathMap :: PathMap

fetch  :: PathMap → PathName → Gate
store  :: PathMap → PathName → Gate → PathMap
adjust :: (PathName → Gate → Gate) → PathMap → PathName → PathMap
```

The type `PathName` represents elements of our sparse matrix, and `PathMap` is the finite map representing the matrix itself. We require the finite map to have functions `fetch`, and `store` which read elements from and replace elements of the map. We also have a convenience function `adjust` for applying a function to a specific element of the matrix and replacing it with the result.

6.4.1 Path Sequences

With our analog of the matrix A, we are free to chose the order of the row and column elements. That is, the columns can be constructed from any ordering of the block labels, and the rows can be constructed from a possibly different ordering of the block labels. Conceptually, our analog to decomposition must order the rows and columns so that the paths in the two triangular matrices do not have any gaps that would require elements from the other matrix. In fact, we will do much better by defining a total ordering on pairs of block labels such that path prefixes can be constructed from prefixes of the ordering. This ordering is called a *path sequence*.

Definition 15. *A* **path sequence** *for a control-flow graph $G = (N, E)$ is a sequence $\overline{(P_i, u_i, v_i)}$ where $u_i, v_i \in N$, and each P_i is a path expression for paths between blocks u_i and v_i such that:*

1. *if $u_i = v_i$, then $P_i = \Lambda$, and*
2. *for any non-empty path $p \in G$, there is a unique set of indices $\overline{i_j}$ and a unique partition $\overline{p_j}$ of p into non-empty paths such that:*

$$\forall a, b. a < b \Rightarrow i_a < i_b \quad \wedge \quad \forall j. p_j \in [\![P_{i_j}]\!] \qquad .$$

In code, we represent a path sequence as a `PathMap` together with an ordered list of block-label pairs.

```
type PathSequence = ([PathName], PathMap)
```

Generating the path sequence is done with our analog of Gaussian elimination. We must produce an ordering of block-label pairs, and a path map such that path elements satisfy the ordering in the sense of being a path sequence. What we mean by this is captured in the definition below.

Definition 16. *a* **decomposed path map** *is a path map such that, for a given ordering of blocks, and for each* (u, v) *either:*

1. *if* u *appears before* v *in the ordering, then* $P(u, v)$ *is a non-redundant path expression with no intermediate blocks which appear after* v *in the ordering, or*
2. *if* u *appears after* v *in the ordering, then* $P(u, v)$ *is a non-redundant path expression with no intermediate blocks which appear before* u *in the ordering.*

For the moment, let us assume we have a function `eliminate` which produces a decomposed path map: a `PathMap` with the properties described above. Then, using `eliminate`, we can create a path sequence using the `generateSequence` function, shown here.

```
generateSequence :: [GBlock] → PathSequence
generateSequence blocks =
    let pm  = eliminate blocks in
    let lowerLeft  = [ (u,w) | u ← labels, w ← sectionFrom u labels
                             , let uw = fetch pm (u,w)
                             , notEmpty uw
                             , notLambda uw ] in
    let upperRight = [ (u,w) | u ← labels, w ← sectionTo u labels
```

```
                                , let uw = fetch pm (u,w)
                                , notEmpty uw ] in
     (lowerLeft ++ reverse upperRight, pm)
  where
    labels = topOrder blocks
```

This function first calls `eliminate` to produce a decomposed path map. Then, we produce a list of elements which correspond to the lower-left and upper-right matrices from before. Notice, that our function orders the rows and columns in topological order, and the list of ordered labels are accessed using the functions `sectionFrom` and `sectionTo`. These functions abstract a simple list so that we can easily replace the implementation with an array.

Our abstraction of simple lists are called "sections". We define sections with an abstract type `Sequence` and the functions shown below.

```
data Section — abstract type

dfOrder  :: [GBlock] → Section
topOrder :: [GBlock] → Section

sectionTo    :: Label → Section → Section
sectionFrom  :: Label → Section → Section
sectionAfter :: Label → Section → Section
```

The two functions `dfOrder` and `topOrder` produce sequences from a set of basic blocks which are either in depth-first or topological order. The function `sectionTo` returns the prefix of a section up to and including the given label. The function `sectionFrom` returns the suffix of a section including and following the given label. The function `sectionAfter` returns the suffix of a section following the given label.

The correctness of `generateSequence` was proved by Tarjan (1981). Formally, Tarjan proved the following theorem:

Theorem 3 (Correctness of `generateSequence`)**.** *Let* `eliminate` *be a function which produces a decomposed path map for a given control-flow graph G, then* `generateSequence G` *produces a path sequence for G.*

6.4.2 Front- and Back-solving

The path sequences presented in the previous section may, at first, seem like arbitrary constructions. To better understand path sequences, we will now look at how they are used to solve the single-source path-expression problem. The algorithm in this section is analogous to the front- and back-solving methods used to solve matrix equations. The main function, `solve`, takes a path sequence and a source label and produces a path map which, for any basic block in the graph, gives a path sequence representing all paths between the source and that block.

The entry point for the simple solving algorithm is shown below. We will use a state monad to track the current path map as it is being built.

```
solve :: PathSequence → Label → PathMap
solve pathSeq source = runST $
    do ref ← newSTRef initialize
       solveLoop pathSeq source ref
       readSTRef ref
 where
   initialize = store emptyPathMap (source,source) Lambda
```

Most of the `solve` function deals with setting up the ST-monad, allocating a reference cell to hold the current path map, and returning the final result. The only interesing part of `solve` is the initialization of the path map. Recall, that our path maps are sparse, and any entry not found in the map is understood to be `Empty`. Therefore, the only entry we need to initialize is the entry for the path from the source to itself,

which starts out as `Lambda`, the empty path.

The main work of the solving algorithm is done in `solveLoop`. Within solve loop we will use monadic versions of `fetch` and `store` to modify the current path map. Following convention, we call these function `get` and `set`.

```
get :: PathName → ST a Gate
get e = readSTRef spm >>= λm → return (fetch m e)

set :: PathName → Gate → ST a ()
set e g = readSTRef spm >>= λm → writeSTRef spm (store m e g)
```

Finally, we have `solveLoop` itself. This function processes the pairs of block labels in the order dictated by the path sequence, and for each one updates the current path map.

```
solveLoop :: PathSequence → Label → STRef a PathMap → ST a ()
solveLoop (labels,pm) s spm =
    forM_ labels $ λ(v,w) →
      do sv ← get (s,v)
         sw ← get (s,w)
         let vw = fetch pm (v,w)
         if v == w
          then set (s,v) (sv :. vw)
          else set (s,w) (sw :+ (sv :. vw))
```

For each (v, w), we will look at the path expression $v \rightarrow^* w$. If $v = w$, then we have a cycle, and we update the path $s \rightarrow^* v$ to include this cycle. Otherwise, we can construct a new path $v \rightarrow^* s \rightarrow^* w$ and add this to the known paths from s to w. At each step, because of the properties of the path seqeunce, we know that we have processed all prefixes of the paths we are considering.

The correctness of `solve` was proved by Tarjan (1981) with respect to path sequences. Formally, Tarjan proved the following theorem:

Theorem 4 (Correctness of `solve`)**.** *If S is a path sequence for a control-flow graph $G = (V, E)$, then for any $s, v \in N$,* `fetch (solve S s) (s,v)` *is a non-redundant path-expression representing all paths from s to v.*

6.4.3 Decomposition

Finally, we come to the problem of decomposition, and the implementation of the function `eliminate`. Recall, that the job of `eliminate` is to produce a path map with the properties listed in Definition 16. Like `solve`, we will introduce a state monad and a reference cell to hold the current path map. We will process the basic blocks in topological order after initializing the path map using `elimInit`.

```
eliminate :: [GBlock] → PathMap
eliminate blocks =
    runST $ do psr ← newSTRef (elimInit blocks)
               elimLoop (topOrder blocks) psr
               readSTRef psr
```

The job of `elimInit` is to start out the algorithm with all of the atomic path expressions for the edges in the control-flow graph. The `elimInit` function is shown below.

```
elimInit :: [GBlock] → PathMap
elimInit blocks =
    foldl (adjust initEdge) emptyPathMap (edges blocks)
 where
   initEdge (x,y) exp = exp :+ evalEdge x y

   edges :: [GBlock] → [(Label,Label)]
   edges blocks = nub $ concatMap outEdges blocks
   outEdges b   = [ (label b, x) | x ← cfgsuc b]
```

Initialization proceeds by calling `adjust initEdge` for each of the edges that appear in the control-flow graph. For each edge, we construct the atomic path expression by calling `evalEdge`. The atomic path expression is added to the current path expression using the union constructor (`:+`). In this case, the union is not necessary since we start with an empty path map and only process each edge once (because of `nub`). However, the union allows us to relax these conditions if necessary.

Most of the work of decomposition is done by the `elimLoop` function below. Like `solveLoop`, this function is monadic, and uses monadic versions of `fetch` and `store`, which are called `get` and `set`.

```
elimLoop :: Section → STRef a PathMap → ST a ()
elimLoop ns psr =
    forM_ ns $ λv →
    do vv ← get (v,v)
       set (v,v) (Star vv)
       forM_ (sectionAfter v ns) $ λu →
         do uv ← get (u,v)
            vv ← get (v,v)
            when (notEmpty uv) $
             do set (u,v) (uv :. vv)
                forM_ (sectionAfter v ns) $ λw →
                  do uw ← get (u,w)
                     uv ← get (u,v)
                     vw ← get (v,w)
                     when (notEmpty vw) $
                      set (u,w) (uw :+ (uv :. vw))
```

For each block label, v, in the depth-first ordering, `elimLoop` first adds the path $(v \rightarrow^* v)^*$ to the set of paths. Then, for each block u which comes after v in the ordering, we add to path expression for (u, v) the path $u \rightarrow^* v \rightarrow^* v$ which captures cycles starting at v. Finally, for each block, w which comes after v in the ordering, we add to path expression for (u, w) the path $u \rightarrow^* v \rightarrow^* w$, which captures paths which

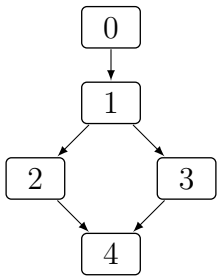


Figure 6.6: Example: a conditional statement.

contain the intemediate block v. The guards of the form `when (notEmpty ...)` are an optimization.

Again, the correctness of this algorithm has been proved by Tarjan (1981). Specifically, Tarjan proved the following theorem:

Theorem 5 (Correctness of `eliminate`)**.** *For a given control-flow graph, G, the function* `eliminate` *produces a decomposed path map of the blocks of G with respect to a topological ordering of G.*

6.4.4 Examples

We will now examine the operation of `solve` through a couple of simple examples. The first example is a conditional statement. The control-flow graph for the first example is shown in Figure 6.6. In this example, execution starts at the entry block 0, and proceeds unconditionally to block 1. From block 1, control can flow either to block 2 or block 3 based on a condition (not shown here). Finally, the control flows to block 4. For our example, we will use block 0 as the source block. There are five paths with source block 0 going to each of the blocks in the graph. We will look at how these paths are constructed through the execution of `solve`.

The first step of the algorithm is to generate a path sequence using `generateSequence`. This function begins by performing decomposition with `eliminate`. Within `eliminate` the path map is initialize with the atomic paths using `elimInit`. After `elimInit`, the path map is initialized and contains the entries below:

$$(0,1) \mapsto \emptyset \cup \gamma_{(0,1)}$$

$$(1,2) \mapsto \emptyset \cup \gamma_{(1,2)}$$

$$(1,3) \mapsto \emptyset \cup \gamma_{(1,3)}$$

$$(2,4) \mapsto \emptyset \cup \gamma_{(2,4)}$$

$$(3,4) \mapsto \emptyset \cup \gamma_{(3,4)}$$

Notice we have not simplified these paths at this point to make it easier to follow the algorithm.

Next, we adjust all of the paths inside of `elimLoop`. In this simple case, this has the net effect of adding empty paths (Λ) for each block to itself. The path map returned from `eliminate` is show below. In this case, we also show the simplified

paths for each entry.

$$
\begin{aligned}
(0,0) &\mapsto \emptyset^* && \mapsto && \Lambda \\
(0,1) &\mapsto \left(\emptyset \cup \gamma_{(0,1)}\right) \cdot (\emptyset^*) && \mapsto && \gamma_{(0,1)} \\
(1,1) &\mapsto \emptyset^* && \mapsto && \Lambda \\
(1,2) &\mapsto \left(\emptyset \cup \gamma_{(1,2)}\right) \cdot (\emptyset^*) && \mapsto && \gamma_{(1,2)} \\
(1,3) &\mapsto \left(\emptyset \cup \gamma_{(1,3)}\right) \cdot (\emptyset^*) && \mapsto && \gamma_{(1,3)} \\
(2,2) &\mapsto \emptyset^* && \mapsto && \Lambda \\
(2,4) &\mapsto \left(\emptyset \cup \gamma_{(2,4)}\right) \cdot (\emptyset^*) && \mapsto && \gamma_{(2,4)} \\
(3,3) &\mapsto \emptyset^* && \mapsto && \Lambda \\
(3,4) &\mapsto \left(\emptyset \cup \gamma_{(3,4)}\right) \cdot (\emptyset^*) && \mapsto && \gamma_{(3,4)} \\
(4,4) &\mapsto \emptyset^* && \mapsto && \Lambda
\end{aligned}
$$

Finally, with the above path map as input, we run the main `solve` algorithm with source block 0. The final result is shown below:

$$
\begin{aligned}
&(0,0) \mapsto \Lambda \\
&(0,1) \mapsto \gamma_{(0,1)} \\
&(0,2) \mapsto \gamma_{(0,1)} \cdot \gamma_{(1,2)} \\
&(0,3) \mapsto \gamma_{(0,1)} \cdot \gamma_{(1,3)} \\
&(0,4) \mapsto \left(\left(\gamma_{(0,1)} \cdot \gamma_{(1,3)}\right) \cdot \gamma_{(3,4)}\right) \cup \left(\left(\gamma_{(0,1)} \cdot \gamma_{(1,2)}\right) \cdot \gamma_{(2,4)}\right)
\end{aligned}
$$

Looking at the final result, we see that for block 4, there are two paths from the source 0 joined with union. The first is the path $0 \to 1 \to 3 \to 4$, and the second is $0 \to 1 \to 2 \to 4$. Suppose we want to compute the gating function for block

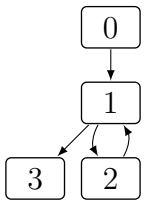


Figure 6.7: Control-flow graph for a simple loop

4. We can do this by interpreting the path expression using our alternate boolean interpretation[2]. If we suppose that the condition splitting the control flow is c, then this interpretation is:

$$\begin{aligned}
&[\![\big(\big(\gamma_{(0,1)} \cdot \gamma_{(1,3)}\big) \cdot \gamma_{(3,4)}\big) \cup \big(\big(\gamma_{(0,1)} \cdot \gamma_{(1,2)}\big) \cdot \gamma_{(2,4)}\big)]\!]_b \\
=&((true\ \wedge \neg c) \wedge true) \vee ((true\ \wedge c) \wedge true) \\
=&\neg c \vee c
\end{aligned}$$

Example 2. For our second example we will look at a simple loop. Consider the control-flow graph shown in Figure 6.7. As with the first example, `elimInit` simply constructs the atomic paths for each edge in the control-flow graph. However, because of the loop in this example, `eliminate` does some interesting work. The configuration

[2]Actually, we would use 1 as a source vertex, not 0, but the result is the same in this case.

of the path map after `eliminate` is show below.

$$
\begin{array}{lcl}
(0,0) \mapsto \emptyset^* & \mapsto & \Lambda \\
(0,1) \mapsto \emptyset \cup \gamma_{(0,1)} & \mapsto & \gamma_{(0,1)} \\
(1,1) \mapsto \emptyset^* & \mapsto & \Lambda \\
(1,2) \mapsto \emptyset \cup \gamma_{(1,2)} & \mapsto & \gamma_{(1,2)} \\
(1,3) \mapsto \emptyset \cup \gamma_{(1,3)} & \mapsto & \gamma_{(1,3)} \\
(2,1) \mapsto \left(\emptyset \cup \gamma_{(2,1)}\right) \cdot \emptyset^* & \mapsto & \gamma_{(2,1)} \\
(2,2) \mapsto \left(\emptyset \cup \left(\left(\left(\emptyset \cup \gamma_{(2,1)}\right) \cdot \emptyset^*\right) \cdot \left(\emptyset \cup \gamma_{(1,2)}\right)\right)\right)^* & \mapsto & \left(\gamma_{(2,1)} \cdot \gamma_{(1,2)}\right)^* \\
(2,3) \mapsto \left(\emptyset \cup \left(\left(\left(\emptyset \cup \gamma_{(2,1)}\right) \cdot \emptyset^*\right) \cdot \left(\emptyset \cup \gamma_{(1,3)}\right)\right)\right) \cdot \emptyset^* & \mapsto & \gamma_{(2,1)} \cdot \gamma_{(1,3)} \\
(3,3) \mapsto \emptyset^* & \mapsto & \Lambda
\end{array}
$$

As before, we have show the initial path expression and the simplified versions of the path expressions for each edge in the control-flow graph, and for the paths $n \to^* n$ for each block n. Notice, that the initial path expression for $(1,1)$ is Λ just as in the previous example. The path expression for $(2,2)$, however, is $(2 \to 1 \to 2)^*$ indicating that a loop is possible. The cycle appears for $(2,2)$ and not $(1,1)$ because we are using a topological ordering to do the decomposition.

Using the path map above, we can evaluate `solve` using an initial block of 1. The

resulting path map is shown below.

$$(1,0) \mapsto \emptyset$$

$$(1,1) \mapsto \Lambda \cup \left(\left(\gamma_{(1,2)} \cdot \left(\gamma_{(2,1)} \cdot \gamma_{(1,2)}\right)^*\right) \cdot \gamma_{(2,1)}\right)$$

$$(1,2) \mapsto \gamma_{(1,2)} \cdot \left(\gamma_{(2,1)} \cdot \gamma_{(1,2)}\right)^*$$

$$(1,3) \mapsto \gamma_{(1,3)} \cup \left(\left(\gamma_{(1,2)} \cdot \left(\gamma_{(2,1)} \cdot \gamma_{(1,2)}\right)^*\right) \cdot \left(\gamma_{(2,1)} \cdot \gamma_{(1,3)}\right)\right)$$

First, notice that the path expression for $(1,1)$, representing all of the paths from block 1 to itself, is either the empty path, or the path which begins with an edge from 1 to 2, followed by zero or more cycles between 2 and 1, and ends with an edge from 2 to 1. The empty path Λ is important because it allows us to properly compute the path expression for $(1,3)$, which includes the simple path $1 \to 3$.

6.4.5 Simple Optimizations

Our simple decomposition technique, `eliminate`, has a complexity which depends on the sparsity of the matrix of edges for the control-flow graph. For a given control-flow graph (N, E), the worst case time complexity is $\mathcal{O}(|N|^3+|E|)$. We can improve on this by combining `eliminate` with a partitioning of the graph into strong components. We can apply `eliminate` to each sub-graph to compute a set of path sequences for each component, and combine the path sequences using the topological ordering of the components. This technique is more or less effective depending on the type of graphs we are considering.

For example, suppose our graph is acyclic. In this case, we can obtain a path sequence by topological ordering alone. Put another way, in an acyclic graph, each block is a strong component. Applying `eliminate` to a single block just returns a

path sequence containing this block. So, we may use the topological ordering of blocks as our path sequence.

A more realistic example is to take advantage of the directed nature of control-flow graphs. Because our graphs are directed, we can compute the strongly-connected components of the graph and treat each strong component independently. Unfortunately, because many control-flow graph have few strongly connected components (compared to the number of blocks), this technique produces only modest improvements. In the next section we will look at a much more powerful decomposition technique based on dominators.

6.5 Decomposition using dominators

In this section we will present the final form of our algorithm for computing gating functions. This version of the algorithm decomposes the graph using the dominator tree. The algorithm performs a depth-first traversal of the dominator tree, and at each step processes the sub-graph containing the children of the tree node. In this way, we only process very small sections of the graph at a time and then combine the results, resulting in much better time complexity. Initially, we will assume our control-flow graphs are reducible. In the next section we will show how to modify the algorithm to handle arbitrary graphs.

For this version of the algorithm, we will change the intermediate representation of gating functions. Each gating function is a path expression which can be thought of as a tree built using as tree nodes the constructors of the `PathExpr` type. Initially, we only have atomic path expressions which are trees of height one. As the algorithm

progresses, we will compose path expressions creating larger trees. Therefore, we will represent the intermediate state of our gating functions as a forest of trees. Each tree in the forest is a path expression representing paths from the root of the tree to the leaves.

Conceptually, the forest is a pair of finite maps. One map captures the structure of the tree by associating with each block label the label of its parent in the forest. The second map tracks the gating function from the root of the tree to a block by associating with each block label a gating function. In code, we use the abstract type `Forest` with the following associated functions:

