UNIVERSITY OF CALIFORNIA, SAN DIEGO

Equality Saturation: Engineering Challenges and Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Michael Benjamin Stepp

Committee in charge:

Professor Sorin Lerner, Chair
Professor Ranjit Jhala
Professor William Griswold
Professor Rajesh Gupta
Professor Todd Millstein

2011

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

The dissertation of Michael Benjamin Stepp is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2011

DEDICATION

First, I would like to express my gratitude to my advisor Sorin Lerner. His guidance and encouragement made this research possible.

I offer thanks to my parents for believing in me, and in my abilities. Their love and support made this academic achievement possible. I am lucky to have been born to the greatest set of parents anyone could ask for, and I see proof of that every day.

Finally, I need to thank Amie McElwain, my partner. She has been endlessly supportive throughout this process and was willing to make compromises and accommodations so that I could complete my education. She uprooted herself and relocated to San Diego so that we could be together. I am extremely grateful for that sacrifice.

EPIGRAPH

Computers are useless. They can only give you answers.

—Pablo Picasso

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my coauthors of the Peggy system and the publications based on it, Ross Tate, Zachary Tatlock, and Sorin Lerner. They all worked very hard to make the idea of Equality Saturation a reality, and none of this work would be possible without them.

I would also like to thank my committee members. Their guidance and feedback during my proposal helped steer me in the right direction to reach this point. Also, their patience and flexibility throughout this entire process is greatly appreciated.

Chapters 2, 7, 9, 10, 11, and 12 contain material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Logical Methods in Computer Science 2010*. The dissertation author was the secondary investigator and author of this paper.

Chapters 2, 7, 9, 10, 11, and 12 contain material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. The dissertation author was the secondary investigator and author of this paper.

Chapter 10 contains material taken from "Equality-Based Translation Validator for LLVM", by Michael Stepp, Ross Tate, and Sorin Lerner, which appears in *Proceedings of the 23rd International Conference on Computer Aided Verifica-*

tion (CAV 2011). The dissertation author was the primary investigator and author of this paper.

VITA AND PUBLICATIONS

2003	B. S. in Computer Science University of Arizona
2005	M. S. in Computer Science University of Arizona
2003-2005	Research Assistant University of Arizona
2006	Internship Intuit Inc. San Diego, California
2007	Internship Google Kirkland, Washington
2006-2011	Research Assistant University of California, San Diego
2011	Ph. D. in Computer Science University of California, San Diego

PUBLICATIONS

Michael Stepp, Ross Tate, and Sorin Lerner. "Equality-Based Translation Validator for LLVM". In the *23rd International Conference on Computer Aided Verification (CAV 2011)*

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality Saturation: A New Approach to Optimization", In *Logical Methods in Computer Science*, vol 7, issue 1, 2011

Ross Tate, Michael Stepp, and Sorin Lerner. "Generating compiler optimizations from proofs". In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '10)*, 2010.

Michael Stepp and Beth Simon. "Introductory computing students' conceptions of illegal student-student collaboration". In *Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10)*. ACM, New York, NY, USA, 295-299

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. "Equality Saturation: a new approach to optimization". In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. ACM, 264-276

Christian Collberg, Ginger Myles, and Michael Stepp. "An empirical study of Java bytecode programs". In *Software: Practice and Experience*, volume 37, issue 6, (May 2007), 581-641.

Christian Collberg, Stephen Kobourov, C. Hutcheson, J. Trimble, M. Stepp. "Monitoring Java Programs Using Music". Technical Report, University of Arizona, 2005.

S. Kobourov, K. Pavlou, J. Cappos, M. Stepp, M. Miles, A. Wixted: "Collaboration with DiamondTouch". In *INTERACT(2005)* 986-989

C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. 2004. "Dynamic path-based software watermarking". In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI '04)*.

C. Collberg, G. Myles, and M. Stepp. "Cheating Cheating Detectors". Technical Report, University of Arizona, 2004.

ABSTRACT OF THE DISSERTATION

Equality Saturation: Engineering Challenges and Applications

by

Michael Benjamin Stepp

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Sorin Lerner, Chair

In this dissertation, I describe the Peggy system for performing program optimization and translation validation. Peggy is based on the concept of Equality Saturation, in which axiomatic reasoning is applied to a program to produce exponentially many equivalent versions of that program, which can then be explored simultaneously. This is achieved by using a custom intermediate representation that facilitates mathematical reasoning over programs. I will specifically address some of the engineering challenges posed by making a working implementation of the Equality Saturation technique, as well as the major applications to which we have applied it.

I implemented front-ends for Peggy to convert both Java bytecode programs and LLVM programs to and from our custom intermediate representation. I designed and implemented a domain-specific language for writing the axioms that are used by the Equality Saturation engine. For the purposes of optimization, I designed the technique whereby we choose the optimal program from our representation of exponentially many equivalent programs. I implemented both our optimization and our translation validation frameworks that use the Equality

Saturation engine. In addition, I performed experiments showing the effectiveness of Equality Saturation at both program optimization and translation validation.

Chapter 1

Introduction

Equality Saturation is a technique developed by Ross Tate, Zachary Tatlock, Sorin Lerner, and myself and was first presented in [TSTL09]. It is a method of exploring a large portion of the space of programs that are equivalent to a particular input program. This is achieved by first converting the program to a purely functional form, and then applying *equality analyses* to it to deduce equivalent forms for the subexpressions within the program. Every new equivalent subexpression that is found greatly increases the number of overall programs that have been discovered. Once we have finished applying equality analyses, we are left with a concise representation of exponentially many programs that are equivalent to the original.

There are two applications for which this technique is especially useful. Firstly, we can explore the set of equivalent programs to find the one that is optimal according to some metric. If we then output this program instead of the original, we have essentially optimized the original program according to that metric. In this way Equality Saturation can form the basis of a general-purpose program optimizer. Secondly, we can easily check for the equivalence of two programs by testing to see if they both exist in the program set that came out of an Equality Saturation instance. In particular, if we start with two input programs instead of one, and apply Equality Saturation on them both simultaneously, we can say they are equivalent if all their nodes are eventually marked equivalent. This allows Equality Saturation to be used for translation validation.

In this dissertation, I will discuss my contributions to designing and building a system based on the Equality Saturation technique, which we call Peggy. A brief summary of my contributions is as follows. I designed a framework around the main Equality Saturation engine that allows it to perform program optimizations as well as translation validation between program pairs. I designed and implemented the process that solves the PEG Selection Problem, which is an important sub-problem of program optimization with Equality Saturation. Finally, I evaluated Equality Saturation experimentally in terms of its effectiveness at both program optimization and translation validation. To achieve these goals I had to design and construct the two front-ends we use within Peggy, that allow us to apply Equality Saturation to both Java bytecode and LLVM bitcode. I also designed the language for defining the equality analyses that Peggy uses to perform Equality Saturation, and implemented code to parse them and integrate them into the system.

The rest of this dissertation is organized as follows. Chapter 2 presents a more detailed overview of the Equality Saturation technique. Chapter 3 describes the design of the front-ends to the system, which allow it to perform Equality Saturation on real programs. Chapter 4 describes what equality analyses actually are, and how we define and represent them. Chapter 5 presents some of the actual equality analyses used within Peggy, and what they are used for. Chapter 6 talks about complications that arise due to the intermediate representation we use within Peggy, and how we handle those complications. Chapter 7 talks about applying Equality Saturation to the task of optimization of programs. Chapter 8 describes the PEG Selection Problem, which is an important sub-problem when optimizing with Equality Saturation. Chapter 9 presents an experimental evaluation of the effectiveness of using Peggy as an optimization system. Chapter 10 describes how we use Equality Saturation to perform translation validation on pairs of input programs. Chapter 11 discusses related work. Finally, Chapter 12 concludes with a summary of our results.

Chapter 2

Overview

In a traditional compilation system, optimizations are applied sequentially, with each optimization taking as input the program produced by the previous one. This approach to compilation has several well-known drawbacks. One of these drawbacks is that the order in which optimizations are applied affects the quality of the generated code, a problem commonly known as the *phase ordering problem*. Another drawback is that profitability heuristics, which decide whether or not to apply a given optimization, tend to make their decisions one optimization at a time, and so it is difficult for these heuristics to account for the effect of future transformations.

We have designed a new approach for structuring optimizers that addresses the above limitations of the traditional approach, and also has a variety of other benefits. Our approach consists of computing a set of optimized versions of the input program and then selecting the best candidate from this set. The set of candidate optimized programs is computed by repeatedly inferring equivalences between program fragments, thus allowing us to represent the effect of many possible optimizations at once. This, in turn, enables the compiler to delay the decision of whether or not an optimization is profitable until it observes the full ramifications of that decision. Although related ideas have been explored in the context of super-optimizers, as Chapter 11 on related work will point out, super-optimizers typically operate on straight-line code, whereas our approach is meant as a general-purpose compilation paradigm that can optimize complicated control flow structures.

At its core, our approach is based on a simple change to the traditional compilation model: whereas traditional optimizations operate by destructively performing transformations, in our approach optimizations take the form of *equality analyses* that simply add equality information to a common intermediate representation (IR), without losing the original program. Thus, after each equality analysis runs, both the old program and the new program are represented.

The simplest form of equality analysis looks for ways to instantiate equality axioms like $a * 0 = 0$, or $a * 4 = a << 2$. However, our approach also supports arbitrarily complicated forms of equality analyses, such as inlining and various forms of user defined axioms. The flexibility with which equality analyses are defined makes it easy for compiler writers to port their traditional optimizations to our equality-based model: optimizations can work as before, except that whereas the optimization formerly would have performed a transformation, it now simply records the transformation as an equality.

The main technical challenge that we face in our approach is that the compiler's IR must now use equality information to represent not just one optimized version of the input program, but multiple versions at once. We address this challenge through a new IR that compactly represents equality information, and as a result can simultaneously store multiple optimized versions of the input program. After a program is converted into our IR, we repeatedly apply equality analyses to infer new equalities until no more equalities can be inferred, a process known as saturation. Once saturated with equalities, our IR compactly represents the various possible ways of computing the values from the original program modulo the given set of equality analyses (and modulo some bound in the case where applying equality analyses leads to unbounded expansion).