```
data Forest — abstract type
emptyForest :: Forest
fetchNode   :: Label → Forest → (Label,Gate)
setParent   :: Label → Label → Forest → Forest
setGate     :: Gate → Label → Forest → Forest
```

The function `fetchNode` returns both the parent label and gating function associated with a given block label. The two functions `setParent` and `setGate` update a forest by changing the parent of gating function associated, respectively, with a block label. As the forest is modified to include larger trees, we will simplify the gating functions when possible. Our forest is therefore an instance of path compression (Tarjan, 1975). We will present the path compression functions in detail in Section 6.5.2.

Because we will be working on incomplete graphs, we will need a variation of the `solve` algorithm for sub-graphs. Like `solve`, the new algorithm (called `solveChildren`) takes in a path sequence and source and sink labels and computes a decomposed path map. As before, we will build the path map by updating a state variable containing the current path map. However, we must be careful to reset the path map for each

sub-graph. For this, and for tracking the current forest of gating functions, we use a state monad to structure the algorithm.

6.5.1 Gating Monad

The gating algorithm runs within a monad. This monad tracks the current forest of gating functions, the state for the solve algorithm, and provides easy (and fast) access to the basic blocks. The monad is defined using the parametrized type `M` shown below.

```
type Blocks = A.Array Label GBlock
data M a = M {
  runM :: Blocks → PathMap → Forest → (PathMap,Forest,a)
}
```

The type `Blocks` is an array of basic blocks indexed by block labels, which are small integers. This array gives us a quick way to find a basic block from its label. The monad `M` is defined as the set of functions from the block array `Blocks`, the state of the solve algorithm `PathMap`, and the state of the forest of gating functions `Forest` to a tuple. The result tuple contains a possibly modified solve state, a possibly modified forest of gating functions, and a computation specific result.

The monad `M` is a combination environment state monad. The `Blocks` array is used as an environment which cannot be modified, but it is available to all computations. Both `PathMap` and `Forest` are state which can be read and written by computations. The different roles of the environment and state can be seen in the implementation of the monad.

```
instance Monad M where
```

```
return x = M $ λe s f → (s,f,x)
m >>= k  = M $ λe s f → let (s',f',x) = runM m e s f in
                        runM (k x) e s' f'
```

In the definition of `M` we see that when two computations are sequenced using the bind operator (`>>=`) the first computation is given the current environment and state variables. The second computation is given the same, unmodified environment and the possible modified state variables.

In order to use the monad `M`, we need to provide some primitive operations on the monad for accessing the internal environment and state. The environment can be accessed by creating a function of the right type which copies the environment to the result position.

```
blocks :: M Blocks
blocks = M (λe s f → (s,f,e))

blockOf :: Label → M GBlock
blockOf l = do bs ← blocks; return (bs ! l)
```

The function `blocks` copies the environment (the first argument) to the result position (the third element of the result tuple) leaving the state unchanged. A simple helper function, `blockOf` finds the basic block associated with a label using the environment and an array access.

The internal state of the monad is access in the same way as the environment. For instance, the current forest of gating function can be read using the `forest` function defined below.

```
forest :: M Forest
forest = M (λe s f → (s,f,f))
```

```
setForest :: Forest → M ()
setForest f = M (λe s _ → (s,f,()))
```

Since the forest is internal state and can be modified, we also provide `setForest` which replaces the current forest with a new one. The same treatment is needed for the solve algorithm state.

```
solveState :: M PathMap
solveState = M (λe s f → (s,f,s))

setSolveState :: PathMap → M ()
setSolveState s = M (λe _ f → (s,f,()))
```

The solve algorithm we use here is similar to the `solve` function from the previous sections. In order to make this similarity clear, we define the functions `get` and `set` which perform the same job as the so named functions from the previous sections. However, our implementations differ in that they use the solve state of our new monad.

```
get :: PathName → M Gate
set :: PathName → Gate → M ()
get n   = solveState >>= λm → return (fetch m n)
set n g = solveState >>= λm → setSolveState (store m n g)

reset :: M ()
reset = setSolveState emptyPathMap
```

As previously mentioned, because our solve algorithm is called many times on subgraphs of our control-flow graph, we need to be careful to reset the internal state of the solver. We provide a primitive `reset` which does this by replacing the solver state with an empty path map.

Finally, we need a function to run our monad computations once we have constructed them with return, bind, and our monadic primitives. Following convention,

this function is called `evalM`, and is shown below.

```
evalM :: [GBlock] → M a → a
evalM cfg m =
    let (_,_,x) = runM m env emptyPathMap emptyForest
    in x
 where
   bounds = (0, length cfg + 1)
   blks   = sortBy (λb1 b2 → compare (label b1) (label b2)) cfg
   env    = A.array bounds [(label b, b) | b ← blks]
```

The `evalM` function uses `runM` to extract the function from the monad `M`. This function is then run using an initial environment, an empty path map, and an empty forest. The initial environment is an array of blocks in label order built using the built-in `Array` module.

6.5.2 Path Compression

Now that we have the monad `M`, we can implement monadic versions of our functions for manipulating the current forest of gating functions. Using these we can then implement path compression on our trees. To begin, we implement the functions `parent` and `gate` which return the current parent or the gating function for a given label, respectively.

```
parent :: Label → M Label
gate   :: Label → M Gate
parent l = liftM (fst ∘ fetchNode l) forest
gate   l = liftM (snd ∘ fetchNode l) forest
```

Both of these function use `fetchNode` (defined earlier) to find the tree node corresponding the a given label. The forest we use is the current forest returned from the monadic primitive `forest`.

Following (Tarjan, 1979), we implement path compression by defining an interface for accessing our forest of gating functions. The interface contains four functions listed below with their types.

```
initialize :: Label → M ()
update     :: Label → Gate → M ()
link       :: Label → Label → M ()
eval       :: Label → M Gate
```

The call `initialize l` sets the parent of `l` to 0, and sets the gating function of `l` to Λ. The call `update l g` sets the gating function associated with label `l` to `g`. The call `link p l` sets the parent of `l` to be `p`, thus relocating the tree rooted at `l` to be a child of `p`. Finally, `eval l` returns the gating function which represents all of the paths from the root of `l`'s tree to `l`.

The implementation of the first three of the interface functions is straight-forward. We implement `update` and `link` using the operations on forests and our monadic primitives.

```
update l g = do f ← forest; setForest (setGate g l f)
link   p l = do f ← forest; setForest (setParent p l f)
```

The `initialize` function can then be implemented in terms of these two functions.

```
initialize v = do link 0 v; update v Lambda
```

The path compression is done within the implementation of `eval`. If the block label is not for the entry node (0), then we perform path compression the label, and then return the gating function.

```
eval :: Label → M Gate
eval 0 = return Lambda
```

```
eval v = do compress v; gate v
```

Path compression starts at a label, `v`, and tries to compress the tree towards the root. The compression is done by building a gating function from `v`'s grandparent to `v` and re-rooting `v` to its grandparent.

```
compress :: Label → M ()
compress v = do p  ← parent v
                pp ← parent p
                when (pp ≠ 0) $
                 do compress p
                    gv ← gate v
                    gp ← gate p
                    update v (simplify (gp :. gv))
                    link v pp
```

Note that in addition to reducing the height of the tree leading to `v`, we also simplify the gating function for `v`. Therefore, compression helps us to maintain minimal trees and minimal gating functions.

6.5.3 Block Processing Order

In order for our algorithm to work properly, we must be careful to process blocks in the correct order. There are several functions we use for building different orderings and sets of blocks. Our algorithms is structured as a depth-first traversal of the dominator tree. We do this by creating an ordering of blocks which represents this ordering and then processing the ordered blocks in a loop. The ordering is created using the `idomOrdering` function shown below.

```
idomOrdering :: M [Label]
idomOrdering = liftM reverse (next [] 0)
 where
```

```
    next ls l = childrenOf l >>= foldM next (l:ls)

childrenOf :: Label → M [Label]
childrenOf l = liftM children (blockOf l)
```

We use and define a small helper function, `childrenOf`, which is a monadic version of the functions for accessing a block's children in the dominator tree.

As we process each block, we must differentiate between control-flow edges which are also edges in the dominator tree, and edges which are not. The functions `tree` and `non_tree` select edges which are and are not in the dominator tree respectively.

```
tree     :: Label → M [(Label,Label)]
non_tree :: Label → M [(Label,Label)]


tree     l = do b ← blockOf l; return [ (x,l) | x ← cfgpre b
                                              , x == idom b ]
non_tree l = do b ← blockOf l; return [ (x,l) | x ← cfgpre b
                                              , x ≠ idom b ]
```

Note that for each block, all of its control-flow in-edges are contained in the union of the sets created by `tree` and `non_tree`.

Finally, in order to use our solve algorithm, we have to create a path sequence for a sub-graph rooted at a specific block. At each step we will process a block and its immediate children in the dominator tree. Therefore, we need a path sequence which includes these blocks. If our graph is reducible, then we can use a topological ordering of the blocks contained in the sub-graph.

```
subsequence :: Label → M PathSequence
subsequence l =
    do cs ← childrenOf l
       bs ← liftM (filter (λb → label b `elem` l:cs) ∘ A.elems) blocks
       let es = [ (l,x) | x ← topOrder bs ]
```

```
        pm ← foldM f emptyPathMap bs
        return $ (es, pm)
  where
    f m b = liftM (store m (l,label b))
                  (pathExprOf l (label b))
```

The function `subsequence` computes a path sequence for a sub-graph assuming our control-flow graph is reducible. We use the list of blocks stored in the environment filtered to contain only the block of interest and its immediate children. From this list we build a simple path map containing the atomic paths. Together, the ordering and the path map consittute a path seqeunce which can be used for reducible graphs. We will return to the issue of irreducible graph in Section 6.6.

6.5.4 Main Algorithm

The main algorithm, `decompose`, is shown in Figure 6.8. After the execution of `decompose`, the forest of path expressions will contain for every block v, a non-redundant path expression representing all paths from `idom`(v) to v. This is exactly the information we need to construct our gating function according to Theorem 2.

The main algorithm begins by building a depth-first sorting of the blocks at line 3. Then, on line 4, we initialize the forest with initial values for each block. Lines 5 through 18 contain the loop which processes each node in the depth-first ordering. The loop begins with lines 6-8 which assign local variables to the basic block, children, and immediate dominator for the current block label `u`. Then, there are three sections contained on lines 9-11, lines 12-13, and lines 14-16. The first section initializes all of the non-tree nodes of the current block's children by calling `evalEdge`. For irreducible graphs, `evalEdge` is easily implemented in terms of `eval`.

```
1   decompose :: M ()
2   decompose =
3     do { labels ← idomOrdering
4        ; forM_ labels initialize
5        ; forM_ labels $ λu →
6          do { blk ← blockOf u
7             ; let cs_u   = children blk
8             ; let idom_u = idom blk
9             ; forM_ cs_u $ λv →
10                do nt ← non_tree v
11                   forM_ nt evalEdge
12            ; subseq ← subsequence u
13            ; pathMap ← solveChildren subseq idom_u u
14            ; forM_ cs_u $ λv →
15                do update v (fetch pathMap (idom_u, u))
16                   link u v
17            ; return ()
18            }
19       ; return ()
20       }
```

Figure 6.8: Main Algorithm

```
evalEdge :: (Label,Label) → M Gate
evalEdge (h,t) = eval t
```

The second section builds a path sequence for the graph containing the current block and its children. This path sequence is used to call `solveChildren` which computes a path map for these blocks. Finally, the third section updates our forest with the information contained in the path map computed by `solveChildren`.

Solving Sub-graphs

The algorithm for solving a sub-graph is similar to our original algorithm `solve`. The sub-graph algorithm is given two labels: `root` and `lbl`. The second label, `lbl` is the block we are working on, and the first label is the root of the tree for this label's gating function. The sub-graph version begins by resetting the internal path map to an empty map.

```
solveChildren :: PathSequence → Label → Label → M PathMap
solveChildren pathSeq root lbl =
  do reset
     cs ← childrenOf lbl
     solveChildrenInit root cs
     solveChildrenLoop pathSeq root lbl
     solveState
```

Then, for each child of `lbl` in the dominator tree, we initialize the path map entries corresponding to the paths from `root` to those labels. Finally, we call `solveChildrenLoop` on the path sequence for the sub-graph of interest.

To initialize the path map for the children, we first get the set of control-flow in-edges to each child which are also in the dominator tree using the `tree` function.

Then, for each of these edges we compute an atomic path expression and union this to the existing path expression for the child.

```
solveChildrenInit :: Label → [Label] → M ()
solveChildrenInit s csu =
  forM_ csu $ λv →
    do set (s,v) Empty
       tr ← tree v
       forM_ tr $ λ(h,t) →
         do sv ← get (s,v)
            ht ← pathExprOf h t
            set (s,v) (sv :+ ht)
```

After `solveChildrenInit`, there is an atomic path expression for each edge between the children and their immediate dominators. Note, in our case, there can be only one such edge for each child. However, for general graphs this may not be the case. The above function could be simplified, in our case, to simply build atomic path expressions for the dominator tree edges which appear in the control-flow graph.

Finally, the main solving loop, `solveChildLoop`, is shown below.

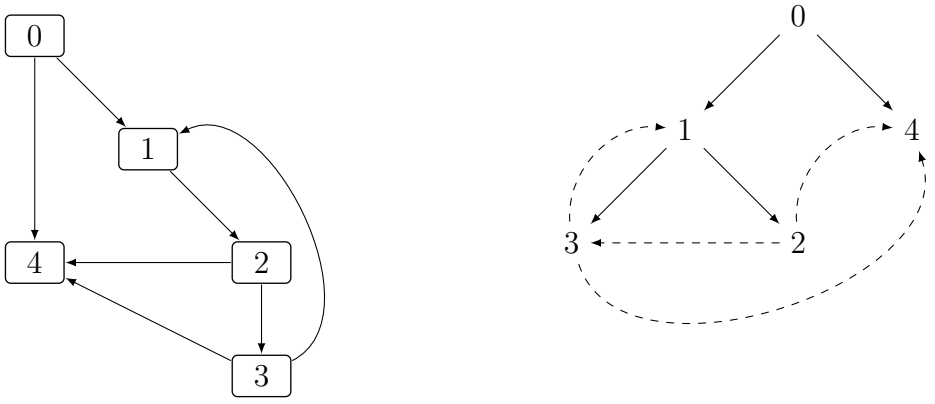
```
solveChildrenLoop :: PathSequence → Label → Label → M ()
solveChildrenLoop (labels,pm) s u =
  forM_ labels $ λ(w,x) →
    do p ← get (w,x)
       sw ← get (s,w)
       sx ← get (s,x)
       if x == w
         then set (s,w) (sw :. p)
         else set (s,x) (sx :+ (sw :. p))
```

This function is almost identical to the corresponding code in `solve`. For each edge (w, x) in the path sequence, we check if $w = x$. If so, then we append $w \rightarrow^* x$ to the path $s \rightarrow^* w$ representing a loop. Otherwise, we add the path $s \rightarrow^* w \rightarrow^* x$ to the path expression for (s, x).

The algorithm presented in this section is only valid for reducible graphs. However, the algorithm is structured so that it is easy to extend it to irreducible graphs. In particular, we only need to revisit how we construct the path sequence and initialize the non-tree edges. In the next section, we will look at why this algorithm does not work for irreducible graphs and extend it to handle arbitrary graphs.

6.6 Irreducible graphs

Consider the control-flow graph shown in Figure 6.9a. This graph is irreducible because of the edge from block 2 to block 4. Figure 6.9b shows the associated dom-



(a) An irreducible graph. (b) Associated Dominator Tree.

Figure 6.9: Irreducible graph and its dominator tree.

inator tree. We have included dashed edges to indicate control-flow edges which do not appear in the tree (these are the "non-tree" edges).

Because of the irreducibility, the algorithm presented in the previous section would not compute the correct result for this graph. There are two problems that we encounter: a technical problem involving the order in which edges are processed, and a more basic problem with the decomposition scheme. The basic problem is that

Block	Partial path sequence	Solved edges	Non-tree edges	Tree edges
3	[]	{}	{(2,3)}	{}
2	[]	{}	{}	{(1,2)}
1	[(1,2)]	{(1,2)}	{(3,1)}	{(0,1)}
4	[]	{}	{(2,4),(3,4)}	{(0,4)}
0	[(0,1),(0,4)]	{(0,1),(0,4)}	{}	{}

Table 6.1: Sequences used for graph in Figure 6.9

the sub-graphs we divide the graph into do not satisfy our criteria for decomposition. Recall, the idea is to divide the graph into components which can be treated separately, and then to combine the results by composing the path sequences for the sub-problems. However, with this graph, our components have multiple entry points. Therefore, we cannot combine the result of the sub-problems by simple composition. A solution to this basic problem lead us to our final algorithm, however it is useful to consider the ordering problem to gain some intuition.

Table 6.1 shows the sets and sequences used to process the irreducible graph in Figure 6.9. The first column lists the blocks of the graph in the order they are processed by the main algorithm. The second column shows the path sequence computed by `sequence` for the sub-graph rooted at each block. The third column shows the edges that are solved for in each iteration of the algorithm. Finally, the fourth and fifth columns show the sets computed by `tree` and `non_tree`.

The main algorithm starts by initializing the non-tree edges for the children of the current block, and then calling `solveChildren` for the associated path sequence. Solve initializes the tree edges for the children of the current block, and proceeds to

the main solving loop. After solve, the "solved edges" are stored in the result path map. Therefore, if we read Table 6.1 from top-to-bottom and left-to-right, we see the order in which the edges of the graph are processed.

We can see from Table 6.1 that our algorithm cannot possibly compute the correct results just by inspecting the order the edges are processed. For example, the path expression for edge $1 \rightarrow 2$ is computed before the edge $3 \rightarrow 1$ has even been initialized. With no information about edge $3 \rightarrow 1$, the path expression for edge $1 \rightarrow 2$ must be incomplete. Similar problems can be spotted for other edges using the table.

Clearly, our algorithm fails for this irreducible graph. Although it may not be obvious from what we have presented, the ordering problem is merely a symptom of the more basic problem; our sub-graphs may have multiple entries or exits. To repair our algorithm we will transform our graph so that each sub-graph we consider has on entry and one exit.

6.6.1 The Derived Graph

The flaw in our algorithm lies in the fact that the sub-graphs induced by the blocks and their children are not simple components with only one entry and one exit. To fix this problem, we define the notion of *dominator strong components*. The dominator strong components for a reducible graph are simply the basic blocks of the graph. For an irreducible graph, the dominator strong components must allow us to compute a partial path sequence which can be combined with other partial path sequence to produce a complete sequence for the graph. Precisely, the dominator strong components are defined as the strong components of the *derived graph*, which

is defined below.

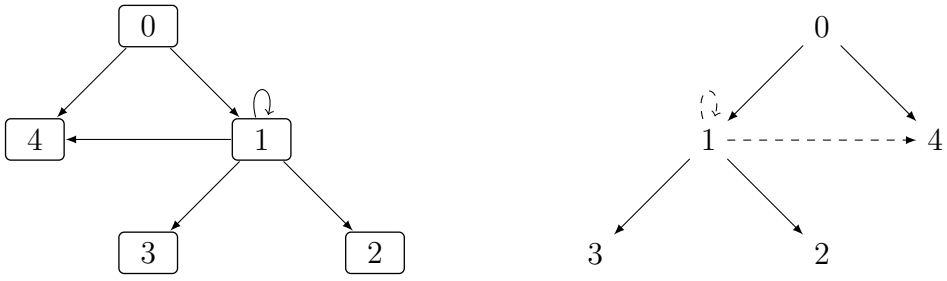
Definition 17. *The* **derived graph** *for a graph $G = (N, E)$, is a graph with nodes N and edges $\widetilde{E}$ such that $(u, v) \in \widetilde{E}$ if either:*

- $(u, v) \in E \wedge u = \texttt{idom}(v)$, *or*
- $u \neq v$, *and* $\exists (u', v) \in E$ *such that* $\texttt{idom}(v) \rightarrow u' \rightarrow^* v$.

There are two slightly different definitions for the derived graph appearing in the literature. The first definition appears in Tarjan (1979), which is the definition we have used. A second definition appears in Georgiadis and Tarjan (2005) which includes fewer edges, but it is not materially different. The following theorem allows us to keep our depth-first algorithm structure which using the derived graph when needed.

Theorem 6 (Georgiadis and Tarjan (2005))**.** *For any graph G and its derived graph $\widetilde{G}$, T is the dominator tree of G if and only if T is also the dominator tree of $\widetilde{G}$.*

Figure 6.10 shows the derived graph for our example. Indeed, the associated dominator tree is unchanged, but the set of "non-tree edges" are not the same. We



(a) Derived graph. (b) Associated Dominator Tree.

Figure 6.10: Graph derived from Control-flow Graph in Figure 6.9

can use the derived graph to run our algorithm. To do this we will use the derived graph to compute the path sequences we need to solve for edges. In this case, each sub-graph we consider will be a simple component, and we can concatenate our results with other components. Also, because the dominator tree is unchanged we can be sure that we have processed the edges in the correct order. Note however, that after we compute a path sequence we must be careful to use path expressions from our original graph for all of the derived edges. We will now look at how this is accomplished with minor modification to our algorithm.

6.6.2 Modifications to Main Algorithm

To handle arbitrary graphs, we will make modifications to the functions `subsequence` and `evalEdge`. The changes to `subsequence` are where the changes to the algorithm take place; the changes to `evalEdge` allow us to keep track of some additional information needed by the new `subsequence`. Th additional information we need to track is the path expressions which correspond to the edges in the derived graph.

For each edge in the derived graph which does not appear in the original graph we need to keep track of the path expression from the original graph that the new derived edge represents. Put another way, for each edge (h', t) which is derived from edge (h, t) we need to associate the path expression for the original path $h \rightarrow^* t$ with the derived edge (h', t). This additional information is tracked within our monad using a new state parameter. The new state parameter is a `PathMap` for the derived edges with the following accessor functions:

```
getDerived   :: M PathMap
```

```
storeDerived :: PathName → Gate → M ()
```

The function `getDerived` returns the current path map for the derived edges. The function `storeDerived` associated a path expression with an edge. The path expressions for the derived edges are stored each time we evaluate an edge with `evalEdge`. Our first modification is to change `evalEdge` to record these path expressions.

```
evalEdge :: (Label,Label) → M Gate
evalEdge (h,t) =
    do p ← eval t
       de ← derivedEdge h t
       storeDerived de p
       return p
```

Just like the original version, the new `evalEdge` implementation returns the result of `eval` on the second component of the input edge. However, this version also computes the derived edge, `de`, and associates the path expression with this edge for later use. Note, the path expressions associated with the derived edges are stored separately so they do not effect the other parts of the algorithm.

To compute the derived edges, we refer to Definition 17. The function `evalEdge` implements this definition.

```
derivedEdge :: Label → Label → M (Label,Label)
derivedEdge h t =
    do tBlock ← blockOf t
       if h == idom tBlock
        then return (h,t)
        else do ss ← siblings t
                h' ← pathTo siblings [t]
                return (h',t)
```

The siblings of a block are those blocks with a common immediate dominator.

```
siblings :: Label → M [Label]
siblings l = do cs ← blockOf l >>=
                     childrenOf ∘ idom
                return [ b | b ← cs, b ≠ l ]
```

To find the block u' from Definition 17 we begin at the destination block and work backwards through the control-flow edges looking for a sibling. This is done by the function `pathTo` shown below.

```
pathTo [] _  = fail "no siblings"
pathTo ss [] = fail "no sibling found"
pathTo ss ls =
    case find (`elem` ss) ls of
      Nothing → do lls' ← mapM cfgpreOf ls
                   pathTo ss (concat lls')
      Just l  → return l
```

With this bookkeeping in place we can now give an alternate definition of `subsequence`. Our new implementation of `subsequence` uses `generateSequence` to produce a path sequence for the derived sub graph containing the block in question and children.

```
subsequence :: Label → M PathSequence
subsequence l =
  do cs ← childrenOf l
     bs ← liftM (filter (λb → label b `elem` l:cs) ∘ A.elems) blocks
     dg ← derivedGraph bs
     substDerived (generateSequence dg)
 where
   derivedGraph :: [GBlock] → M [GBlock]
```

The function `derivedGraph` uses `derivedEdge` to replace the edges in the sub-graph with their derived counterparts. Once we have this derived graph we can generate a path sequence for this sub-graph knowing it is a single entry component of the derived graph. Then, to use this sequence we must substitute the path expressions for the

original graph where the derived edges appear in the sequence. This last step is done by `substDerived`.