Our approach for structuring optimizers is based on the idea of having optimizations propagate equality information to a common IR that simultaneously represents multiple optimized versions of the input program. The main challenge in designing this IR is that it must make equality reasoning *effective* and *efficient*.

To make equality reasoning *effective*, our IR needs to support the same kind of basic reasoning that one would expect from simple equality axioms like

$a * (b + c) = a * b + a * c$, but with more complicated computations such as branches and loops. We have designed a representation for computations called Program Expression Graphs (PEGs) [TSTL09] that meets these requirements. Similar to the *gated SSA* representation [TP95, Hav93], PEGs are *referentially transparent*, which intuitively means that the value of an expression depends only on the value of its constituent expressions, without any side-effects. As has been observed previously in many contexts, referential transparency makes equality reasoning simple and effective. However, unlike previous SSA-based representations, PEGs are also *complete*, which means that there is no need to maintain any additional representation such as a control flow graph (CFG). Completeness makes it easy to use equality for performing transformations: if two PEG nodes are equal, then we can pick either one to create a program that computes the same result, without worrying about the implications on any underlying representation.

In addition to being effective, equality reasoning in our IR must be *efficient*. The main challenge is that each added equality can potentially double the number of represented programs, thus making the number of represented programs exponential in the worst case. To address this challenge, we record equality information of PEG nodes by simply merging PEG nodes into equivalence classes. We call the resulting equivalence graph an Equivalence PEG, or EPEG, and it is this EPEG representation that we use in our approach. Using equivalence classes allows EPEGs to efficiently represent exponentially many ways of expressing the input program, and it also allows the equality saturation engine to efficiently take into account previously discovered equalities. Among existing IRs, EPEGs are unique in their ability to represent multiple optimized versions of the input program.

We illustrate the main features of our approach by showing how it can be used to implement loop-induction-variable strength reduction. The idea behind this optimization is that if all assignments to a variable `i` in a loop are increments, then an expression `i * c` in the loop (with `c` being loop invariant) can be replaced with `i`, provided all the increments of `i` in the loop are appropriately scaled by `c`.

As an example, consider the code snippet from Figure 2.1(a). The use of `i*5` inside the loop can be replaced with `i` as long as the two increments in the

```
i := 0;
while (...) {
   use(i * 5);
   i := i + 1;
   if (...) {
      i := i + 3;
   }
}
```

(a)

```
i := 0;
while (...) {
   use(i);
   i := i + 5;
   if (...) {
      i := i + 15;
   }
}
```

(b)

Figure 2.1: Loop-induction-variable strength reduction: (a) shows the original code, and (b) shows the optimized code.

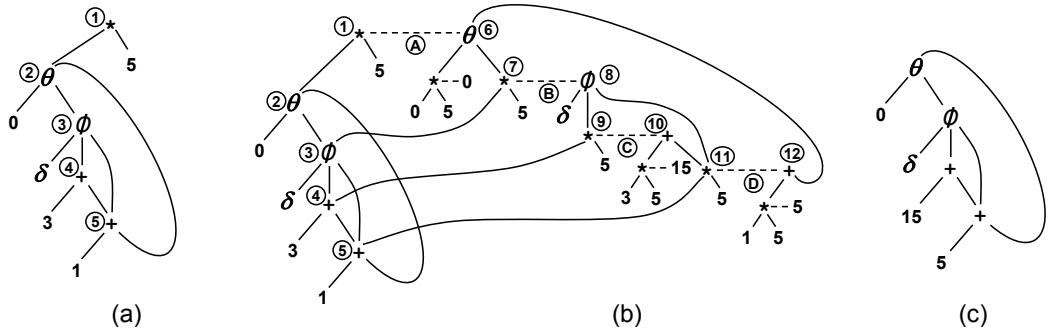


Figure 2.2: Loop-induction-variable Strength Reduction using PEGs: (a) shows the original PEG, (b) shows the EPEG that our engine produces from the original PEG and (c) shows the optimized PEG, which results by choosing nodes 6, 8, 10, and 12 from (b).

loop are scaled by `5`. The resulting code is shown in Figure 2.1(b).

2.1 Representations

Here we briefly recap the descriptions of the custom intermediate representations that we use for performing Equality Saturation. These representations were originally presented in [TSTL09].

2.1.1 Program Expression Graphs

A Program Expression Graph (PEG) is a graph containing: (1) operator nodes, for example "plus", "minus", or any of our built-in nodes for representing conditionals and loops, and (2) "dataflow" edges that specify where operator nodes get their arguments from. As an example, consider the "use" statement in Figure 2.1(a). This is meant as a placeholder for any kind of use of the value `i*5`; it is used to mark the specific location inside the loop where we examine this value. The PEG for the value `i*5` is shown in Figure 2.2(a). At the very top of the PEG we see node 1, which represents the `i*5` multiply operation from inside the loop. Each PEG node represents an operation, with the children nodes being the arguments to the operation. The links from parents to children are shown using solid (non-dashed) lines. For example, node 1 represents the multiplication of node 2 by the constant 5. PEGs follow the notational convention used in E-graphs [NO79, NO80, DNS05] and Abstract Syntax Trees (ASTs) of displaying operators above the arguments that flow into them, which is the opposite convention typically used in Dataflow Graphs [CFR$^+$89, AWZ88]. We use the E-graph/AST orientation because we think of PEGs as recursive expressions.

Node 2 in our PEG represents the value of variable `i` inside the loop, right before the first instruction in the loop is executed. We use θ nodes to represent values that vary inside of a loop. A PEG contains one θ node per variable that is live in the loop, and a variable's θ node represents the entire sequence of values that the variable takes throughout the loop. Intuitively, the left child of a θ node computes the initial value, whereas the right child computes the value at the current iteration in terms of the value at the previous iteration. In our example, the left child of the θ node is the constant 0, representing the initial value of `i`. The right child of the θ node uses nodes 3, 4, and 5 to compute the value of `i` at the current iteration in terms of the value of `i` from the previous iteration. The two plus nodes (nodes 4 and 5) represent the two increments of `i` in the loop, whereas the ϕ node (node 3) represents the merging of the two values of `i` produced by the two plus nodes. In traditional SSA, a ϕ node has only two inputs (the true value and the false value) and as a result the node itself does not know which of the two

inputs to select, relying instead on an explicit control-flow join to know at run-time which case of the branch was taken. In contrast, our ϕ nodes are like those in *gated* SSA [TP95, Hav93]: they take an additional parameter (the first left-most one) which is used to select between the second and the third parameter. As a result, our ϕ nodes are executable by themselves, and so there is no need to explicitly encode a control-flow join. Our example doesn't use the branch condition in an interesting way, and so we just let δ represent the PEG sub-graph that computes the branch condition. Furthermore, since this PEG represents the value of `i` *inside* the loop, it does not contain any operators to describe the `while`-condition, since this information is only relevant for computing the value of `i` after the loop has terminated.

From a more formal point of view, each θ node produces a *sequence* of values, one value for each iteration of the loop. The first argument of a θ node is the value for the first iteration, whereas the second argument is a sequence that represents the values for the remaining iterations. For example, in Figure 2.2, the nodes labeled 3 through 5 compute this sequence of remaining values in terms of the sequence produced by the θ node. In particular, nodes 3, 4 and 5 have been implicitly lifted to operate on this sequence. The fact that a single θ node represents the entire sequence of values that a loop produces allows us to represent that two loops compute the same sequence of values with a single equality between two θ nodes.

PEGs are well-suited for equality reasoning because all PEG operators, even those for branches and loops, are mathematical functions with no side effects. As a result, PEGs are *referentially transparent*, which allows us to perform the same kind of equality reasoning that one is familiar with from mathematics. Though PEGs are related to functional programs, in our work we have used PEGs to represent intra-procedural imperative code with branches and looping constructs. Furthermore, even though all PEG operators are pure, PEGs can still represent programs with state by using heap summary nodes: stateful operations, such as heap reads and writes, can take a heap as an argument and return a new heap. This functional representation of stateful programs allows our Peggy compiler to use

PEGs to reason about Java programs. The heap summary node can also be used to encode method/function calls in an intra-procedural setting by simply threading the heap summary node through special nodes representing method/function calls. We discuss the heap summary node in more detail in Chapter 6.

2.1.2 Equivalence PEGs

A PEG by itself can only represent a single way of expressing the input program. To represent *multiple* optimized versions of the input program, we need to encode equalities in our representation. To this end, an EPEG is a graph that groups together PEG nodes that are equal into equivalence classes. As an example, Figure 2.2(b) shows the EPEG that our engine produces from the PEG of Figure 2.2(a). We display equalities graphically by adding a dashed edge between two nodes that have become equal. These dashed edges are only a visualization mechanism. In reality, PEG nodes that are equal are grouped together into an equivalence class.

Reasoning in an EPEG is done through the application of optimizations, which in our approach take the form of equality analyses that add equality information to the EPEG. An equality analysis consists of two components: a trigger, which is an expression pattern stating the kinds of expressions that the analysis is interested in, and a callback function, which should be invoked when the trigger pattern is found in the EPEG. The saturation engine continuously monitors all the triggers simultaneously, and invokes the necessary callbacks when triggers match. When invoked, a callback function adds the appropriate equalities to the EPEG.

The simplest form of equality analysis consists of instantiating axioms such as $a*0 = 0$. In this case, the trigger would be $a*0$, and the callback function would add the equality $a*0 = 0$. Even though the vast majority of our reasoning is done through such declarative axiom application, our trigger and callback mechanism is much more general, and has allowed us to implement equality analyses such as inlining, tail-recursion elimination, and constant folding.

The following three axioms are the equality analyses required to perform loop-induction-variable strength reduction. They state that multiplication dis-

tributes over addition, θ, and ϕ:

$$(a + b) * m = a * m + b * m \tag{2.1}$$

$$\theta(a, b) * m = \theta(a * m, b * m) \tag{2.2}$$

$$\phi(a, b, c) * m = \phi(a, b * m, c * m) \tag{2.3}$$

After a program is converted to a PEG, a saturation engine repeatedly applies equality analyses until either no more equalities can be added, or a bound is reached on the number of expressions that have been processed by the engine.