```
substDerived :: PathSequence → M PathSequence
substDerived (es,pm) =
    do dm ← getDerived
       let pm' = mapGates pm (lookupGate dm)
       return (es,pm')
 where
   getDerived :: M PathMap
   mapGates   :: PathMap → (PathName → Gate → Gate) → PathMap
   lookupGate :: PathMap → PathName → Gate → Gate
```

As we can see from the implementation, `substDerived` simply maps the substitutions in the derived path map over the path expressions contained in the path seqeunce. We use the helper functions `mapGates` and `lookupGate` to perform the substitution.

Correctness. This completes our algorithm for computing gating functions for arbitrary graphs. The algorithm is a variant of the single-source path-expressions problem for irreducible graphs. As such, its correctness is a corollary to the correctness of Tarjan's original algorithm.

Theorem 7 (Correctness of `decompose`)**.** *For a graph $G = (N, E)$, the function* `decompose` *computes a path map, P, such that $\forall v \in N$, $P(\mathit{idom}(v), v)$ is an irreducible path sequence from $\mathit{idom}(v)$ to v.*

Proof. This theorem is a direct corollary of Lemma 8(i) found in Tarjan (1979). □

Complexity. The running time of the algorithm presented here depends on two factors: the complexity of `decompose` and the complexity of computing path sequences with `generateSequence`. For a control-flow graph $G = (N, E)$, the total running time

of the algorithm is:

$$\mathcal{O}\left(|E|\, ln(|N|) + \ell + \Sigma_{v=1}^{|N|} |\{(u,v) \in E \mid u > v\}| \times |\{(v,w) \in E \mid w > v\}|\right)$$

where ℓ is the total length of the path sequences for the derived graph. The first term:

$$\mathcal{O}\left(|E|\, ln(|N|)\right)$$

comes from the main algorithm `decompose`. It is possible to use a more aggressive form of path compression (called *stratified* path compression) to achieve a near-linear running time of:

$$\mathcal{O}\left(|E|\, \alpha(|E|\,, |N|)\right)$$

where α is proportional to the inverse Ackermann function.

The second term comes is the time required to compute the path sequences with `generateSequence`. This bound can also be improved by sophisticated numerical analysis techniques (Bunch and Rose, 1974; Rose et al., 1980). However, in practice, we expect our sub-graphs to be small (only a few graph nodes) so this contribution should be small. One notable exception is very large switch-statements which have blocks with many children in the dominator tree. In this case, the computation of the path sequence for the switch sub-graph can be long.

Chapter 7

Experimental Evaluation

In this chapter we present an experimental evaluation of our translation validation prototype. Our evaluation investigates two prototypical use cases: a compiler developer verifying the correctness of new optimizations, and a compiler user who wants to use validated optimizations. For input programs, we use the Spec CPU benchmarks and the CompCert regression test library. Our analysis looks at the number of functions our tool is able to validate using different configurations of rules, and at the overall effect on run-time when used as a validated optimizer.

The goal of our experimental evaluation is to determine if LLVM M.D. is capable of being a *practical* software development tool. Whether of not a tool is practical depends on how the tool is used and by whom. We will consider two prototypical users: a compiler developer and a compiler user. For each of these users, "practical" will take on slightly different meanings. For the compiler user, practical means:

- The validation tool is able to validate most of the correct optimizations, and does not significantly degrade the run-time performance of programs.
- The validation tool does not take too long to run, and can be integrated into a normal build process.

For a compiler developer, "practical" also means:

- The validation tool is easy to configure.
- The validation tool does not require extensive configuration.
- The validation tools output is easy to understand, and can be used to correct compiler bugs or modify the configuration.

In our experimental evaluation, we aim to measure the effectiveness of our prototype validator with respect to the above criteria. For the compiler user, we expect the compiler to be run once with a pipeline of optimizations. Therefore, we will measure the effectiveness of our tool when the inputs are an unoptimized program and its counterpart after a series of optimizations have been applied. We will attempt to answer the following questions:

1. How effective is the tool for an aggressive pipeline of optimizations?

2. What is the impact on running time if we only use validated optimizations?

3. How long does validation take compared to compile time?

Our measure of effectiveness is simple: the fewer false alarms our tool produces, the more optimized the code will be. Put another way, assuming the optimizer is always correct, what is the cost of only using validated code?

For the compiler developer we expect the compiler to be run with a single optimization at a time. After performing a single optimization, we expect the validator to be run and the output used to detect bugs or new configuration rules that may be needed to account for the new optimization. We are interested in the following additional questions:

4. How effective is the tool optimization-by-optimization, and does this correlate to the performance on a pipeline of optimizations?

5. What is the effect of adding new normalization rules on the system?

6. How hard is it to modify the configuration to account for new false alarms?

7. How easy is it to interpret the output of the validation tool?

We believe these questions directly address the notion of practical we have described, and can determine if our validation tool can be useful for compiler users and developers.

7.1 Experimental Setup

Our testing uses two sets of benchmark programs. The first set contains large programs to stress our validator. These are the pure C programs of the SPECCPU 2006 (SpecCPU, 2006) benchmark[1]. In addition, we also included the SQLite embedded database (SQLite3, 2011) in the first set. The second set of programs are smaller benchmarks which are more compute intensive with less input and output. The second set of programs are derived from the programming language shootout benchmarks (Shootout, 2010), and are also used in the CompCert compiler test suite (Leroy et al., 2003–2008). The two sets of programs are shown in Table 7.1. The tables list the size of the programs, the total number of lines of code, and the total number of functions. The size and lines of code are included to give a sense of the relative sizes of the benchmarks. The number of functions is relevant because we will validate or fail to validate at the granularity of functions.

Each benchmark program is preprocessed before we begin the validation process. First, each program is compiled with Clang version 2.8, LLVM's front-end compiler for C (LLVM, 2010), and then processed with the "memory to register" (`mem2reg`) pass of the LLVM compiler to place ϕ-nodes, and ensure the assembly code is in proper SSA form. These assembly files make up the unoptimized inputs to our validation tool.

For our initial experiments, we configured LLVM M.D. using the SQLite benchmark program. The preprocessed SQLite input was given to LLVM configured to run one of a set of optimizations. Then, by examining the output of the validator, we

[1]The xalancbmk benchmark is missing because the LLVM bit-code linker fails on this large program.

Large Benchmarks			
name	size	LOC	functions
SQLite	5.6M	136K	1363
bzip2	904K	23K	104
gcc	63M	1.48M	5745
h264ref	7.3M	190K	610
hmmer	3.3M	90K	644
lbm	161K	5K	19
libquantum	337K	9K	115
mcf	149K	3K	24
milc	1.2M	32K	237
perlbench	15M	399K	1998
sjeng	1.5M	39K	166
sphinx	1.7M	44K	391

Small Benchmarks			
name	size	LOC	functions
aes	122K	1453	7
almabench	48K	351	6
binarytrees	11K	164	5
bisect	28K	376	7
chomp	42K	370	22
fannkuch	13K	154	2
fft	29K	191	2
fib	2.1K	19	2
integr	3.8K	32	4
knucleotide	47K	369	18
lists	10K	81	5
mandelbrot	8.1K	92	1
nbody	24K	174	5
nsieve	6.1K	57	3
nsievebits	6.9K	76	4
perlin	17K	75	6
qsort	11K	50	3
sha1	40K	234	10
spectral	11K	81	5
vmach	18K	216	2

Table 7.1: Test suite information

discovered useful rewrite rules and added them to the configuration. In the end, we added less than 20 rules. Using this configuration, we then applied our validator to the other benchmark programs.

7.1.1 Pipeline information

We have configured the LLVM compiler in two different modes for our tests. In the first mode, we run LLVM with a pipeline of optimizations and validate the results. In the second mode, we run LLVM with only one optimization at a time. For our experiments, we used the following optimizations:

- ADCE (advanced dead code elimination), followed by
- GVN (global value numbering),
- SCCP (sparse-condition constant propagation),
- LICM (loop invariant code motion),
- LD (loop deletion),
- LU (loop unswitching),
- DSE (dead store elimination).

These optimizations were chosen since they are the most advanced versions of the intra-procedural optimizations available in LLVM. For instance, we do not include constant propagation and constant folding because both of these are subsumed by sparse-conditional constant propagation (SCCP). Similarly, dead-code and dead-instruction elimination are subsumed by aggressive dead-code elimination (ADCE). Missing from

our list are the reassociate and instcombine optimizations. While these last two optimizations are conceptually simple, they require a more sophisticated theory of arithmetic than we currently have.

To test the pipeline of optimizations, we run LLVM with all of the optimizations enabled for each benchmark. Then, we run our validation tool and attempt to validate the optimizations applied to each function. If we are unable to validate an entire function, we count the entire function as having failed validation. This is an over-simplification, however it makes engineering the testing framework straight-forward.

7.2 Compiler User Experiments

The first set of experiments are geared toward answering the questions related to the practicality of our tool for a compiler user. For these experiments we will use the LLVM compiler configured with all of our optimizations enabled as a single pipeline. First we will look at the results of our validator using the configuration derived from SQLite. Then, we will investigate the compile-time and run-time impact of using the validator.

7.2.1 Pipeline Results

The results of our experiment for the optimization pipeline are shown in figure 7.1. We have displayed the two sets of benchmarks in two separate graphs for readability. Along the X-axis, we have the names of each of the benchmarks. The Y-axis shows the percentage of functions that our tool validated. For each benchmark there is a split bar. The lower bar indicates the percentage of functions validated, and the

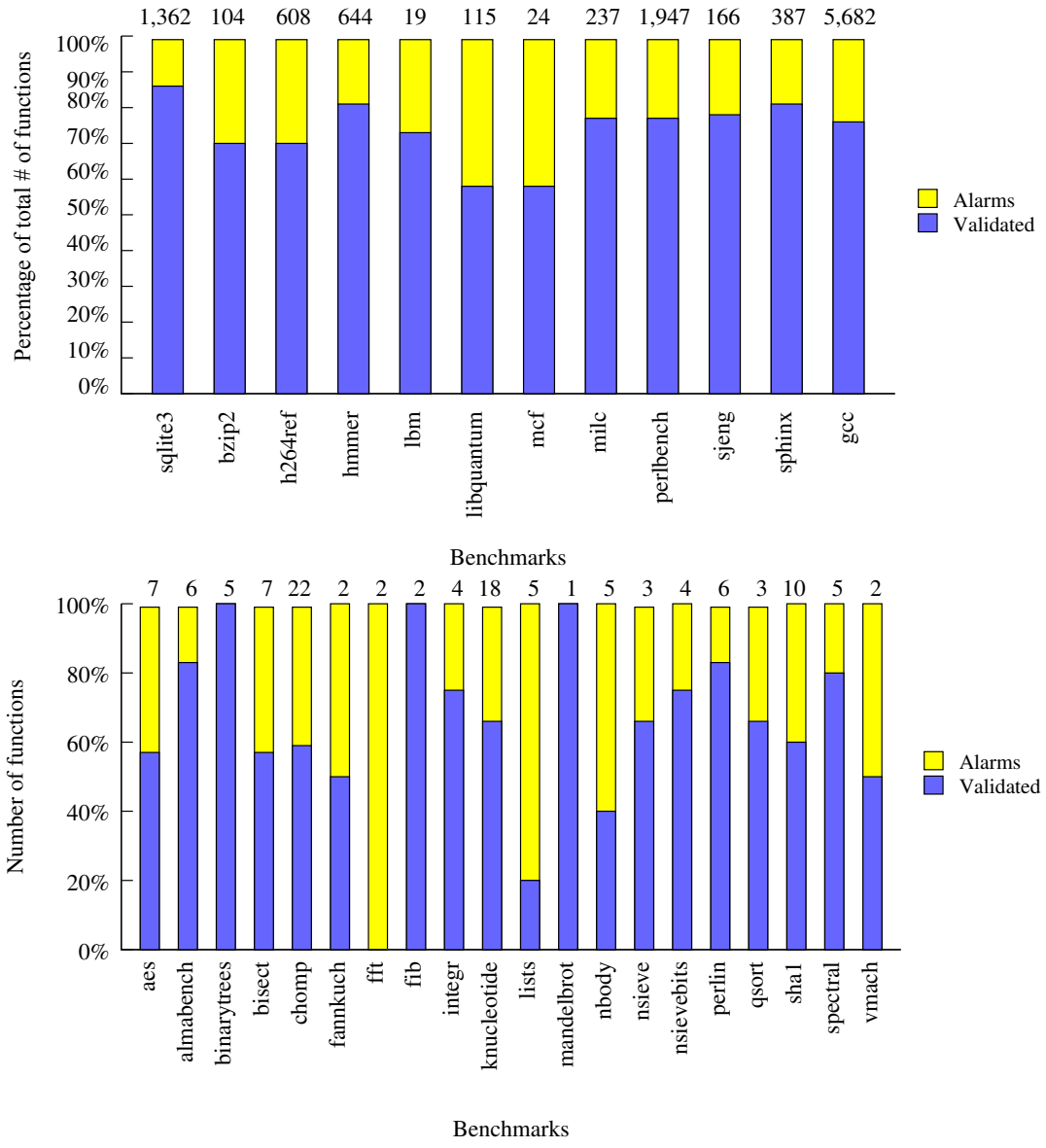


Figure 7.1: Validation results for optimization pipeline

upper bar indicates the unvalidated functions.

For our measurements, we counted the number of functions for which we could validate all of the optimizations performed on the function: even though we may validate many optimizations, if even one optimization fails to validate we count the entire function as failed. We found that this conservative approach, while rejecting more optimizations, leads to a simpler design for our validated optimizer which rejects or accepts whole functions at a time.

Overall, with a small number of rewrite rules, we can validate 80% of the per-function optimizations. We used the SQLite benchmark to engineer our rules, so it is not surprising that, that benchmark is very close to 90%. The rules chosen by studying SQLite are also very effective across the other benchmarks. We do not do quite as well for the perlbench and gcc benchmarks. One reason may be that we do not handle global constants or floating point expressions. We will investigate causes for false alarms in more detail in the next section. First, we will look at the run-time and compile-time impact of LLVM M.D. To investigate the run-time impact, we will use our validator to build a verified optimizer.

7.2.2 Validated Optimization

As mentioned in the introduction to this dissertation, a validated optimizer can be constructed by using our tool to check each optimization and only accept those that can be validated. Such a validator can be constructed by writing a simple wrapper around the optimizer. For instance, consider the pseudo-code in Figure 7.2. The function `vopt` implements a validated optimizer. For a given input file, `vopt`

```
function vopt(var input, var args) {
  output = opt args input
  for each function f in input {
    extract f from input as fi and output as fo
    if (!validate fi fo) {
      replace fo by fi in output
    }
  }
  return output
}
```

Figure 7.2: A validated optimizer using LLVM M.D. and an off-the-shelf optimizer.

first calls the off-the-shelf LLVM optimizer on the input program with the selected optimizations (optimizations flags are passed in the `args` parameter). Then, for each function in the input file we extract the original and optimized versions of the function. Then, we call the LLVM M.D. validator on these two versions of the function. If the validator cannot prove the functions are equivalent, then it will replace the optimized function in the output with the original, unoptimized function.

When used in this way, any false alarms produced by our tool will result in code that is less optimized. The natural question is: how much slower are the validated programs than the fully optimized versions? We attempt to answer this question using the compute-intensive shootout benchmark set. For our experiment, we produced three versions of each program: an unoptimized version, a version optimized with our pipeline of optimizations, and a validated version which uses the optimization pipeline but discards optimizations which we do not validate. The run-times for all versions of the benchmarks in shown in Table 7.2. We can see from this table that, in all cases, the verified version is approximately the same speed as the unverified version. However, the difference in run-time between the unoptimized and optimized

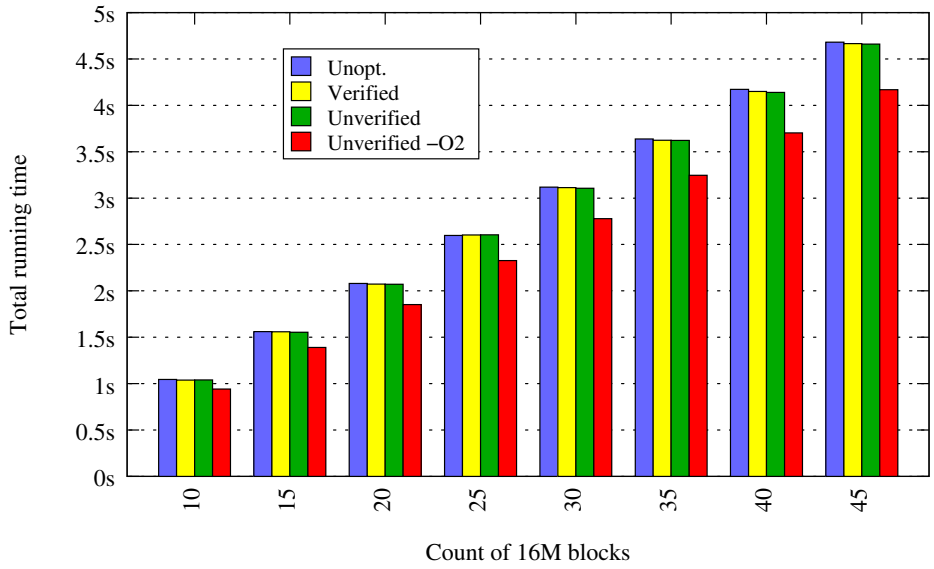
Benchmark	Unoptimized	Verified	Unverified	Percent Difference
aes.c	1.050s	1.036s	1.064s	2.6%
almabench.c	0.370s	0.369s	0.380s	2.9%
binarytrees.c	4.134s	4.128s	4.128s	$< 0.1\%$
bisect.c	4.522s	4.558s	4.523s	$< 0.1\%$
chomp.c	1.449s	1.585s	1.581s	$< 0.1\%$
fannkuch.c	0.221s	0.219s	0.220s	0.5%
fft.c	0.059s	0.061s	0.058s	$< 0.1\%$
fib.c	0.108s	0.108s	0.106s	$< 0.1\%$
integr.c	0.039s	0.011s	0.012s	8.3%
knucleotide.c	0.053s	0.052s	0.051s	$< 0.1\%$
lists.c	0.308s	0.306s	0.306s	$< 0.1\%$
mandelbrot.c	1.048s	0.965s	1.052s	8.3%
nbody.c	4.094s	4.110s	4.100s	$< 0.1\%$
nsieve.c	0.107s	0.107s	0.106s	$< 0.1\%$
nsievebits.c	0.061s	0.060s	0.060s	$< 0.1\%$
perlin.c	4.991s	5.001s	5.004s	$< 0.1\%$
qsort.c	0.246s	0.244s	0.245s	0.4%
sha1.c	0.300s	0.297s	0.299s	0.7%
spectral.c	1.743s	1.760s	1.742s	$< 0.1\%$
vmach.c	3.640s	3.637s	3.538s	$< 0.1\%$

Table 7.2: Timing Results

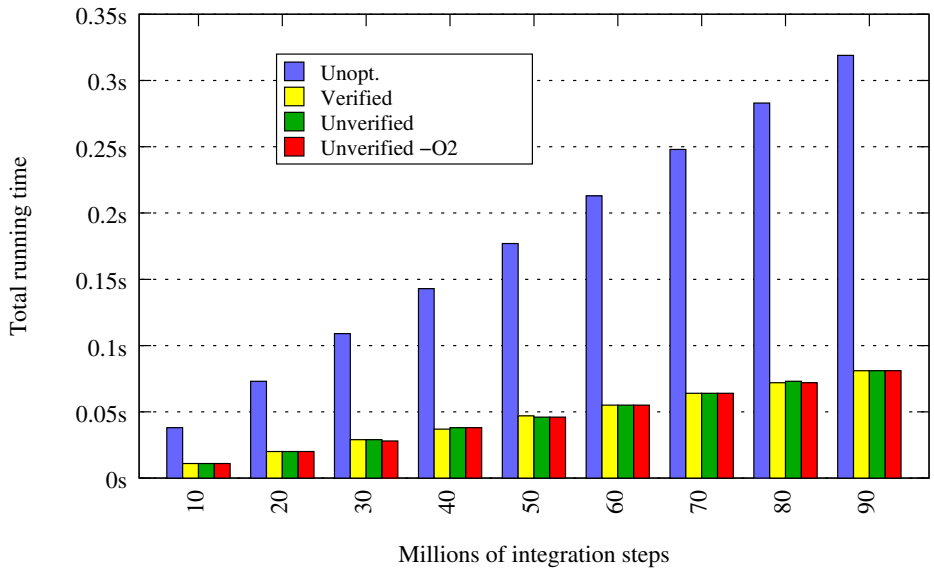
versions is often negligible. Therefore, we do not know if our validator is doing a good job, or if the optimizer is doing a poor job.

In order to get a better sense of how well our validator is doing, we will focus on two of the benchmarks: aes because it is very compute intensive, and the integr benchmark because the optimizer seems to be effective in this case. We modified the benchmarks to be able to tune the amount of computation being performed. In the case of aes, we parametrized the program by the amount of data to compress, and in the case of the integr benchmark we parametrized the program by the number of integration steps to consider. We compare the unoptimized, verified and unverified versions of these benchmarks with different amount of computation. In addition, we compare our versions to an unverified version optimized at "-O2". The "-O2" level of optimization includes all of our optimizations and several others including function inlining.

The results of our detailed experiment are shown in Figure 7.3. The first graph shows the result of running the aes benchmark on different amounts of data. The Y-axis shows the total running time for the program, and the X-axis shows the amount of data processed. As expected, the unoptimized version is always the slowest. The verified and unverified versions are roughly equivalent with the unverified version slightly faster on large amount of data. The "O2" version is faster in all cases. The second graph shows the result of running the integr benchmark with different numbers of integration steps. The Y-axis shows the total running time for the program, and the X-axis shows the number of integration steps performed. Again, the unoptimized version is slowest: in this case significantly slower. The verified and unverified versions



7.3(a): Detailed timing results for aes benchmark.



7.3(b): Detailed timing results for integr benchmark.

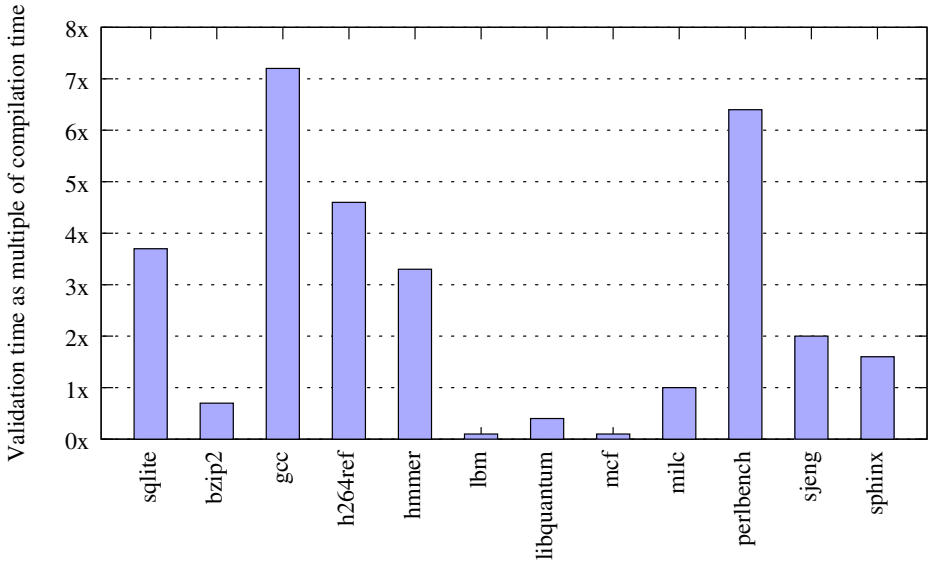
Figure 7.3: Comparison of validated and unvalidate AES benchmark.

are roughly equivalent, and also equivalent to the "O2" version.

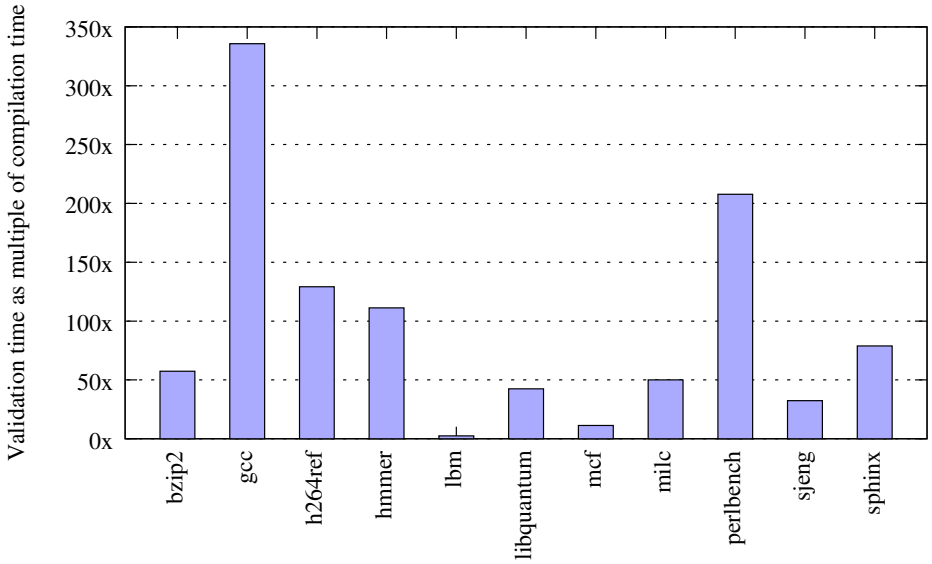
These are only two small benchmarks, however the results suggest that LLVM M.D. is validating the important optimizations. By important, we mean that the optimizations that LLVM M.D. does validate seem to yield most, if not all, of the run-time benefit. In the next section we will look at the compile-time cost of validation.

7.2.3 Validation Time

In order for our validator to be a practical tool, it is important that it does not take too long to validate programs. In order to measure this, we have timed the running of our validator on the large SPECCPU benchmark set. The benchmarks contain varying amounts of code, so we have normalized the validation time to the compile time. The results for LLVM M.D. are shown in Figure 7.4(a). Along the X-axis are the SPECCPU benchmarks. The Y-axis shows the total validation time as a multiple of the compilation time. For instance, the total compile time for gcc, the largest benchmark, is 1 minute and 10 seconds. The validation time for gcc is just over 7 times the compile time, or about 8 minutes. For bzip2, a smaller benchmark, the total compile time is 7 seconds. The validation time for bzip2 is a fraction of the compile time at around 0.9 seconds. The data shows an interesting pattern: for small benchmarks containing around 30,000 lines of code or less the validation time is less than the total compile time. For larger benchmarks validation time is slower than compilation by as much as an order of magnitude. This makes sense because for large pieces of code the validator may spend a lot of time rewriting large graphs only to fail to validate a function.



7.4(a): Timing Results for LLVM M.D. validator.



7.4(b): Timing Results for Peggy validator.

Figure 7.4: Validation time normalized to compilation time.

For comparison we have included the timing results for the Peggy validator (Stepp et al., 2011). Stepp et al. reproduced our experiments for the SPECCPU benchmarks using their validator based on equality saturation. In their report they list the number of validated functions and the average time for validating successfully and unsuccessfully. Therefore we have estimated their total run-time with the formula

$$[\text{avg. time success}] * [\text{validated}] + [\text{avg. time failure}] * [\text{failed}] \quad .$$

The estimated timing results for Peggy are shown in Figure 7.4(b); note SQLite was not included in their experiments. We see from the table that the same general pattern holds true for Peggy. For the smallest benchmark, lbm, Peggy is able to validate in roughly the same time it takes to compile. However, for larger benchmarks the validation time can be several hundred times slower.

7.3 Compiler Developer Experiments

The second set of experiments are geared toward answering the questions related to the practicality of our tool for a compiler developer. For these experiments we will use the LLVM compiler configured with one optimization at a time. First we will look at the results of our validator on each optimization using the configuration derived from SQLite. Then, we will look at the effect individual rewrite rules have on the number of validated functions for each optimization. Finally, we will try to get a sense of how difficult it is to configure our validator by improving a specific benchmark by adding new rewrite rules.

7.3.1 Testing Individual Optimizations

First we will look at the results of our validator on each optimization. In this setup, we run a single optimization on a benchmark and validate the results in the same manner as for the pipeline experiments. The charts in Figures 7.5 and 7.6 summarize the results of validating a few of the benchmark programs for each of the optimizations. The height of each bar indicates the total number of functions transformed for a given benchmark and optimization. The bar is split showing the number of validated (below) and unvalidated (above) functions. The total number of optimized functions is lower in these charts that in our previous charts because individual optimizations perform fewer transformations than the whole pipeline.

The results for individual optimizations show a clear pattern: GVN with alias analysis is the most challenging optimization for our tool. Most of our false alarms come from failing to validate the GVN translations. It is also the most important as it performs many more transformations than the other optimizations. Other optimizations like advanced dead-code elimination, and sparse conditional constant propagation can be completely validated in almost all cases.

Surprisingly, loop deletion (a form of dead code elimination), is not always validated. This reveals the complexities of a real-world optimizer like LLVM: in this case, other kinds of transformations, like re-association of expressions, are performed as side-effect of the analysis. One strength of our design is that we have the ability to validate some of the functions, and attempt to address others with additional rewrite rules. In the next section we will explore improvements to our tool by studying the effectiveness of rewrite rules on our system.

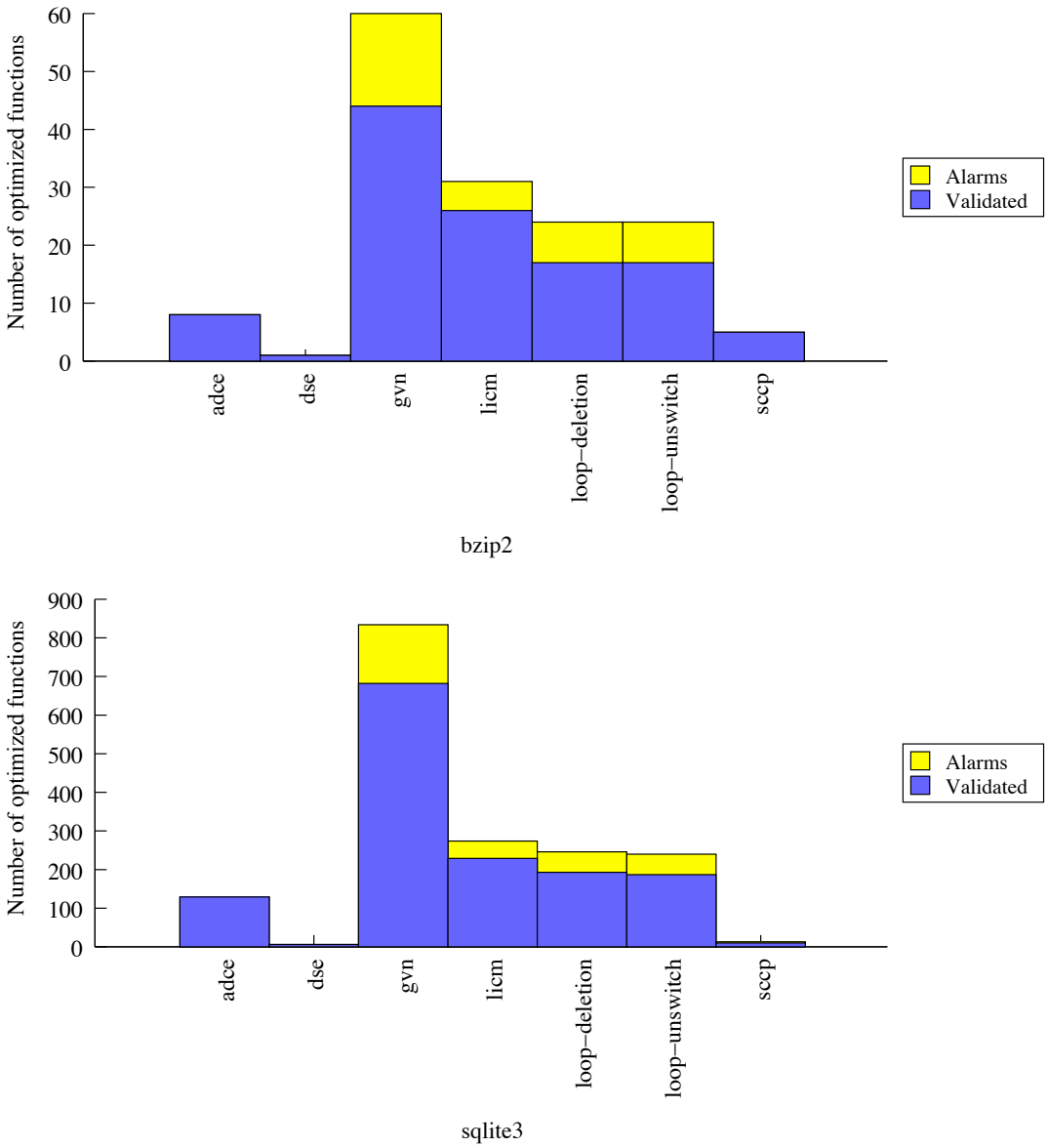


Figure 7.5: Validator results for individual optimizations

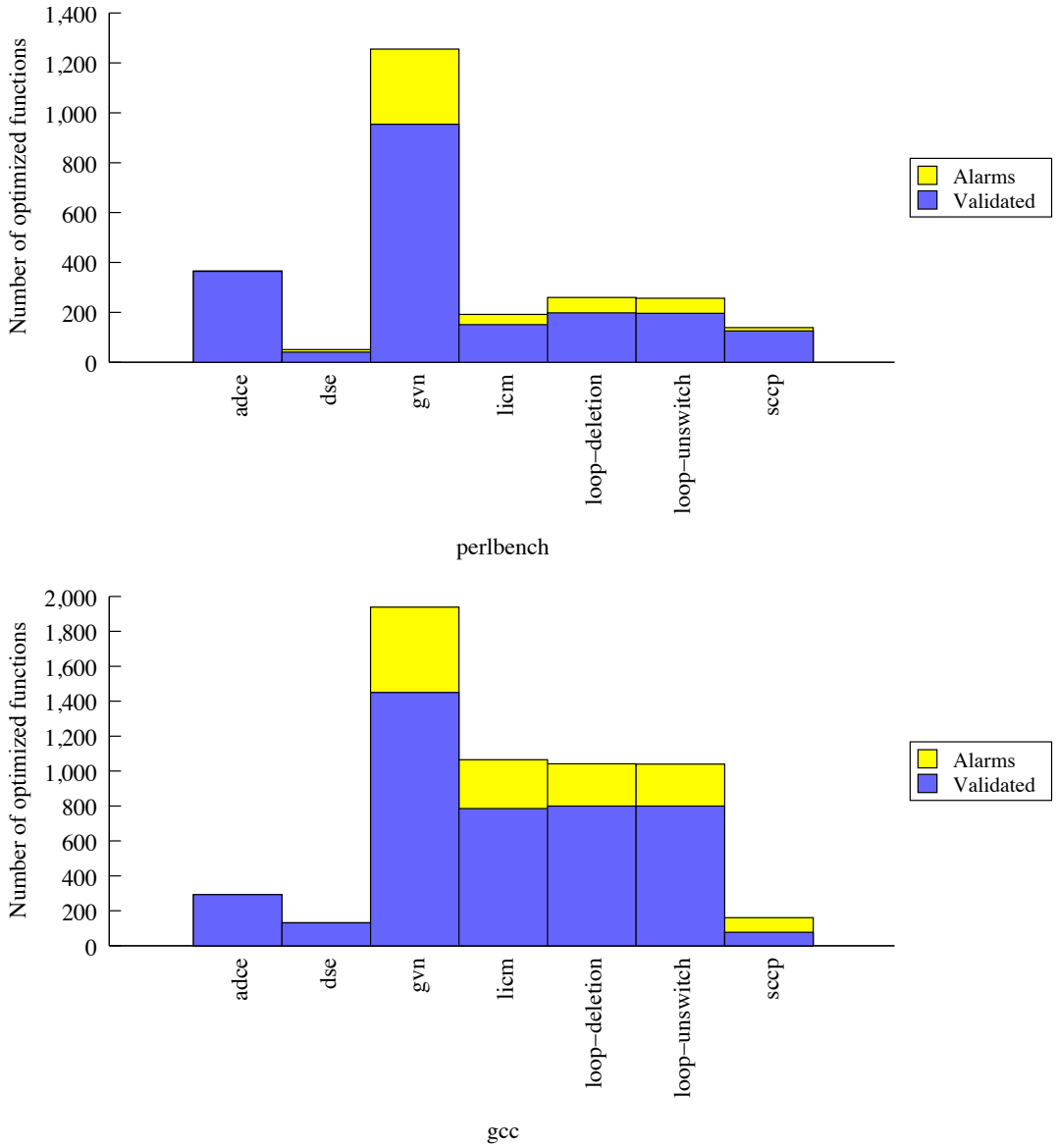
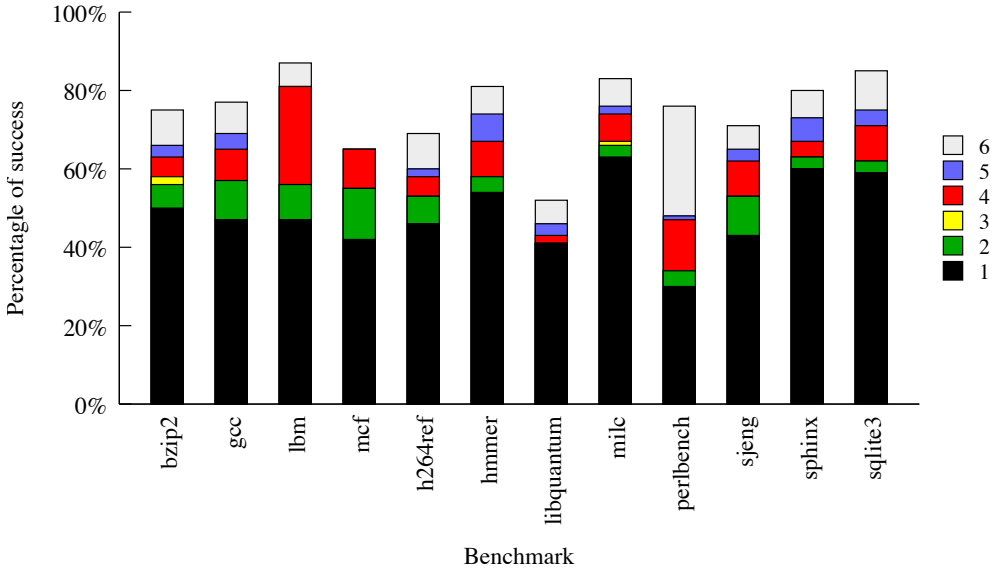


Figure 7.6: Validator results for individual optimizations (continued)



1. no rules 2. ϕ simplification 3. constant folding

4. load/store simplification 5. η simplification 6. commuting rules

Figure 7.7: Results for GVN optimization

7.3.2 Rewrite Rules

GVN. Figure 7.7 shows the effect of different rewrite rules for the GVN optimization with our benchmarks. The total height of each bar shows the percentage of functions validated for each benchmark using different sets of rewrite rules. The bars are divided to show how the results improve as we add rewrite rules to the system. We start with no rewrite rules, then we add rules and measure the improvement. The bars correspond to adding rules as follows:

1. no rules

2. ϕ simplification, e.g.

$$\phi\,\{\ldots, \overline{\texttt{true}_\texttt{i}} \to t, \ldots\} \downarrow t$$

$$\phi\,\{\overline{\overline{c_i} \to t}\} \downarrow t$$

3. constant folding, e.g.

$$\texttt{add}\ 3\ 2 \downarrow 5$$

$$\texttt{mul}\ 3\ 2 \downarrow 6$$

$$\texttt{sub}\ 3\ 2 \downarrow 1$$

4. load/store simplification, e.g.

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

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

5. η simplification, e.g.

$$\eta(c, \mu(x, x)) \downarrow x$$

$$\eta(c, y \mapsto \mu(x, y)) \downarrow x$$

6. commuting rules, e.g.

$$y + 7 + x \downarrow 7 + x + y$$

We have already described the first four rule sets (2-5 above) in Chapter 2. The last set of rules, "commuting rules" tries to rearrange arithmetic expressions to enable the former rules.

We can see from Figure 7.7 that different benchmarks are effected differently by the different sets of rewrite rules. For example, SQLite is not improved by adding rules for constant folding or ϕ-simplification. However, load/store simplification has an effect. This is probably because SQLite has been carefully tuned by hand and does not have many opportunities for constant folding or branch elimination. The lbm benchmark, on the other hand, benefits quite a lot from ϕ-simplification.

It is also interesting to note that from the data we have, it seems that our technique is able to successfully validate approximately 50% of GVN optimizations with no rewrite rules at all. This makes intuitive sense because our symbolic evaluation hides many of the syntactic details of the programs, and the transformations performed by many optimizations are, in the end, minor syntactic changes. By adding rewrite rules we can dramatically improve our results.

Up to this point, we have avoided adding "special purpose" rules. For instance, we could improve our results by adding rules that allow us to reason about specific C library functions. For example, the rule:

$$\begin{array}{ll} x = atoi(p); & y = atoi(q); \\ \Longrightarrow & \\ y = atoi(q); & x = atoi(p); \end{array}$$

can be added because `atoi` does not modify memory. Another example is:

$$\begin{array}{rcl} memset(p, x, l_1); & & \\ & \underset{l_2<l_1}{\Longrightarrow} & y = x \\ y = load(getelemptr(p, l_2)) & & \end{array}$$

which enables more aggressive constant propagation. Both of these rules seem to be used by the LLVM optimizer, but we have not added them to our validator at this time. However, adding these sorts of rules is fairly easy, and in a realistic setting

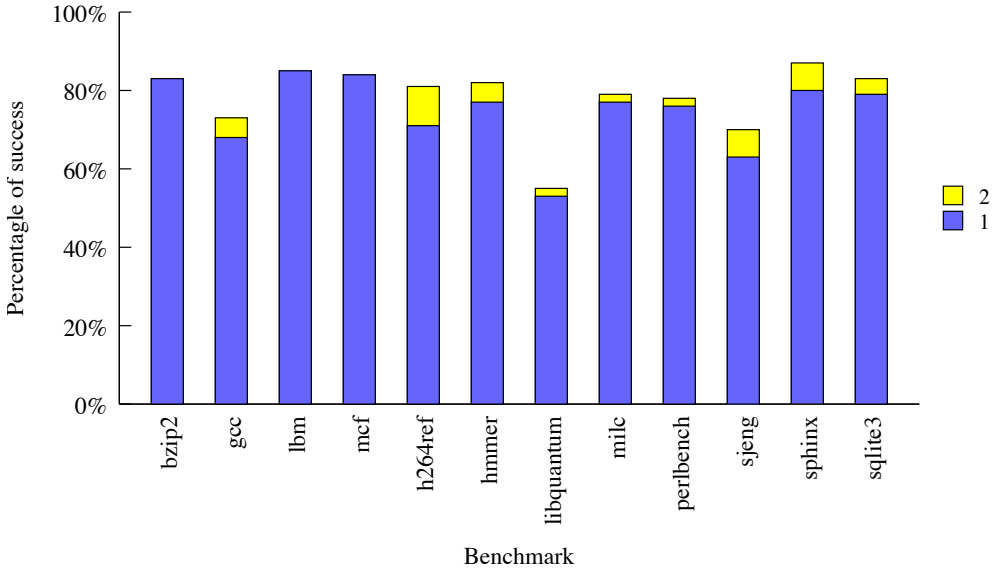


Figure 7.8: LICM

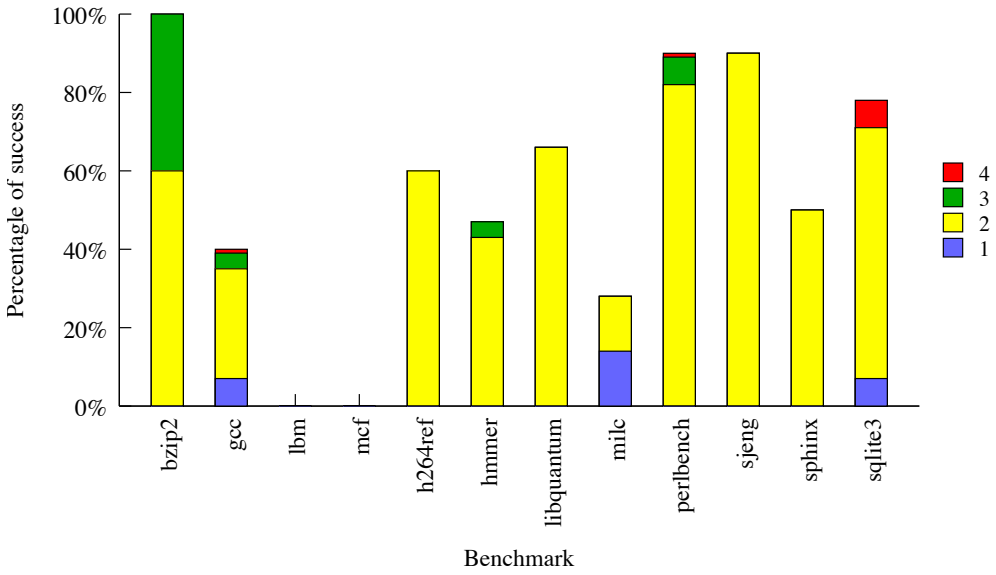
many such rules would likely be desired.

LICM. Figure 7.8 shows similar results for loop-invariant code motion (LICM). The baseline validator, with no rewrite rules, can validate approximately 75-80% of functions optimized by LICM. If we add in all of our rewrite rules, we only improve very slightly. In theory, we should be able to completely validate LICM with no rules. However, again, LLVM uses specific knowledge of certain C library functions. For example, in the following loop:

```
for (int i = 0; i < strlen(p); i++) x += p[i];
```

the call to `strlen` is known (by LLVM) to be constant. Therefore, LLVM will lift the call to `strlen` out of the loop:

```
int tmp = strlen(p);
for (int i = 0; i < tmp; i++) x += p[i];
```



1. no rules 2. constant folding 3. ϕ-simplification 4. all rules

Figure 7.9: SCCP

Our tool does not have any rules for specific library functions, and therefore we do not validate this transformation. The reason why we sometimes get a small improvement in LICM is because very occasionally a rewriting like the one above corresponds to one of our general rules.

SCCP. Figure 7.9 shows the effect of our rewrite rules on sparse-conditional constant propagation (SCCP). For this optimization, we used four configurations:

1. no rules
2. constant folding
3. ϕ-simplification
4. all rules

As expected, with no rules the results are very poor. However, if we add rules for constant folding, we see an immediate improvement. If we also add rules for reducing ϕ-nodes, bzip2 immediately goes to 100%, even though these rules have no effect on SQLite. However, additional rules do improve SQLite, but not the other benchmarks.

These observations indicate that the effect of new rewrite rules is difficult to predict. Our configuration was derived from looking at specific cases of failed validations and adding general rules to improve the results. In order for our tool to be useful, this process needs to be relatively easy to do for a new optimization. We will now investigate the difficulty of adding new rules through examples.

7.3.3 Improving Results with Additional Rewrite Rules

These results show that a small, general configuration derived from a single benchmark is effective across larger benchmarks. This is an important part of a practical tool. However, it is also important to be able to incrementally improve the validation result by adding new rewrite rules. To evaluate the ability to improve the validation result by adding rewrite rules, we will now take two benchmarks, mandlebrot and SHA1, which are not fully verified and add new rewrite rules to improve the validation results. Ultimately, we will add three new rules which eliminate all false alarms for the mandlebrot and SHA1 benchmarks. The rule for the mandlebrot benchmark was very easy to find. However, the rules needed for the SHA1 benchmark required a careful analysis of the mismatched value graphs. The final alarm for SHA1 required a deep knowledge of the validation process and several hours of effort studying the mismatched value graphs to solve. This highlights a significant weakness in the current

tools: interpreting false alarms is difficult.

Mandlebrot

The first benchmark we will study is the mandlebrot benchmark. This benchmark is small consisting of only one function (`main`) and 92 lines of C code. We can see from Figure 7.1 that this benchmark is completely unvalidated. This makes sense because there is only one function, and we failed to completely validate this function. Therefore, the whole function is counted as a false alarm.

The first step is diagnosing this benchmark is to get a breakdown of how the validator performs on each optimization. The LLVM M.D. tool can generate a report detailing this information. The output of this report for the mandlebrot benchmark is shown in Table 7.3. The table shows for each optimization the number of functions

	Unoptimized	Validated	Alarm	Total
gvn	0	1	0	1
sccp	0	0	1	1
licm	0	0	1	1
loop-deletion	0	1	0	1
adce	1	0	0	1
dse	1	0	0	1
loop-unswitch	0	1	0	1

Table 7.3: Per optimization results for mandlebrot

which are unoptimized, optimized and validated, and optimized and not validated (a.k.a. false alarms). Since we only have one function in this benchmark, the table shows us the the `adce` and `dse` optimizations had no effect; `gvn`, `loop-deletion`, and

`loop-unswitch` were successfully validated; and `sccp` and `licm` were not validated.

At this point, we can get more information by selecting a specific optimization to study. For instance, we can get a detailed report from the LLVM M.D. tool for the `sccp` optimization. Because we only have one function, we will also focus the report on the function `main`. After building and reducing the value graph, LLVM M.D. will attempt to show the differences between the mismatched value graphs. It does this by starting with the return nodes in the graph and highlighting those graph nodes which appear in the same positions in the two graphs, but which are structurally different. For this benchmark and function, the report shows a constant aligned with a floating point multiplication.