Figure 2.2(b) shows the saturated EPEG that results from applying the above distributivity axioms, along with a simple constant folding equality analysis. In particular, distributivity is applied four times: axiom (2.2) adds equality edge A, axiom (2.3) edge B, axiom (2.1) edge C, and axiom (2.1) edge D. Our engine also applies the constant folding equality analysis to show that $0 * 5 = 0$, $3 * 5 = 15$ and $1 * 5 = 5$. Note that when axiom (2.2) adds edge A, it also adds node 7, which then enables axiom (2.3). Thus, equality analyses essentially communicate with each other by propagating equalities through the EPEG. Furthermore, note that the instantiation of axiom (2.1) adds node 12 to the EPEG, but it does not add the right child of node 12, namely $\theta(\ldots) * 5$, because it is already represented in the EPEG.

Once saturated with equalities, an EPEG compactly represents multiple optimized versions of the input program – in fact, it compactly represents all the programs that could result from applying the optimizations in any order to the input program. For example, the EPEG in Figure 2.2(b) encodes 128 ways of expressing the original program (because it encodes 7 independent equalities, namely the 7 dashed edges). In general, a single EPEG can efficiently represent exponentially many ways of expressing the input program.

After saturation, a global profitability heuristic can pick which optimized version of the input program is best. Because this profitability heuristic can inspect the entire EPEG at once, it has a global view of the programs produced by various optimizations, *after* all other optimizations were also run. In our example, starting at node 1, by choosing nodes 6, 8, 10, and 12, we can construct the graph

in Figure 2.2(c), which corresponds exactly to performing loop-induction-variable strength reduction in Figure 2.1(b).

More generally, when optimizing an entire function, one has to pick a node for the equivalence class of the return values and nodes for all equivalence classes that the return values depend on. There are many plausible heuristics for choosing nodes in an EPEG. In our Peggy implementation, we have chosen to select nodes using an Integer Linear Programming (ILP) solver. In particular, we use an ILP solver and a static cost model for every node to compute the lowest-cost program that is encoded in the EPEG. In the example from Figure 2.2, the ILP solver picks the nodes described above. Chapter 8 describes our technique for selecting nodes in more detail.

2.2 Benefits of our Approach

Our approach of having optimizations add equality information to a common IR until it is saturated with equalities has a variety of benefits over previous optimization models.

2.2.1 Optimization Order Does Not Matter

The first benefit of our approach is that it removes the need to think about optimization ordering. When applying optimizations sequentially, ordering is a problem because one optimization, say A, may perform some transformation that will irrevocably prevent another optimization, say B, from triggering, when in fact running B first would have produced the better outcome. This so-called *phase ordering problem* is ubiquitous in compiler design. In our approach, however, the compiler writer does not need to worry about ordering, because optimizations do not destructively update the program – they simply add equality information. Therefore, after an optimization A is applied, the original program is still represented (along with the transformed program), and so any optimization B that could have been applied before A is still applicable after A. Thus, there is no way that applying an optimization A can irrevocably prevent another optimization B

from applying, and so there is no way that applying optimizations will lead the search astray. As a result, compiler writers who use our approach do not need to worry about the order in which optimizations run. Better yet, because optimizations are allowed to freely interact during equality saturation, without any consideration for ordering, our approach can discover intricate optimization opportunities that compiler writers may not have anticipated, and hence would not have implemented in a general purpose compiler.

To understand how our approach addresses the phase ordering problem, consider a simple peephole optimization that transforms `i * 5` into `i << 2 + i`. On the surface, one may think that this transformation should always be performed if it is applicable – after all, it replaces a multiplication with the much cheaper shift and add. In reality, however, this peephole optimization may disable other more profitable transformations. The code from Figure 2.1(a) is such an example: transforming `i * 5` to `i << 2 + i` disables loop-induction-variable strength reduction, and therefore generates code that is worse than the one from Figure 2.1(b).

The above example illustrates the phase ordering problem. In systems that apply optimizations sequentially, the quality of the generated code depends on the order in which optimizations are applied. Whitfield and Soffa [WS97a] have shown experimentally that enabling and disabling interactions between optimizations occur frequently in practice, and furthermore that the patterns of interaction vary not only from program to program, but also within a single program. Thus, no one order is best across all compilation.

A common partial solution consists of carefully considering all the possible interactions between optimizations, possibly with the help of automated tools, and then coming up with a carefully tuned sequence for running optimizations that strives to enable most of the beneficial interactions. This technique, however, puts a heavy burden on the compiler writer, and it also does not account for the fact that the best order may vary between programs.

At high levels of optimizations, some compilers may even run optimizations in a loop until no more changes can be made. Even so, if the compiler picks the

wrong optimization to start with, then no matter what optimizations are applied later, in any order, any number of times, the compiler will not be able to reverse the disabling consequences of the first optimization.

In our approach, the compiler writer does not need to worry about the order in which optimizations are applied. The previous peephole optimization would be expressed as the axiom `i * 5 = i << 2 + i`. However, unlike in a traditional compilation system, applying this axiom in our approach does not remove the original program from the representation — it only adds information — and so it cannot disable other optimizations. Therefore, the code from Figure 2.1(b) would still be discovered, even if the peephole optimization was run first. In essence, our approach is able to simultaneously explore all possible sequences of optimizations, while sharing work that is common across the various sequences.

In addition to reducing the burden on compiler writers, removing the need to think about optimization ordering has two additional benefits. First, because optimizations interact freely with no regard to order, our approach often ends up combining optimizations in unanticipated ways, leading to surprisingly complicated optimizations given how simple our equality analyses are. Second, it makes it easier for end-user programmers to add domain-specific axioms to the compiler, because they don't have to think about where exactly in the compiler the axiom should be run, and in what order relative to other optimizations.

2.2.2 Global Profitability Heuristics

The second benefit of our approach is that it enables *global profitability heuristics*. Even if there existed a perfect order to run optimizations in, compiler writers would still have to design profitability heuristics for determining whether or not to perform certain optimizations such as inlining. Unfortunately, in a traditional compilation system where optimizations are applied sequentially, each heuristic decides in isolation whether or not to apply an optimization at a particular point in the compilation process. The local nature of these heuristics makes it difficult to take into account the effect of future optimizations.

Since profitability heuristics in traditional compilers tend to be local in

nature, it is difficult to take into account the effect of future optimizations. For example, consider inlining. Although it is straightforward to estimate the *direct cost* of inlining (the code-size increase) and the *direct benefit* of inlining (the savings from removing the call overhead), it is far more difficult to estimate the potentially larger *indirect benefit*, namely the additional optimization opportunities that inlining exposes.

To see how inlining would affect our running example, consider again the code from Figure 2.1(a), but assume that instead of `use(i * 5)`, there was a call to a function `f`, and the use of `i*5` occurred *inside* `f`. If `f` is sufficiently large, a traditional inliner would not inline `f`, because the code bloat would outweigh the call-overhead savings. However, a traditional inliner would miss the fact that it may still be worth inlining `f`, despite its size, because inlining would expose the opportunity for loop-induction-variable strength reduction. One solution to this problem consists of performing an *inlining trial* [DC94], where the compiler simulates the inlining transformation, along with the effect of subsequent optimizations, in order to decide whether or not to actually inline. However, in the face of multiple inlining decisions (or more generally multiple optimization decisions), there can be exponentially many possible outcomes, each one of which has to be compiled separately.

In our approach, on the other hand, inlining simply adds an equality to the EPEG stating that the call to a given function is equal to its body instantiated with the actual arguments. The resulting EPEG simultaneously represents the program where inlining is performed and where it is not. Subsequent optimizations then operate on both of these programs at the same time. More generally, our approach can simultaneously explore exponentially many possibilities in parallel, while sharing the work that is redundant across these various possibilities. In the above example with inlining, once the EPEG is saturated, a global profitability heuristic can make a more informed decision as to whether or not to pick the inlined version, since it will be able to take into account the fact that inlining enabled loop-induction-variable strength reduction.

Our approach, allows the compiler writer to design profitability heuristics

that are global in nature. In particular, rather than choosing whether or not to apply an optimization locally, these heuristics choose between fully optimized versions of the input program. Our approach makes this possible by separating the decision of whether or not a transformation is *applicable* from the decision of whether or not it is *profitable*. Indeed, using an optimization to add an equality in our approach does not indicate a decision to perform the transformation – the added equality just represents the *option* of picking that transformation later. The actual decision of which transformations to apply is performed by a global heuristic *after* our IR has been saturated with equalities. This global heuristic simply chooses among the various optimized versions of the input program that are represented in the saturated IR, and so it has a global view of all the transformations that were tried and what programs they generated.

There are many ways to implement this global profitability heuristic, and in our prototype compiler we have chosen to implement it using an Integer Linear Programming (ILP) solver. In particular, after our IR has been saturated with equalities, we use an ILP solver and a static cost model for every node to pick the lowest-cost program that computes the same result as the original program.

2.2.3 Translation Validation

The third benefit of our approach is that it can be used not only to optimize programs, but also to prove programs equivalent: intuitively, if during saturation an equality analysis finds that the return values of two programs are equal, then the two programs are equivalent. Our approach can therefore be used to perform *translation validation*, a technique that consists of automatically checking whether or not the optimized version of an input program is semantically equivalent to the original program. For example, we can prove the correctness of optimizations performed by existing compilers, even if our profitability heuristic would not have selected those optimizations. In this way, our approach can be used to perform translation validation for any compiler (not just our own), by checking that each function in the input program is equivalent to the corresponding optimized function in the output program.

For example, our approach would be able to show that the two program fragments from Figure 2.1 are equivalent. Furthermore, it would also be able to validate a compilation run in which `i * 5 = i << 2 + i` was applied first to Figure 2.1(a). This shows that we are able to perform translation validation regardless of what optimized program our own profitability heuristic would choose.