```
# validate mandlebrot sccp main
 [...]
173 /= 59
  BinOp fmul "" float64 30 30
  Const float "4.0"
30 -> Const float "2.0"
 [...]
```

For this example, it is very easy to spot the problem. The report says that the graph nodes 173 and 59 appear in structurally equivalent locations, but are built from different node types. In the original program we have node 173 which is a 64-bit floating point multiplication of two constants which are represented by graph node 30. Below we see that graph node 30 is the 64-bit floating point constant "2.0". In the optimized program we have graph node 59 which is the constant "4.0". The validator missed this optimization because it does not try to simplify floating point operations.

LLVM M.D. does not attempt to handle floating point operations because there are complex rules governing floating point arithmetic at different widths, and many

different representations for floating point constants. However, for this simple case, we have constants which can be unambiguously parsed, fit easily within the size constraints of the 64-bit registers, and will not raise any exceptional conditions. Under these constraints we can add the following rewrite rule:

$$k_1 *_{f64} k_2 \downarrow k_1 * k_2 \qquad k_1, k_2 \text{ simple constants}$$

where the side-condition (k_1, k_2 simple constants) enforces our constraints. With this one rule added, the mandlebrot benchmark is fully validated for all optimizations and the optimization pipeline.

SHA1

The second benchmark we will study is the SHA1 benchmark. This benchmark has 10 functions and 234 lines of C code. We can see from Figure 7.1 that this benchmark is 80% validated. As before, we first generate a per-optimization report using the LLVM M.D. tool. The report is show in Table 7.4. The table tells us that

	Unoptimized	Validated	Alarm	Total
gvn	2	6	2	10
sccp	10	0	0	10
licm	8	2	0	10
loop-deletion	8	2	0	10
adce	7	3	0	10
dse	9	1	0	10
loop-unswitch	8	2	0	10

Table 7.4: Per optimization results for SHA1

the global value numbering optimization is not validated for 2 of the 10 functions accounting for all of the alarms.

Unlike our previous benchmark, we now have multiple functions and we do not know which functions are validated and which are not. So, the first step is to ask LLVM M.D. to list the results of each function individually. The result is shown below.

```
# validate sha1 gvn
OK 32 0.676ms SHA1_init
OK 205 14.681ms SHA1_add_data
ALARM 426 27.449ms SHA1_transform
OK 79 1.236ms SHA1_finish
OK 97 1.775ms SHA1_copy_and_swap
ALARM 66 1.468ms main
OK 49 0.544ms do_test
OK 49 0.768ms do_bench
```

The output prints a single line for each function. The first column indicates if the function is completely validated or not. The second column is the size of the combined value graph (in this benchmark all of the graphs are small). The third column shows the total amount of time spent validating the function, and the last column is the function name. From this we see that the two functions `SHA1_transform` and `main` are not validated.

First, we will focus on the `main` function. If we look at the source code, we find the following code fragment:

```
/* Determine endianness */
union { int i; unsigned char b[4]; } u;
u.i = 0x12345678;
switch (u.b[0]) {
case 0x12: arch_big_endian = 1; break;
case 0x78: arch_big_endian = 0; break;
default: printf("Cannot determine endianness\n"); return 2;
```