Although our translation validation technique is intraprocedural, we can use interprocedural equality analyses such as inlining to enable a certain amount of interprocedural reasoning. This allows us to reason about transformations like reordering function calls. Chapter 10 discusses translation validation in greater detail.

Summary

Equality Saturation has many advantages over traditional, linear optimization techniques. Not only does it explore a much larger portion of the program space, but it does so in a way that requires no ordering on the individual optimizations that are applied. Furthermore, it separates the notions of optimization applicability from optimization profitability, which allows a single global profitability heuristic to run after saturation has completed, which can take all possible programs into consideration.

In the next chapter, we will discuss the starting point of the Peggy pipeline. Namely, we will describe the front-ends to Peggy, which take programs written in imperative code and convert them to a representation that we can more easily manipulate. This allows Peggy to perform Equality Saturation on real, concrete programs.

Acknowledgements

Portions of this research were funded by the US National Science Foundation under NSF CAREER grant CCF-0644306.

This chapter contains material taken from “Equality Saturation: a New

Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Logical Methods in Computer Science 2010*. The dissertation author was the secondary investigator and author of this paper.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. The dissertation author was the secondary investigator and author of this paper.

Chapter 3

Frontends

In order to use Peggy as an optimizer, we must be able to convert code from an existing programming language into PEGs and EPEGs that Peggy can manipulate. This requires a separate front-end for each programming language that is targetted. In this section, we present a detailed description of these front-ends and how we go about representing these languages in a manner that Peggy can handle.

3.1 Language-Independent Components

In the Java implementation of Peggy, we represent PEGs using instances of the class `PEGInfo<L,P,R>`. This class is abstracted to take 3 generic type parameters, which allows us to use this same class to represent PEGs for any source language. The contents of a `PEGInfo<L,P,R>` instance include an instance of a `CRecursiveExpressionGraph<FlowValue<P,L>>`. This is a directed (possibly cyclic) graph object where every node has ordered outgoing edges, and where the nodes have labels of type `FlowValue<P,L>`. The `FlowValue` is used to represent the mathematical operator of the node, or the function that the node encodes. It is similarly abstracted to take 2 type parameters so that a `FlowValue` will be suitable to represent any kind of function for any source language. These classes make up the language-independent portion of our PEG representation.

A `FlowValue<P,L>` can represent any function of any language, but there

are also some language-independent functions that are used in PEGs for every language. For example, θ_i, eval_i, pass_i, and ϕ are all language-independent operators. Furthermore, since a PEG typically represents an entire function, there must be a way to represent the input parameters to the function. These include the method's formal parameters. For stateful languages, there will also be an input heap summary node σ that is taken as an input parameter. In Java, non-static methods have an additional input parameter to represent the `this` reference. Aside from the parameters and the language-independent operators, there will be language-dependent operators. The `L` and `P` type parameters are used to specify the classes used to define the language-dependent operators and input parameters, respectively. So, a `FlowValue<P,L>` instance represents either: (1) a language-dependent operator of type `L`, (2) a (language-dependent) input parameter of type `P`, or (3) one of several pre-defined language-independent operators such as ϕ.

The `CRecursiveExpressionGraph<FlowValue<P,L>>` represents the PEG graph, but more information than that is included in an instance of `PEGInfo<L, P, R>`. Specifically, the `R` type parameter is used to define the return values of the PEG. In a procedural language like Java, each PEG logically has two return values. The first is the standard return value of the Java method (where a special null value is used for void functions). The second is the output heap summary node, σ. One can imagine languages where the σ return is not needed, or where additional return values are defined. The `R` type parameter specifies a class that will identify the set of all return values defined for the source language. In addition to the graph, the `PEGInfo<L,P,R>` contains a mapping from `R` instances to nodes of the graph. These special nodes are the ones that compute the return values, and form the roots of the graph.

3.2 Java Bytecode

The Java programming language was our first target for optimization. Java is an object-oriented language that is typically compiled down to a machine-independent intermediate bytecode, in one or more ".class" files. Each Java class

produces one class file, and the bytecode instructions within each class file are stack-based. Each method call gets its own isolated operand stack at runtime, and a typical bytecode instruction will pop its input operands off the stack and push its result onto the stack afterward. Each entry in the operand stack is 32 bits wide, and hence can hold one integer, one float, one opaque reference pointer, or half of a long integer or double precision float. Similarly, the are a maximum of 65536 local variable slots per method which can contain the same kind of data as the operand stack. These slots initially hold the method's actual parameters, and bytecode instructions are used to load and store from the local variables. Since there are no explicit pointers in Java, one cannot get the address of any stack location or local variable. Hence, aliasing is impossible between local variables and stack locations, so all local variables and stack positions can be abstracted away when converting to a PEG.

We use the Soot Java optimization framework [VRHS+99] to parse the Java bytecode files and manipulate them at the lowest level. Soot is itself written in Java, which allows easy integration with Peggy, which is also written in Java. Soot converts the raw bytecode into a stackless 3-address code called Jimple. The Jimple code contains instructions, which inherit from class `soot.jimple.Stmt`, and values, which inherit from class `soot.Value`. It will also build a CFG of `Stmt`'s, which makes translation to the PEG representation much easier.

The Java frontend uses the classes `JavaLabel`, `JavaParameter`, and `JavaReturn` in place of the `L`, `P`, and `R` type parameters mentioned above. The `JavaParameter` class has subclasses to represent formal parameters, the `this` reference, and the heap summary node. The `JavaReturn` class has subclasses to represent the method's Java return value, and the output heap summary node. The `JavaLabel` has several subclasses, all of which are detailed in the translation described in Appendix A. This table describes how the Java bytecode instructions are translated to Soot objects, and then how those Soot objects are converted to PEG nodes using `JavaLabel` and `JavaParameter` instances.

There are a few interesting quirks about Java bytecode that we must address in our translation. First of all, even though java has several integral primitive

types (`short, boolean, char, int, byte, long`), all of these except for `long` are represented as `int`'s at runtime. This actually makes our representation simpler, since we have fewer distinct types to worry about.

Secondly, the Java language has exception handling, and several of the Java bytecode instructions may throw exceptions at runtime. For every PEG node that represents an operation that could potentially throw an exception, the result of that operator must be a disjoint union. The result of the instruction is either the normal return value, or a value representing the exception object that the operator threw. So the `INVOKEVIRTUAL` operator, for instance, has a type of $(\sigma, E|V)$, where σ represents the set of heap summaries, E represents the set of exception objects, and V represents the set of all first-class Java values. We also introduce two new operators: `IsException[type]`, which tests a disjoint union to see if it is an exception value, and `RHO_EXCEPTION`, to extract the exception value from a disjoint union. Hence, every operator from the table in Appendix A that has a `RHO_VALUE` on it may also have a `RHO_EXCEPTION` on it. While we can easily convert exception-handling code to a PEG representation, it is extremely difficult to revert it back to an exceptional CFG. We do not currently have a reversion algorithm that supports this, and hence we cannot revert any methods that contain exception-handling code.

3.3 LLVM Bitcode

The other frontend we currently have implemented is one for the Low-Level Virtual Machine (LLVM) [llv]. LLVM is a strongly-typed pseudo-assembly language, where the instructions are implicitly divided into basic blocks that are in static single-assignment form. LLVM has a fixed type system that includes integers of any bit-width, 5 different types of floating-point numbers, pointers, arrays, vectors, C-like structs, functions, and basic blocks (labels). LLVM uses structural type equivalence, and allows recursive types (but only through pointers). In LLVM all functions are globally defined. LLVM has global variables and global aliases, which are simply indirect references to globals.

There are a few constructs in LLVM that we cannot represent in a PEG. First of all, LLVM allows you to take the address of any basic block as a first-class label value. The only use of this label is as an operand to a `Call` instruction, when the function being called is a piece of inline target-dependent assembly code. These labels cannot be represented in a PEG because their usage in the assembly hides some of the control flow of the function. Similarly, in LLVM 2.8 there is a new instruction called `IndirectBranch`, which takes a different kind of basic block label as a parameter. However, the former label type is not a first-class value, and can only be used as an argument to a `Call` instruction, whereas the latter is represented as an `i8*`, and is treated as a general first-class value. Hence, the `IndirectBranch` instruction also obscures the normal control flow of the function, so we cannot represent it in a PEG.

Like Java, LLVM has explicit exception handling. The throw is accomplished with the `Unwind` instruction, which takes no parameters. Hence there is no exception value associated with the throw, just the change in control flow. The analog to 'catch' is the `Invoke` instruction, which jumps to one of two basic blocks afterwards, depending on whether the callee terminated normally or whith an exception. If a function is called with a `Call` instruction rather than `Invoke`, then an `Unwind` will terminate the callee and the caller, continuing up the call stack until an `Invoke` is found. Since LLVM has only one type of exception, it is much simpler to represent in a PEG and hence simple enough to revert back to a CFG. Hence, our current implementation does support LLVM functions with exception-handling behavior.

Unlike for Java, we have no analog of the Soot library to process LLVM at the lowest level. The LLVM bitcode manipulation code was written directly into Peggy. An LLVM PEG is represented by an instance of `PEGInfo<LLVM-Label,LLVMParameter,LLVMReturn>`. The `LLVMParameter` class has subclasses to represent function parameters and the input heap summary node (unlike Java, there are no objects, and hence no 'this' reference). The `LLVMReturn` class has subclasses to represent the function's LLVM return value and the output heap summary value. The `LLVMLabel` class represents all the operators defined in the

tables from Appendix B.

Summary

We have seen how our two target languages – Java and LLVM – are converted from their default representations into a PEG that can be used by the Equality Saturation engine. We saw how we can represent a PEG from any target programming language using the same set of classes; namely `PEGInfo`, `FlowValue`, and `CRecursiveExpressionGraph`. We also saw how we perform the conversion from Java to PEG and LLVM to PEG. For Java, we leverage the Soot Jimple representation, which does the work of converting the stack-based bytecode into a three-address code that is easier to handle. For LLVM, the bitcode is already is a pseudo-assembly format and already divided into basic blocks, which makes it easy to convert directly to a PEG.