```
}
```

In the optimized code, the expression `u.b[0]` has been replaced with `0x12`. Not surprisingly, when we generate a detailed report using LLVM M.D. it reports that the two subgraphs below are not equivalent.

$$\text{load}_{i8}\ p\ (\text{store}_{i32}\ p\ \text{0x12345678}\ m) \neq \text{0x12}$$

In this case, the LLVM optimizer has some information about the target machine that our tool does not have: namely that it is a little-endian machine. Although, LLVM is machine independent, each compilation unit is configured with some details about the intended target architecture, including the byte ordering. Using this information, LLVM knows that an 8-bit load of a 32-bit location will read the most-significant byte of the 32-bit word, and can simplify the load to a constant.

We can fix this alarm by adding a rewrite rule that is only valid for little-endian machines. The rule is:

$$\text{load}_{i8}\ p\ (\text{store}_{i32}\ p\ v\ m) \downarrow \text{trunc}_{i8}\ v \qquad .$$

This rule will rewrite a 8-bit load of a 32-bit store as a truncation of the original value. For constants, the truncation will simplify again to a smaller constant. With this additional rule, the two subgraphs above can be shown equivalent and the main function is fully validated.

Now, we turn our attention to the `SHA1_transform` function. Unfortunately, this alarm is not as easily solved as the previous two. This function takes in a structure which holds the current state of the hash algorithm. After processing an additional 64 bytes of data, the structure is updated with the new state. Hence, the value graphs for

this function involve lots of reads and writes to memory and complex values defined by a cascading series of loops. When we run the validator on this function it reports that an η-node is aligned with a store node and no further progress can be made.

After much study, the solution turned out to be an omission in the pointer equivalence checking. In LLVM, pointer arithmetic is done with a special instruction called `getelemptr`. Roughly speaking, the instruction `getelemptr p 0,0` is equivalent to the C-expression `&(p[0])+0`. Of course, this is also equivalent to just `p`, and this is the equivalence rule that was missing. After adding the following equivalence:

$$\text{getelemptr p 0, 0} \equiv p$$

the SHA1 benchmark is fully verified. Unfortunately, this rule could not be added as a normal rewrite law, but had to be manually added to the pointer aliasing relation.

For a user without deep knowledge of LLVM M.D., fixing this false alarm would not have been possible. Even if pointer equivalences could be added without modifying the code, diagnosing the problem was quite difficult. To diagnose the problem, we used LLVM M.D. to generate two sub-graphs which must be shown equivalent to validate the optimization. These two sub-graphs are shown in Figures 7.10 and 7.11. These two sub-graphs represent part of the final memory state of the optimized and unoptimized functions. The first sub-graph is rooted in an η-node indicating that the memory state is defined within a loop. The second sub-graph is rooted in a chain of stores leading to an η-node which is similar to, but not exactly the same as, the first sub-graph. One area where LLVM M.D. could be improved is to provide more powerful analysis tools to aide in interpreting these value graphs.

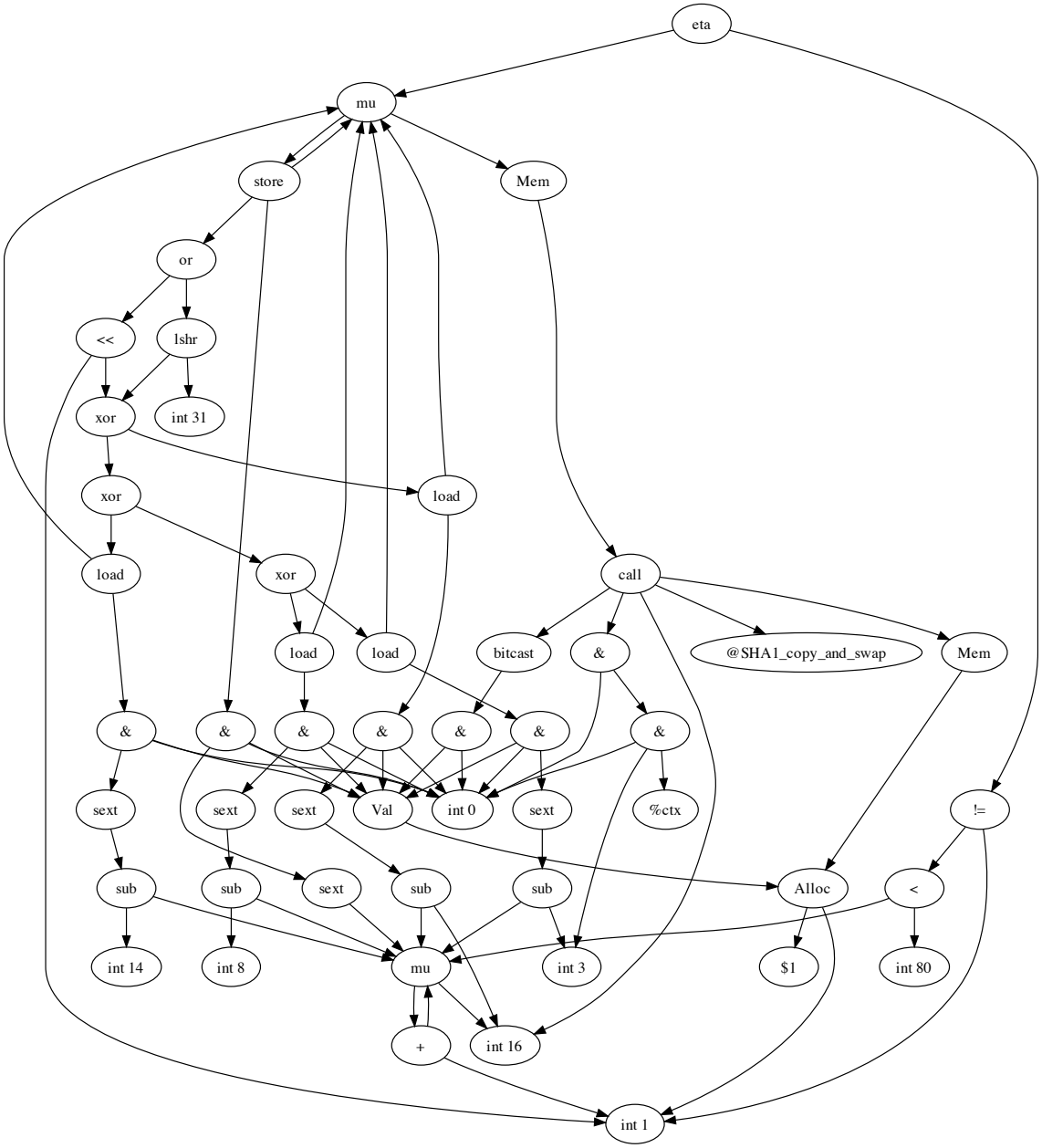


Figure 7.10: Subgraph for graph node 267

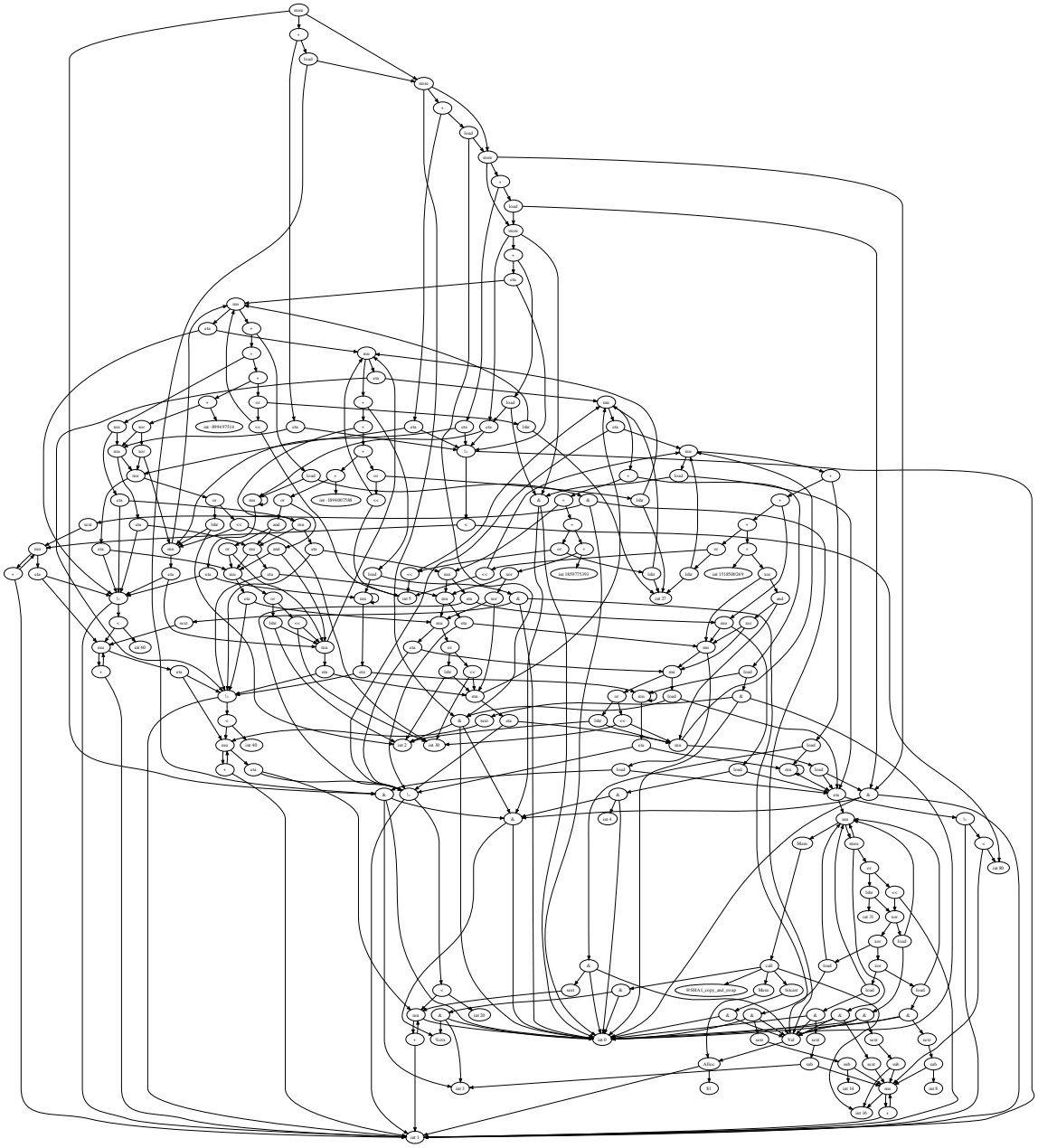


Figure 7.11: Subgraph for graph node 228

Chapter 8

Conclusion

In this dissertation, 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. Our experimental work has shown that our design will scale to real-world compilers. Because of this, we believe that normalizing value-graph translation validation of industrial-strength compilers without instrumentation is feasible. The design relies on well established algorithms and is simple enough to implement. We have been able to build a tool that can validate the optimizations of a decent LLVM pipeline on challenging benchmarks, with a reasonable rate of false alarms. Better yet, we know that many of the false alarms that we witness now require the addition of normalization rules but no significant changes in the design. For instance, insider knowledge of libc functions, floating-points constant folding and folding of global

variables are sources of false alarms that can be dealt with by adding normalization rules. There is also room for improvement of the runtime performance of the tool.

There are still a few difficult challenges ahead of us, the most important of which is inter-procedural optimizations. With LLVM, even -O1 makes use of such optimizations and, even though it is clear that simulation-based translation validation can handle inter-procedural optimizations (Pnueli and Zaks, 2008), we do not yet know how to precisely generalize normalizing translation. We remark that, in the case of safety-critical code that respects standard code practices (Association, 2004), as can be produced by tools like Simulink (Simulink, 2010), the absence of recursive functions allows us to inline every function (which is reasonable with hash-consing). Preliminary experiments indicate that we are able to validate very effectively inter-procedural optimizations in such a restricted case. Advanced loop transformations are also important, and we believe that this problem may not be as hard as it may seem at first. Previous work (Leroy, 2006; Tristan and Leroy, 2010) has shown that it can be surprisingly easy to validate advanced loop optimizations such as software pipelining with modulo variable expansion if we reason at the value-graph level.

8.0.4 Discussion

While implementing our prototype, we were surprised to find that essentially all of the technical difficulties lie in the complex ϕ-nodes. In an earlier version of this work we focused on structured code with binary ϕ-nodes. The binary ϕ-nodes did not present any real difficulties. However, once we removed the structured-code restriction we encountered more problems. First, although the algorithms are known, computing

the gates for arbitrary control flow is a fairly involved task. Also, since the gates are dependent on the paths in the CFG, and general C code does not have a simple structured control flow, optimizations will often change the gating conditions even if the control flow is not changed.

Another important aspect of the implementation is the technique for maximizing sharing within the graph. The rewrite rules do a good job of exposing equivalent leaf nodes in the graphs. However, in order to achieve good results, it is important to find equivalent cycles in the graphs and merge them. Again, matching complex ϕ-nodes seems to be the difficult part. To match cycles, we find pairs of μ-nodes in the graph, and trace along their paths in parallel trying to build up a unifying substitution for the graph nodes involved. For ϕ-nodes we sort the branches and conditions and perform a syntactic equality check. This technique is very simple, and efficient because it only needs to query and update a small portion of the graph.

We also experimented with a Hopcroft partitioning algorithm(Hopcroft, 1971). Rather than a simple syntactic matching, our partitioning algorithm uses a prolog-style backtracking unification algorithm to find congruences between ϕ-nodes. Surprisingly, the partitioning algorithm with backtracking does not perform better than the simple unification algorithm: both algorithms give us roughly the same percentage of validation. Our implementation uses the simple algorithm by default, and when this fails it falls back to the slower, partitioning algorithm. Interestingly, this strategy performs only slightly better than either technique alone. All of these techniques can be seen as reintroducing the bisimiulation relation discussed in Chapter 3 Section 3.4.

Matching expressions with complex ϕ-nodes seems well within the reach of any

SMT prover. Our preliminary experiments with Z3 suggest that it can easily handle the sort of equivalences we need to show. However, this seems like a very heavy-weight tool. One question in our minds is whether or not there is an effective technique somewhere in the middle: more sophisticated than syntactic matching, but short of a full SMT prover.

8.1 Future work

We believe our system could be improved by better equality checking algorithms. 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 (Chandra et al., 2009), 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 (Gulwani

and Necula, 2004). 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.

Bibliography

Z. M. Ariola and S. Blom. Lambda calculi plus letrec. Technical report, Vrije Universiteit, Amsterdam, 1997.

Zena M. Ariola and Jan Willem Klop. Cyclic lambda graph rewriting. In *In Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 416–425. IEEE Computer Society Press, 1994.

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

H. P. Barendregt. *The Lambda Calculus: Its Synatx and Semantics*. North-Holland, Amsterdam, 1984.

Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

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*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.

Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. URL `citeseer.ist.psu.edu/berry92esterel.html`.

R. F. Blute, J.R.B. Cockett, and R.A.G. Seely. Categories for computation in context and unified logic: The "intuitionist" case, 1997.

F. Borceux. Non-pointed strongly protomodular theories. *Applied Categorical Structures*, 12(4):319–338, 2004.

James R. Bunch and Donald J. Rose. Partitioning, tearing and modification of sparse linear systems. *Journal of Mathematical Analysis and Applications*, 48(2):574 – 593, 1974. ISSN 0022-247X. doi: 10.1016/0022-247X(74)90179-6.

Paul Caspi and Marc Pouzet. Synchronous kahn networks. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 226–238, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-770-7. doi: http://doi.acm.org/10.1145/232627.232651.

Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Submitted to publication, 2002.

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.

Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, 1999. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796899003342.

Coq development team. The Coq proof assistant. Software and documentation available at http://coq.inria.fr/, 1989–2008.

Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, September 2001.

Haskell Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20:584–590, 1934.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/115372.115320. URL `http://doi.acm.org/10.1145/115372.115320`.

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

Levent Erkök. *Value recursion in monadic computations*. PhD thesis, OGI School of Science and Engineering, October 2002.

Levent Erkök and John Launchbury. A recursive do for haskell. In *Haskell Workshop 2002*, October 2002.

Loukas Georgiadis and Robert E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 433–442, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. ISBN 0-89871-585-7. URL `http://portal.acm.org/citation.cfm?id=1070432.1070492`.

Paola Giannini, Furio Honsell, and Simona Ronchi Della Rocca. Type inference: some results, some problems. *Fundam. Inf.*, 19(1-2):87–125, 1993. ISSN 0169-2968.

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: http://doi.acm.org/10.1145/165180.165214. URL `http://doi.acm.org/10.1145/165180.165214`.

Jean-Yves Girard. Geometry of interaction 1: Interpretation of system f. In S. Valentini R. Ferro, C. Bonotto and A. Zanardo, editors, *Logic Colloquium '88, Proceedings of the Colloquium held in Padova*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221 – 260. Elsevier, 1989. doi: DOI: 10.1016/S0049-237X(08)70271-4. URL `http://www.sciencedirect.com/science/article/pii/S0049237X08702714`.

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

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. URL `citeseer.ist.psu.edu/halbwachs91synchronous.html`.

Paul Havlak. Construction of thinned gated single-assignment form. In *In Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 477–499. Springer Verlag, 1993.

Paul Havlak. Construction of thinned gated single-assignment form. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 477–499, London, UK, 1994. Springer-Verlag. ISBN 3-540-57659-2. URL `http://portal.acm.org/citation.cfm?id=645671.665393`.

John Hopcroft. An n log n algorithm for minimizing states of a finite automaton. In *The Theory of Machines and Computations*, 1971.

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.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School*, volume 2638 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5. URL `http://dl.acm.org/citation.cfm?id=647698.734150`.

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

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *2000 Marktoberdorf Summer School*, July 2002.

Achim Jung, Der Technischen Hochschule Darmstadt, Dipl. math Achim Jung, Referent Prof, Dr. K. Keimel, Koreferent Prof, and Dr. K. h. Hofmann. Cartesian closed categories of domains, 1988.

G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

Aditya Kanade, Amitabha Sanyal, and Uday Khedker. A PVS based framework for validating compiler optimizations. In *4th Software Engineering and Formal Methods*, pages 108–117. IEEE Computer Society, 2006.

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.

John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: http://doi.acm.org/10.1145/178243.178246. URL `http://doi.acm.org/10.1145/178243.178246`.

Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

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

Xavier Leroy et al. The CompCert verified compiler. http://compcert.inria.fr, 2003–2008.

version 2.8 LLVM. http://llvm.org, 2010.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-54396-1. URL `http://portal.acm.org/citation.cfm?id=127960.128035`.

E. Moggi. Computational lambda-calculus and monads. In *4th Logic in computer science*, pages 14–23. IEEE, 1989.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *In ICFP*, pages 62–73. ACM Press, 2006.

Aleksandar Nanevski, Paul Govereau, and Greg Morrisett. Towards type-theoretic semantics for transactional concurrency. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-420-1. doi: http://doi.acm.org/10.1145/1481861.1481872. URL `http://doi.acm.org/10.1145/1481861.1481872`.

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

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.

Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallinn, Estonia, September 2005. ACM Press. to appear.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 51–64, Pittsburgh, PA, October 2002. ACM Press.

Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990a. ACM. ISBN 0-89791-364-7. doi: http://doi.acm.org/10.1145/93542.93578. URL `http://doi.acm.org/10.1145/93542.93578`.

Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not.*, 25:257–271, June 1990b. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/93548.93578. URL `http://doi.acm.org/10.1145/93548.93578`.

Ross Paterson. Arrows and computation. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.

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.

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.

John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, October 1997.

John Power and Hayo Thielecke. Closed Freyd- and kappa-categories. In *ICALP*, volume 1644 of *LNCS*. Springer, 1999.

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

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

D.J. Rose, G.G. Whitten, A.H. Sherman, and R.E. Tarjan. Algorithms and software for in-core factorization of sparse symmetric positive definite matrices. *Computers and Structures*, 11(6):597 – 608, 1980. ISSN 0045-7949. doi: 10.1016/0045-7949(80)90066-8.

Benchmarks for Programming Language Comparisons Shootout. http://shootout.alioth.debian.org/, 2010.

Simulink. http://mathworks.com, 2010.

2006 SpecCPU. http://www.spec.org/cpu2006/, 2006.

SQLite3. http://www.sqlite.org, 2011.

Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 737–742, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. URL `http://dl.acm.org/citation.cfm?id=2032305.2032364`.

S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *ADVANCED FUNCTIONAL PROGRAMMING*, pages 184–207. Springer-Verlag, 1996.

V. Vene T. Uustalu. The essence of dataflow programming. *In Revised Lectures from Central European Functional Programming School*, July 2005. to appear.

Robert E Tarjan. Applications of path compression on balanced trees. Technical report, Stanford University, Stanford, CA, USA, 1975.

Robert Endre Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26:690–715, October 1979. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322154.322161. URL `http://doi.acm.org/10.1145/322154.322161`.

Robert Endre Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28:594–614, July 1981. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322261.322273. URL `http://doi.acm.org/10.1145/322261.322273`.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *36th Principles of Programming Languages*, pages 264–276. ACM, 2009.

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.

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.

Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *37th Principles of Programming Languages*, pages 83–92. ACM Press, 2010.

Peng Tu and David Padua. Efficient building and placing of gating functions. In *Programming Language Design and Implementation*, pages 47–55, 1995a.

Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 47–55, New York, NY, USA, 1995b. ACM. ISBN 0-89791-697-2. doi: http://doi.acm.org/10.1145/207110.207115. URL `http://doi.acm.org/10.1145/207110.207115`.

Peng Tu and David Padua. Efficient building and placing of gating functions. *SIGPLAN Not.*, 30:47–55, June 1995c. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/223428.207115. URL `http://doi.acm.org/10.1145/223428.207115`.

Tarmo Uustalu and Varmo Vene. Signals and comonads. *Journ. of Universal Comp. Sci*, 11:1310–1326, 2005.

William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-729650-6.

Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 45–52, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-142-3.

Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Theoretical Computer Science, (Special issue of selected papers from 2'nd European Symposium on Programming)*, pages 231–248, 1990.

Daniel Weise, Roger F. Crew, Michael D. Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *21st Principles of Programming Languages*, pages 297–310, 1994.

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.

Appendix A

Data-flow Semantics

A.1 Operational Semantics

In this section we will present a precise operational semantics for our term language. The semantics are similar to a synchronous data-flow semantics. We will make use of the step relation defined in Chapter 3 for our expression language.

Our judgement is of the form: $t \xrightarrow{v} t'$. This can be read as t steps to t' and produces the value v on its output stream. The operational semantics is shown in Figure A.1.

A.2 Static Semantics

The static semantics is a combination of a traditional clock calculus without recursion, and the simply typed expression language from Chapter 3. The syntax of types is shown below.

$$
\begin{array}{lcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} & \text{Base Types} \\
 & \mid & \tau \to \tau & \text{Function Types} \\
 & \mid & [\tau] & \text{Stream Type} \\
\textbf{clk} & ::= & c & \text{Basic Clock} \\
 & \mid & c \,\texttt{on}\, e & \text{Slowed Clock}
\end{array}
$$

The typing rules are shown in Figure A.2. The following soundness theorem holds.

Theorem 8 (Soundness)**.** *For all t, τ and c, if $\Gamma \vdash t : \tau, c$, then there exists v and t' such that*

$$t \xrightarrow{v} t'$$

$$\Gamma \vdash t' : \tau, c \qquad .$$

$$\frac{}{\texttt{const}\ v \xrightarrow{\epsilon} \texttt{const}\ v}\ (pure_0) \qquad \frac{}{\texttt{const}\ v \xrightarrow{v} \texttt{const}\ v}\ (pure_n) \qquad \frac{x \xrightarrow{\epsilon} x'}{\texttt{map}\ f\ x \xrightarrow{\epsilon} \texttt{map}\ f\ x'}\ (map_0)$$

$$\frac{x \xrightarrow{v} x' \qquad f\ v \rightarrow^* w}{\texttt{map}\ f\ x \xrightarrow{w} \texttt{map}\ f\ x'}\ (map_n) \qquad \frac{x \xrightarrow{\epsilon} x' \qquad y \xrightarrow{\epsilon} y'}{\texttt{zip}(x,y) \xrightarrow{\epsilon} \texttt{zip}(x',y')}\ (zip_0) \qquad \frac{x \xrightarrow{v} x' \qquad y \xrightarrow{w} y'}{\texttt{zip}(x,y) \xrightarrow{(v,w)} \texttt{zip}(x',y')}\ (zip_n)$$

$$\frac{c \xrightarrow{\epsilon} c' \qquad x \xrightarrow{\epsilon} x' \qquad y \xrightarrow{\epsilon} y'}{\phi(c,x,y) \xrightarrow{\epsilon} \phi(c',x',y')}\ (\phi_0) \qquad \frac{c \xrightarrow{\texttt{true}} c' \qquad x \xrightarrow{v} x' \qquad y \xrightarrow{w} y'}{\phi(c,x,y) \xrightarrow{v} \phi(c',x',y')}\ (\phi_t)$$

$$\frac{c \xrightarrow{\texttt{false}} c' \qquad x \xrightarrow{v} x' \qquad y \xrightarrow{w} y'}{\phi(c,x,y) \xrightarrow{w} \phi(c',x',y')}\ (\phi_f) \qquad \frac{x \xrightarrow{\epsilon} x' \qquad y \xrightarrow{v} y'}{\mu(x,y) \xrightarrow{\epsilon} \epsilon : \mu(x',y')}\ (\mu_0) \qquad \frac{x \xrightarrow{v} x' \qquad y \xrightarrow{w} y'}{\mu(x,y) \xrightarrow{\epsilon} \texttt{pre}\ v\ y'}\ (\mu_n)$$

$$\frac{x \xrightarrow{\epsilon} x' \qquad c \xrightarrow{\epsilon} c'}{\eta(x,c) \xrightarrow{\epsilon} \eta(x',c')}\ (\eta_0) \qquad \frac{x \xrightarrow{\texttt{false}} x' \qquad c \xrightarrow{w} c'}{\eta(x,c) \xrightarrow{w} \texttt{const}\ w}\ (\eta_f) \qquad \frac{x \xrightarrow{\texttt{true}} x' \qquad c \xrightarrow{w} c'}{\eta(x,c) \xrightarrow{\epsilon} \eta(x',c')}\ (\eta_t)$$

$$\frac{x \xrightarrow{\epsilon} x' \qquad c \xrightarrow{\epsilon} c'}{\sigma(v,x,c) \xrightarrow{\epsilon} \sigma(v,x',c')}\ (\sigma_0) \qquad \frac{x \xrightarrow{\texttt{true}} x' \qquad c \xrightarrow{w} c'}{\sigma(v,x,c) \xrightarrow{w} \sigma(w,x',c')}\ (\sigma_t) \qquad \frac{x \xrightarrow{\texttt{false}} x' \qquad c \xrightarrow{w} c'}{\sigma(v,x,c) \xrightarrow{v} \sigma(v,x',c')}\ (\sigma_f)$$

Figure A.1: Operational Semantics for terms.

$$\text{(PURE)}\quad \frac{\Gamma \vdash_\lambda v : \tau}{\Gamma \vdash \texttt{const}\ v : [\tau], c} \qquad \text{(ZIP)}\quad \frac{\Gamma \vdash x : \tau_1, c \qquad \Gamma \vdash y : \tau_2, c}{\Gamma \vdash \texttt{zip}(x, y) : [\tau_1 \star \tau_2], c}$$

$$\text{(MAP)}\quad \frac{\Gamma \vdash_\lambda f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : [\tau_1], c}{\Gamma \vdash \texttt{map}\ f\ x : [\tau_2], c}$$

$$(\phi)\quad \frac{\Gamma \vdash x : [\texttt{bool}], c \qquad \Gamma \vdash y : [\tau], c \qquad \Gamma \vdash z : [\tau], c}{\Gamma \vdash \phi(x, y, z) : [\tau], c} \qquad (\mu_1)\quad \frac{\Gamma \vdash x : [\tau], c \qquad \Gamma \vdash y : [\tau], c}{\Gamma \vdash \mu(x, y) : [\tau], c}$$

$$(\mu_2)\quad \frac{\Gamma \vdash x : [\tau], c \qquad \Gamma \vdash y : [\tau], c\ \texttt{on}\ e}{\Gamma \vdash \mu(x, y) : [\tau], c\ \texttt{on}\ e} \qquad (\eta)\quad \frac{\Gamma \vdash x : [\tau], c \qquad \Gamma \vdash y : [\texttt{bool}], c}{\Gamma \vdash \eta(x, y) : [\tau], c\ \texttt{on}\ y}$$

$$(\sigma)\quad \frac{\Gamma \vdash_\lambda v : \tau \qquad \Gamma \vdash x : [\tau], c \qquad \Gamma \vdash y : [\texttt{bool}], c}{\Gamma \vdash \sigma(v, x, y) : [\tau], c}$$

Figure A.2: Typing rules for terms.