In the next chapter we will look at how the axioms for the Equality Saturation engine are defined. The axioms, in addition to the PEG provided by the front-end, comprise the other main input to the engine. The two of them together define the environment in which the engine runs, and the axioms define the set of equalities that the engine is able to deduce.

Chapter 4

Defining Equality Analyses

The Equality Saturation engine functions by adding nodes to the EPEG and finding equivalences between them. This is accomplished through the use of *equality analyses*. An equality analysis acts on the EPEG in two phases; the trigger and the response. In the trigger phase, the analysis looks for a particular pattern of nodes, edges, and equivalences within the EPEG. In the response phase, it adds new nodes and equivalences to the EPEG, based on the information from the trigger phase. For instance, a very simple analysis could look for a '+' node with two children (call them L and R) as its trigger phase, and then in its response phase it could create a new '+' node that points to R and L (order reversed) and add an equivalence between the '+' nodes. This encodes the commutativity of '+' in a way that the EPEG can use.

In addition to standard nodes inside the EPEG, there are two special distinguished nodes that are used specifically in equality analyses. They are the `true` and `false` nodes. They have no children, and their operator labels represent boolean true and false, respectively. While simple, these two nodes allow a great deal of complicated analysis to occur inside the EPEG. For instance, an equality analysis may now take a node representing a relational operator (such as "$\geq$") and make it equivalent to the `true` node to show that the result of the relational operation is known to be true. Furthermore, the trigger phase of an equality analysis can use equivalence-to-true or equivalence-to-false as a predicate over nodes. This allows complicated constant folding and branch folding to occur. In addition, the `true`

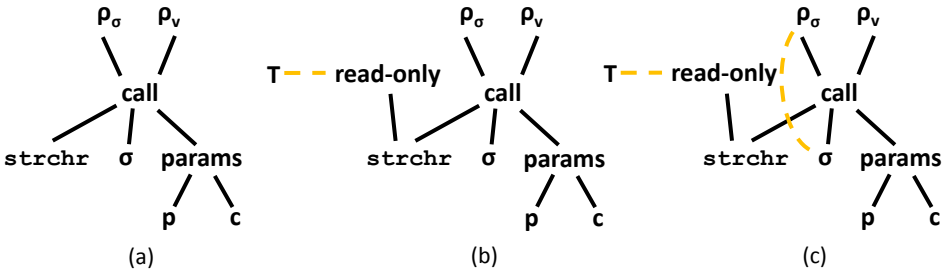


Figure 4.1: An axiom making use of the `true` node.

and `false` nodes allow the EPEG to encode and propagate facts about the nodes that are deduced during saturation.

As an example, consider the PEG in Figure 4.1. Part (a) depicts a function call to the libc function `strchr`, which is a read-only function that does not modify memory. We can add this fact to the EPEG by adding a "read-only" node on top of the `strchr` node, and making it equivalent to `true`, as shown in part (b). The "read-only" label is then not so much a function label as a predicate label. It represents the predicate "X is a read-only function". Then the fact that this is equivalent to `true` is an assertion that read-only(`strchr`) is true. This is useful because future analyses can now detect this assertion in the EPEG. For instance, an analysis could encode the fact that any `call` operator whose function node has the "read-only" assertion is a read-only call, which means that its input heap summary node is equivalent to its output heap summary node, as seen in part (c).

4.1 Axioms

The simplest kind of equality analyses are what we call *axioms*. Axioms are defined in terms of *expressions*. An expression is a rooted digraph meant to resemble a sub-PEG. The nodes of an expression are either operator nodes or meta-variable nodes. An operator node, like a PEG node, has an operator label and an ordered list of 0 or more child nodes. A meta-variable node has no children and no operator label, and is meant to represent "any" EPEG node. Expressions are used for pattern-matching against groups of EPEG nodes in the trigger phase of

an axiom. An operator node E will successfully pattern-match against an EPEG node N if the following hold: (1) E and N have the same operator label, (2) E and N have the same number of children, and (3) the n-th child of E pattern-matches against the n-th child of N, for all n. A meta-variable node can match against any EPEG node. So in the '+' example above, the trigger expression could be an operator node with a '+' label, with two meta-variable nodes as children.

The trigger phase of an axiom is divided into 4 sections, each of which contains a set of expressions. The first section is the *exists*, which is a set of expressions that must "exist", or successfully pattern-match in the EPEG. Note that expressions may share meta-variables, and the mapping from meta-variable to EPEG node is shared across all expressions in the entire axiom. The second section is the *trues*, which is a set of expressions that must exist and also be equivalent to the `true` node. Similarly, the third section is the *falses*, which must exist and be equivalent to the `false` node. Finally, the *invariants* are set of expressions that must exist and match against nodes that are invariant w.r.t. a given loop depth index i.

The response phase of an axiom only applies if the trigger phase is completely satisfied. As mentioned before, expressions may share meta-variables. However, the treatment of meta-variables differs from the trigger phase to the response phase. In the trigger phase, the meta-variables are introduced and bound by a successful match. In the response phase, no new meta-variables may be introduced, and references to any meta-variables must have been already bound in the trigger phase.

The response phase is also divided into 4 sections. The first section is the *creates*, which is a set of expressions that will be created within the EPEG. Any operator nodes in a *creates* expression will be newly-created in the EPEG, and any meta-variables will be replaced by the EPEG nodes they matched in the trigger phase. The second section is the *trues*, which is analogous to the *trues* section of the trigger phase. The *trues* section of the response phase is a set of expressions that will be created and then made equivalent to the `true` node. Similarly, the third section is the *falses*, which will be created and made equivalent to the `false` node.

Finally, the fourth section is the *equalities*, which is a set of pairs of expressions. Each expression in each pair will be created, and then the two resulting EPEG nodes will be made equivalent.

One important detail about the response phase is the matter of node reuse. When we say that an expression is "created", we implicitly mean that it is only created if necessary. If a set of nodes already exists within the EPEG that completely matches the set that would be created by an expression, then those nodes are used and no new ones are created. This optimization is critical to the efficiency of the equality saturation engine. If duplicate groups of nodes are created, then any future analysis that is triggered by one will be triggered by the other. This means that the analysis does double the work and produces yet more duplicate groups of nodes. This causes a vicious cycle of duplication which will bloat the EPEG and greatly reduce the useful work done by the saturation engine.

4.1.1 Creating Axioms

Though there are many built-in axioms used within Peggy, it is also possible to allow the user to define his own axioms. Peggy is designed to be an extensible compiler, and hence we wish to make the process of adding user-defined axioms as painless as possible. To this end, I have designed a simple XML-based specification language to describe the two phases of an axiom. XML was a logical choice for this language, because it allows for flexibility as well as human-readability. It also allows the easy creation of cyclic expressions by using 'id' attributes to refer to other nodes. The toplevel design of an XML axiom definition is given in Figure 4.2.

The grammar for the expression tags is somewhat domain-specific, since it must describe operators that are specific to a given source language. However, there are some common domain-independent tags that are used in every source language. These common tags are shown in Figure 4.3. In addition to these tags, there is a common system for referencing other nodes within the XML specification. Any expression tag may have an optional 'id' attribute, which allows the user to assign a unique id string to an expression node. This id may be used later to reference that node with the `<ref id="..."/>` tag. The id attribute of a `ref` tag

```
<rule name='an arbitrary label for the axiom'>
   <trigger>
      <!-- All sections of the trigger go in here -->
      <exists>
         <!-- Expression tags for the "exists" section -->
      </exists>
      <trues>
         <!-- Expression tags for the "trues" section -->
      </trues>
      <falses>
         <!-- Expression tags for the "falses" section -->
      </falses>
      <invariant index='a positive integer'>
         <!-- Expression tags for the "invariants" section,
              for the given loop depth index. There can be
              one invariant tag for each loop depth index -->
      </invariant>
   </trigger>

   <response>
      <!-- All sections of the response go in here -->
      <creates>
         <!-- Expression tags for the "creates" section -->
      </creates>
      <trues>
         <!-- Expression tags for the "trues" section -->
      </trues>
      <falses>
         <!-- expression tags for the "falses" section -->
      </falses>
      <equalities>
         <!-- pairs of expression tags for
              the "equalities" section -->
      </equalities>
   </response>
</rule>
```

Figure 4.2: XML grammar for axiom description.

```
<theta index='###'>
   <!-- Represents a theta node.
        Expects 2 child expression tags. -->
</theta>

<eval index='###'>
   <!-- Represents an eval node.
        Expects 2 child expression tags. -->
</eval>

<pass index='###'>
   <!-- Represents a pass node.
        Expects 1 child expression tag. -->
</pass>

<phi>
   <!-- Represents a phi node.
        Expects 3 child expression tags. -->
</phi>

<ref id='...'/>
<!-- References an expression that was
     defined earlier. The id attribute
     is mandatory. This tag can have
     no children. -->

<variable/>
<!-- Creates a new meta-variable node.
     This tag can have no children. -->
```

Figure 4.3: The XML expression tags that are common to every source language.

refers to an existing id rather than creating a new one. The scope of an id label is every expression that is later in the "document ordering" of the XML specification, within the same axiom. This means any element whose start tag occurs after the given element's start tag. This allows for the creation of cyclic expression nodes, as shown in Figure 4.4.

The expression tags specific to the Java source language are given in Figure 4.5. In addition to the attributes shown in the figure, every tag may also have an id attribute, as described above. The `op` tag may have any of the follow as the value of its `value` attribute: primitivecast, cast, arraylength, getfield, getstat-

```
<theta index='1' id='TOP'>
   <intconstant value='0'/>
   <op value='add'>
      <ref id='TOP'/>
      <intconstant value='1'/>
   </op>
</theta>
<!-- This expression represents a common
     structure seen in a PEG, which is a counter
     that starts at 0 and increments by 1 every
     iteration.
-->
```

Figure 4.4: Example of a cyclic expression. The id of the `theta` expression is referenced in the `ref`, which creates a cycle.

icfield, getarray, instanceof, injr, void, add, sub, mul, div, mod, cmpl, cmpg, gte, gt, lte, lt, eq, ne, and, or, shl, shr, ushr, xor, neg, params, invokestatic, invokevirtual, invokeinterface, invokespecial, newarray, newmultiarray, dims, newinstance, entermonitor, exitmonitor, setarray, setfield, setstaticfield, throw, rho_value, or rho_sigma.

The expression tags for the LLVM source language are given in Figures 4.6 and 4.7. In addition to the attributes shown in the figure, every tag may also have an id attribute, as described above. The `op` tag may have any of the follow as the value of its `value` attribute: injr, call, tailcall, invoke, rho_value, rho_sigma, rho_exception, shufflevector, insertelement, getelementptr, indexes, select, extractelement, getresult, malloc, free, alloca, volatile_load, load, volatile_store, store, params, unwind, void, returnstructure, vaarg, is_exception, insertvalue, extractvalue, and offsets.

This tag-based language gives the ability to specify a new axiom by explicitly defining the trigger and response expressions. It is expressive enough to describe any expression, but it is lacking somewhat in brevity. For example, consider the axiom in Figure 4.8. This axiom encodes a fact about the relationship between the `extractelement` instruction and the `insertelememt` instruction, which get and set elements of a vector, respectively. It encodes the fact that an extract operation can effectively ignore an earlier insert operation, if it is known that the

```
<method name='...' class='...' signature='...'/>
<!-- Describes a Java method, for use by operators
     like INVOKEVIRTUAL. No children. -->

<field name='...' class='...' signature='...'/>
<!-- Describes a Java field, for use by operators
     like GETSTATIC. No children. -->

<type value='...'/>
<!-- Describes a Java type, for use by operators
     like CAST. No children. -->

<intconstant value='...'/>
<!-- Describes a literal int constant. No children. -->

<longconstant value='...'/>
<!-- Describes a literal long constant. No children. -->

<floatconstant value='...'/>
<!-- Describes a literal float constant. No children. -->

<doubleconstant value='...'/>
<!-- Describes a literal double constant. No children. -->

<stringconstant value='...'/>
<!-- Describes a literal string constant. No children. -->

<nullconstant/>
<!-- Describes the constant null reference. No children. -->

<op value='...'>
   <!-- Describes a simple Java operation, which is
        one of the operators defined by the SimpleJavaLabel
        class mentioned above. This tag must have the same number of
        children as the operator named in the 'value' attribute.
   -->
</op>
```

Figure 4.5: The expression tags defined for Java.

```
<function name='...' signature='...'/>
<!-- Describes a function header, with the given
     name and function signature. No children. -->

<global name='...' type='...'/>
<!-- Describes a global variable, with the given
     name and type. No children. -->

<alias name='...' type='...'/>
<!-- Describes an alias value, with the given
     name and type. No children. -->

<type value='...'/>
<!-- Describes an LLVM type. No children. -->

<numeral value='...'/>
<!-- Describes a numerical value that is not an
     LLVM value. For example, the calling convention
     id number for the INVOKE instruction will be
     a numeral. No children. -->

<undefconstant type='...'/>
<!-- Describes an undef value with the given type.
     No children. -->

<intconstant width='...' value='...'/>
<!-- Describes a literal integer constant, with the
     given bit-width and value. No children. -->

<floatconstant value='...'/>
<!-- Describes a literal float constant, with the
     given value. No children. -->

<doubleconstant value='...'/>
<!-- Describes a literal double constant, with the
     given value. No children. -->

<stringconstant value='...'/>
<!-- Describes a literal string constant (of type [N x i8]).
     No children. -->
```

Figure 4.6: The constant expression tags defined for LLVM.

```
<binop type='...'>
   <!-- Describes a binary operator, where the type is
        one of: add, sub, mul, udiv, sdiv, fdiv, urem,
        srem, frem, shl, ashr, ushr, and, or, xor.
        Expects 2 children. -->
</binop>

<icmp type='...'>
   <!-- Describes an integer comparison operator, where
        the type is one of: eq, ne, ugt, uge, ult, ule,
        sgt, sge, slt, sle. Expects 2 children. -->
</icmp>

<fcmp type='...'>
   <!-- Describes a floating-point comparison operator,
        where the type is one of: false, oeq, ogt, oge,
        olt, ole, one, ord, uno, ueq, ugt, uge, ult,
        ule, une, true. Expects 2 children. -->
</fcmp>

<cast type='...'>
   <!-- Describes a type casting operator, where
        the type is one of: trunc, zext, sext, fptoui,
        fptosi, uitofp, sitofp, fptrunc, fpext, ptrtoint,
        inttoptr, bitcast. Expects 2 children. -->
</cast>

<nullconstant>
   <type value='...'/>
   <!-- Describes a null constant of a particular type,
        where the type is given by the type tag child. -->
</nullconstant>

<op value='...'>
   <!-- Describes a simple LLVM operation, which is
        one of the operators defined by the SimpleLLVMLabel
        class mentioned above. This must have the same number of
        children as the operator named in the 'value' attribute.
   -->
</op>
```

Figure 4.7: The non-constant expression tags defined for LLVM.

```
<rule>
   <trigger>
      <exists>
         <op value='extractelement' id='E'>
            <op value='insertelement'>
               <variable id='V'/>
               <variable/>
               <variable id='I'/>
            </op>
            <variable id='J'/>
         </op>
      </exists>
      <falses>
         <op value='eq'>
            <ref id='I'/>
            <ref id='J'/>
         </op>
      </falses>
   </trigger>

   <response>
      <equalities>
         <ref id='E'/>
         <op value='extractelement'>
            <ref id='V'/>
            <ref id='J'/>
         </op>
      </equalities>
   </response>
</rule>
```

Figure 4.8: Tag-based XML description of an axiom.

two operations used different indexes. This is an important axiom for simplifying vector expressions, and it encodable in the tag-based axiom language, but it is clearly quite verbose.

Simple Axiom Language. The verbosity of the tag-based language gave rise to a second axiom specification language that specifically addresses this issue. It stemmed mostly from the fact that I noticed I was essentially rewriting the axiom in simpler terms inside the `name` attribute of the `rule` tag. For the axiom from Figure 4.8, I had given it the name string `"extract(insert(V,_,I),J) = extract(V,J), if !eq(I,J)"`. After some reflection, it became clear that this

description was complete, unambiguous, and shorter than the tag-based version. Hence, it was the basis for a full language of axioms.

The simpler language is still written inside of an XML specification, but it is just text inside of a new XML tag `simpleRule`. This allows both the new and old languages to exist side by side in the same axiom file. Figure 4.9 shows the grammar for the simple XML axiom language, and Figure 4.10 shows the same axiom as Figure 4.8, but in the simpler notation. Figure 4.9 also shows the abbreviated version, the `simpleTransform` tag. This tag represents a common case of axioms that have low complexity. Essentially, a transform is an axiom with one 'exists' expression and one 'creates' expression. The axiom then marks the former and latter expressions equivalent during the response.

The new axiom format is based on a prefix notation for expressions, where every expression operand is either another expression or a string. The strings are interpreted in a context-specific manner based on their operator parent. For instance, the expression "`int("32","1")`" in LLVM defines an integer constant where the first string specifies the bit-width of the integer type and the second string defines the integer's value. The non-domain operators in the simple language are prefixed with a '`%`' character, just to distinguish them from domain operators (i.e. "`%theta-1(...)`"). Labels are applied to expressions by prefixing them with "`@`*`labelname`*`:`", and referencing them with "`@`*`labelname`*" as an operand. Hence we can construct the cyclic expression from Figure 4.4 as "`@TOP:%theta-1(int("32","0"), add(@TOP, int("32","0")))`".

4.2 Complex Analyses

In addition to the simple axioms, our system allows arbitrarily complex equality analyses as well. These are similar to the axioms in that they still have a trigger and response phase. They differ in that the trigger may include the 4 sections as described above, but may also contain an arbitrary predicate over the matched nodes. Also, the trigger may contain "wildcard expressions", which are expressions where only the arity is used for pattern-matching, and not the oper-

```
<simpleRule name='a short description of the axiom'>
   rule := (exists | trues | falses | invariant)+ "==>"
           (creates | trues | falses | equalities)+

   exists := node
   trues := "{" node+ "}"
   falses := "!{" node+ "}!"
   invariant := "~" [0-9]+ "{" node+ "}"     // ex:  1{ ... }
   creates := node
   equalities := node "=" node

   node :=  "@" ident                      // label reference
          | ("@" ident ":")? op            // domain op [with label]
          | ("@" ident ":")? nondomain     // nondomain op [with label]
          | ("@" ident ":")? "*"           // metavariable [with label]

   op := ident "(" operandlist? ")"
   nondomain := nondomain_ident "(" nodelist? ")"

   ident := [a-zA-Z0-9_]+                  // identifier
   nondomain_ident := "%theta-" [0-9]+     // example:  %theta-1
                    | "%eval-" [0-9]+
                    | "%pass-" [0-9]+
                    | "%phi"

   nodelist := node ("," node)*
   operandlist := operand ("," operand)*

   operand := node | string

   string := // double-quoted string, allows escape chars
</simpleRule>

<simpleTransform name='...'>
   node "=" node
</simpleTransform>
```

Figure 4.9: The grammar for the simple XML axiom language.

```
<simpleRule>
   @E:extractelement(insertelement(@V:*, *, @I:*), @J:*)
   !{ eq(@I, @J) }!
   ==>
   @E = extractelement(@V, @J)
</simpleRule>
```

Figure 4.10: The same axiom as in Figure 4.8, in the simple language.

ator. Hence, a wildcard expression of arity 2 would match any binary operator. The operator of a wildcard expression can be examined separately in the arbitrary predicate portion. The response phase may make arbitrarily complicated structures, and may refer to any part of the nodes matched in the trigger phase.

As a simple example, this could allow very elaborate constant folding, such as the simplification of the Java expression `"abcd".concat("efgh")`. We could write the trigger of this analysis to match against a call to the `concat` method, where the target (call it T) and the parameter (call it P) of the call are wildcard expressions of arity 0 (constants with unknown operators). Then in the predicate portion of the trigger, we can examine the operators of T and P in order to determine whether or not they are literal strings. If they are, then the response phase can take their values and compute the concatenation, then create a new EPEG node for the result and make it equivalent to the return value of the call.

Clearly, these complex analyses are more elaborate than the simple axioms we have presented so far. As such, they can only be described in a Turing-complete language. Many of our analyses are written directly in the source code of Peggy, so both the trigger predicate and the response action can be written directly in Java. However, this is not convenient for the purposes of extensibility. To that end, I have designed an input language for defining these types of analyses. The language is once again based around XML, but it allows the user to include arbitrary Turing-complete code fragments to define the trigger predicate and the response actions.

The grammar for the analysis definition language is presented in Figure 4.11. As in the original XML-based axiom language, there is a `<trigger>` tag and a `<response>` tag. The trigger tag allows the same section tags as in the previous grammar, but it also allows the `<match>` tag. This tag defines the (optional) arbitrary trigger predicate, in the form of a single JavaScript function named “match”. This function can examine the nodes that matched in the other trigger sections with arbitrary JavaScript code. The function must return a boolean determining whether or not the predicate is satisfied. The purpose of this predicate code is to compute additional restrictions that cannot be expressed through the other trigger tags. The response tag has no subelements, but contains only JavaScript code.

```
<analysis name='...'>
   <trigger>
      <!-- Allows same sections as
           trigger tag defined above -->
      <match>
         function match() {
            /* Arbitrary JavaScript code */
         }
      </match>
   </trigger>

   <response>
      function build() {
         /* Arbitrary JavaScript code */
      }
   </response>
</analysis>
```

Figure 4.11: Grammar for the complex analysis definition language.

This code must be the definition of a single function "build". The build function is designed to perform all the response actions of the analysis, by directly creating nodes and adding equalities to the EPEG. The build function is only called if the entire trigger phase is satisfied.

The JavaScript code for the match and build functions is run in a special context that defines several useful functions for examining and manipulating the EPEG. A list of these is found in Figure 4.12. By using these functions, the JavaScript code can compute arbitrarily complex predicates over the nodes matched in the trigger, and perform arbitrarily complex alterations to the EPEG.

An example of an analysis written in this form is found in Figure 4.13. This analysis encodes the equivalence: "if $C \geq 0$, then $A \geq B \equiv (A * C) \geq (B * C)$". This cannot be done in the simple axiom language, since it can only do *exact* matches on operators. Hence, we could write this axiom for any specific value of C, but not for arbitrary non-negative C. Checking the actual value of the integer can only be done in the arbitrary trigger predicate, as shown in the JavaScript "match" function in Figure 4.13. This code fetches the node that matches id "`C`" and examines its operator to determine whether or not it is a constant integer that

is non-negative. The response code then creates an expression for $(A*C) \geq (B*C)$ and marks it equivalent to the expression for $A \geq B$.

Inlining. There is one equality analysis that Peggy can perform which is inter-procedural as opposed to intra-procedural, and that is function inlining. Inlining can be encoded as an equality analysis in the following way. The trigger of the analysis matches against any call node that calls a function that we want to inline. This must be a function for which we have access to the source code, and hence can build a PEG for it. The response will build a version of the inlinee's PEG into the EPEG. This new PEG will have all of its parameter nodes replaced by the actual parameters to the original call node, and the input heap summary node of the new PEG will be replaced by the call's input heap summary node. Once the new PEG is built, we mark the PEGs roots equivalent to the call node, and then we have essentially substituted an entire PEG in for the call. One slight complication is that the call node logically returns a tuple of (result value, result heap summary) and uses projection operators (typically ρ_{value} and ρ_σ) to get the individual elements. We can sidestep this issue by having axioms to react with the projection operators appropriately. We explain this in detail by way of an example.

Consider the example in Figures 4.14 and 4.15. Figure 4.14 shows the source code and PEG for an integer square root function. The roots of the PEG are marked with arrows. Note that the heap root is equal to the heap input parameter, since this function does not modify the heap at all. Figure 4.15(a) shows some source code that calls the `sqrt` function, and we wish to inline this call in Peggy. Figure 4.15(b) shows the PEG for the code in part (a), and part (c) shows the resulting EPEG after applying the inlining analysis.

The complete inlining is achieved with 3 axiom applications. The first effectively splices the PEG from Figure 4.14(b) into the EPEG, with the n parameter replaced by the expression for "$\texttt{a} * \texttt{a} + \texttt{b} * \texttt{b}$", and the σ parameter replaced by the input σ for the `call` node. There is also an "`inlinePair`" node, which points to the two roots of the inlined PEG, and becomes equivalent to the `call` node. This equivalence edge is marked with an ⓐ. Since the ρ_σ node is now effectively

Function	Description
`$(id)`	Returns the EPEG node that matched in the trigger with the given id.
`getTrue()`	Returns the `true` node of the EPEG.
`getFalse()`	Returns the `false` node of the EPEG.
`makeEqual(node,node)`	Adds a new equivalence between the given EPEG nodes.
`makeEqual(futureNode,node)`	Creates the future node in the EPEG, and adds an equality to the other existing EPEG node.
`futureNode(op,childsources)`	Creates a representation of a new node to create in the EPEG. The child sources are various descriptions of where the node's children come from.
`copySource(node,n)`	Returns a child source that refers to the n-th child of an existing EPEG node.
`concreteSource(node)`	Returns a child source that refers to an existing EPEG node.
`futureSource(futureNode)`	Returns a child source that refers to another soon-to-be-created EPEG node.
`futureSource(op,childsources)`	Returns a child source that refers to another soon-to-be-created EPEG node.

Figure 4.12: Several useful functions defined in the JavaScript context.

```
<analysis name='[A >= B] == [(A*C) >= (B*C)], if C>=0'>
  <trigger>
    <exists>
      <icmp type='sge' id='AgteB'>      <!-- A >= B -->
        <variable id='A'/>
        <variable id='B'/>
      </icmp>
      <binop type='mul' id='AtimesC'>  <!-- A * C -->
        <ref id='A'/>
        <wild id='C' value='1'/>
      </binop>
    </exists>
    <match>
      function match() {
        var c_op = $("C").getOp();
        if (c_op.isDomain() &&
            c_op.getDomain().isConstantValue()) {
          var constant =
            c_op.getDomain().getConstantValueSelf().getValue();
          return constant.isInteger() &&
                 !constant.getIntegerSelf().isNegative();
        }
        return false;
      }
    </match>
  </trigger>
  <response>
    function build() {
      var Asrc = copySource($("AgteB"), 0);
      var Bsrc = copySource($("AgteB"), 1);
      var Csrc = concreteSource($("C"));
      var times = $("AtimesB").getOp();
      var result = futureNode(
                     $("AgteB").getOp(),
                     futureSource(futureNode(times, Asrc, Csrc)),
                     futureSource(futureNode(times, Bsrc, Csrc)));
      makeEqual(result, AgteB);
    }
  </response>
</analysis>
```

Figure 4.13: An example of a complex analysis written in the input format.

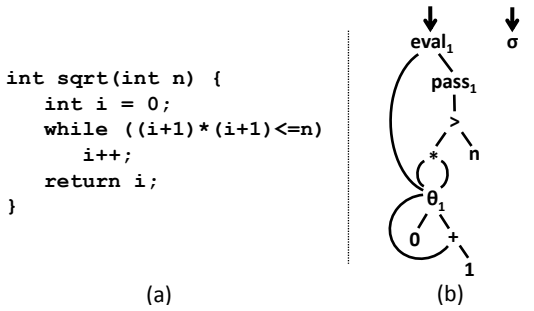


Figure 4.14: The source code (a) and PEG (b) for an integer square root function that will be inlined using Peggy.

on top of the `inlinePair` node, it can activate the second axiom which states "`rho_sigma(inlinePair(@A:*,@B:*)) = @B`". Hence the ρ_σ node becomes equivalent to the σ node, and this equivalence edge is marked with a ⓑ. Similarly, the third axiom states that "`rho_value(inlinePair(@A:*,@B:*)) = @A`", and hence the ρ_σ node becomes equivalent to the eval_1 node, with its equivalence edge marked with a ⓒ.

This separation of the inlining into 3 axioms has some advantages over trying to do it in one. First of all, it is helpful to have the second and third axioms fire independently, since one or the other might not be necessary. If the inlined function is void, then the return value will not be used, so there will be no ρ_{value} node on top of the `call` node. Similarly, if the heap summary output of the call is not used, then there would not be a ρ_σ node. In order to make the inlining into a single axiom, it would have to match against the ρ_{value} and ρ_σ in order to make them equivalent to the inlinee's roots. Hence, both would need to be present or else the axiom would not be triggered. Second, this same technique would generalize to PEGs that have a different number of roots. This could be useful when trying to use Peggy to optimize a programming language which supports more than two return values per function.

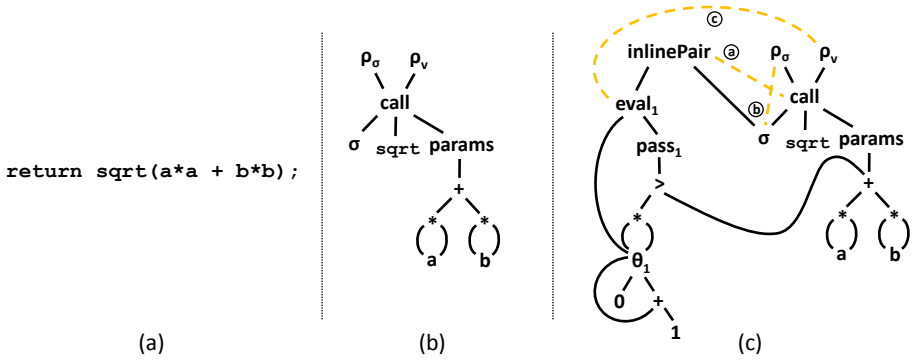


Figure 4.15: Example of inlining using Peggy. The inliner's code is in part (a), the PEG for part (a) is in part (b), and the EPEG during inlining is in part (c).

Summary

In this chapter we described what equality analyses are made of, and described a few domain-specific languages for defining them. The simplest type of analyses are the "axioms", which have a trigger and a response that can be fully described by axiom expressions over explicit operators and metavariables. An arbitrarily complicated analysis can be defined using axiom expressions, a trigger predicate over those expressions, and some arbitrary JavaScript code to describe the response action.

In the next chapter, we will use the languages we have described here to enumerate the most important axioms and analyses that are used within Peggy. These will give a more detailed idea of what kinds of axioms are useful to Equality Saturation, and how they can interact to deduce useful equalities.

Chapter 5

Axioms

In some ways, Equality Saturation is only as good as the set of axioms that is used within the engine. In the previous chapter we described what makes up an axiom, and how we go about defining them. In this chapter, we will look at some of the actual axioms themselves. This should provide a better idea of exactly how Peggy performs optimizations in a step-by-step manner. In this chapter we will describe several categories of axioms and give examples of each. In the interest of brevity, we will show few examples here, with many more presented in Appendix C.

5.1 Arithmetic Axioms

The first and possibly largest group of axioms we will discuss are the arithmetic axioms. These encode simple mathematical facts about operations such as addition, multiplication, comparison operators, etc. Versions of these axioms exist for both the Java and LLVM targets, but the examples here are taken from the Java version.

Commutativity: These axioms encode the commutativity of binary operators.

```
<simpleTransform name='A+B = B+A'>
   add(!!A:*,@B:*) = add(@B,@A)
</simpleTransform>
```

Distributivity: These axioms encode how some operators distribute over others.

```
<simpleTransform name='A*(B+C) = A*B + A*C'>
   mul(@A:*,add(@B:*,@C:*)) = add(mul(@A,@B),mul(@A,@C))
</simpleTransform>
```

Associativity: These axioms encode the associativity of binary operators.

```
<simpleTransform name='A+(B+C) = (A+B)+C'>
   add(@A:*,add(@B:*,@C:*)) = add(add(@A,@B),@C)
</simpleTransform>
```

Relational Equivalence: These axioms relate comparison operators.

```
<simpleTransform name='(A < B) == (B > A)'>
   lt(@A:*,@B:*) = gt(@B,@A)
</simpleTransform>
```

Implications: These axioms encode implications about relational operators.

```
<simpleRule name='(A > B) => (A >= B)'>
  {gt(@A:*,@B:*)}
  ==>
  {gte(@A,@B)}
</simpleRule>
```

Miscellaneous: These axioms encode other facts about mathematical operators.

```
<simpleTransform name='(X*C1)<<C2 = X*(C1<<C2)'>
   shl(mul(@X:*,@C1:*),@C2:*) = mul(@X,shl(@C1,@C2))
</simpleTransform>
```

5.2 Constant Value Axioms

The next set of axioms are those that refer to specific constant values. For example, we can encode the fact that "0+A = A". However, we must be careful with such axioms because they implicitly place type restrictions on some of the operands. For instance, what version of "0" are we using in the axiom above? In Java the 0 value could be an int, long, float, or double. In LLVM, there are 2^{24} different integer types, and there is a 0 value defined for all of them. Hence, we must write our axioms in such a way as to specify the type of the constant values either implicitly or explicitly. Throughout this section, we will use a running example of the axiom "$\forall A,\ 0 + A = A$".

Java Version. The Java version of these axioms only needs to consider four specific numeric types, so it is easiest just to make special-case axioms for each type. Since we assume that the PEG is properly typed, the context of the 0 value implies that both the A operand and the addition itself will have the same type as the 0 value (at this level, type coercions are explicit, so the addition can only accept two parameters of the same exact type).

Example:

```
<simpleTransform name='A+0=A (int)'>
   add(@A:*,int("0")) = @A
</simpleTransform>
// Anything plus 0 is itself.  Analogous rules for long/float/double.
```

LLVM Version. In LLVM there are over 2^{24} different integer types, so we cannot do special-casing as before. Instead, we can form this axiom as an analysis, and use a wildcard operator in place of the constant 0. Then in the trigger predicate, we can examine the constant value and ensure that it is a constant integer 0. In essence, we are representing 2^{24} different axioms using one analysis.

Example:

```
<analysis name='A+0 = A (integer)'>
   <trigger>
      <exists>
         <binop type='add'>
            <variable id='A'/>
            <wild value='1' id='ZERO'/>
         </binop>
      </exists>
      <match>
        function match() {
           // ensure that zero points to a constant integer 0
           var zero = $("ZERO");
           if (zero.getOp().isDomain()) {
              var domain = zero.getOp().getDomain();
              if (domain.isConstantValue()) {
                 var cv =
                    domain.getConstantValueSelf().getValue();
                 return cv.isInteger() &&
                        cv.getIntegerSelf().isZero();
              }
           }
           return false;
        }
      </match>
   </trigger>
   <response>
      function build() {
         makeEqual($("A"), $("ZERO"));
      }
   </response>
</analysis>
```

In the interest of brevity, we will not present the rest of the one-constant axioms in this form, but will instead summarize some of them in the table below.

$$A * 0 = 0, \quad A/1 = A, \quad A * 1 = A,$$
$$A - 0 = A, \quad A\hat{}0 = A, \quad A|0 = A, \quad A\&0 = 0,$$
$$(A < 0) \wedge (B < 0) \implies (A * B > 0),$$
$$(A < 0) \wedge (B < 0) \implies (A + B < 0),$$
$$(A > 0) \wedge (B > 0) \implies (A * B > 0),$$
$$(A > 0) \wedge (B > 0) \implies (A + B > 0),$$
$$(A > 0) \wedge (B < 0) \implies (A * B < 0)$$

5.3 Nondomain Axioms

There are also several axioms that are completely language-independent, in that they focus solely on the nondomain operators such as eval, θ, and ϕ. Many of these axioms are considered "built-in" and are added to the engine by default, because they are valid no matter what target language the PEG is representing.

Boolean Axioms: These are axioms that encode facts about boolean logic.

```
<simpleRule name='A && false = false'>
   @TOP:%and(@A:*, @B:*)
   !{@B}!
   ==>
   !{@TOP}!
</simpleRule>
```

Phi Axioms: These axioms encode facts about the ϕ operator.

```
<simpleRule name='phi(true,B,C) = B'>
   @TOP:%phi(@A:*,@B:*,*)
   {@A}
   ==>
   @TOP = @B
</simpleRule>
// If the condition is true, the result is the first child.
```

Loop Axioms: These axioms encode facts about loop operators.

```
<simpleRule name='eval(theta(A,B),zero) = A, if A invariant'>
   @TOP:%eval-1(%theta-1(@A:*,@B:*),%zero)
   ~1{@A}
   ==>
   @TOP = @A
</simpleRule>
// Evaluating a loop-varying value at the 0-th iteration gives the initial
    value.
```

Loop Operator Factoring Axioms: These axioms encode how domain operators can distribute through loop operators.

```
<simpleRule name='theta(S1,S2) + B = theta(S1 + eval(B,0), S2 +
    shift(B))'>
   @TOP:add(%theta-1(@S1:*,@S2:*),@B:*)
   ==>
   @TOP = %theta-1(add(@S1,%eval-1(@B,%zero)), add(@S2, %shift-1(@B))
       )
</simpleRule>
// Addition may distribute through ?theta?  as right child.
```

This axiom describes how the addition operator may distribute through a θ operator. Essentially, this amounts to applying the '+' operator to both the initial value of the loop and the inductive value of the loop. This requires modification to the other operand (B) of the + operator, since it previously may refer to a value that varies inside of the loop. That is the purpose of the inclusion of the eval and shift operators. For the base case of the θ, we must take the value of B at its own base case iteration. For the inductive case, we must evaluate B at the following iteration. If, however, B was already a loop-invariant value, then both the eval and the shift operators could be removed since they map all loop-invariant values to themselves.

Even though this axiom is valid for all B operands, it is far less general than we would like. We would like to have an axiom that allowed: 1) other operators besides '+', 2) the θ operator to be the second child as well as the first child, and 3) operators with different arities, besides just binary operators. We can do this, but it requires using an equality analysis rather than a simple axiom. An example of this is presented in Appendix C.

5.4 Language-Specific Axioms

The axioms in this section are language-specific, in that they encode properties of the language operators that are particular to the target language. For our current implementation of Peggy, this includes Java-specific axioms and LLVM-specific axioms.

5.4.1 Java-specific Axioms

These axioms are specific to the Java language. They cover facts about field access, object manipulation, how Java manipulates the program state, and more.

Field Access Axioms: These axioms encode facts about how Java accesses and modifies fields of classes.

```
<simpleTransform name='get(set(T,F,V),T,F) = V'>
   rho_value(getfield(
      rho_sigma(setfield(*,@T:*,@F:*,@V:*)),
      @T,
      @F))
   = @V
</simpleTransform>
```

This axiom states that if you set a field's value and then immediately fetch it, it will equal the value stored.

Array Access Axioms: These axioms encode facts about how Java accesses and modifies arrays and array elements.

```
<simpleTransform name='get(set(A,I,V),A,I) = V'>
   rho_value(getarray(
      rho_sigma(setarray(@SIGMA:*,@A:*,@I:*,@V:*)),
      @A,
      @I))
   = @V
</simpleTransform>
```

This axiom is analogous to the above field axiom.

5.4.2 LLVM-specific Axioms

These axioms are specific to the LLVM language, and encode the semantics of the various LLVM operators, and facts about how they interact.

Pointer Axioms: These axioms have to do with pointers to memory in LLVM, and the operations that manipulate them.

```
<simpleTransform name='store P (load P) = no-op'>
   store(@S:*,@P:*,rho_value(load(@S,@P,@N:*)),@N) = @S
</simpleTransform>
// Storing the value that was just loaded from the same pointer is a no-op.

<simpleTransform name='gep(B,0) = B'>
   getelementptr(@B:*,@T:*, indexes(int("32","0"))) = @B
</simpleTransform>
// Pointer arithmetic with offset 0 gives the original pointer.
```

The second axiom is actually a constant axiom, but it relates to the GET-ELEMENTPTR operator, which performs opaque pointer arithmetic. Essentially, it states that if your pointer arithmetic only adds 0 to the base pointer, then you are left with the original pointer value.

Vector Axioms: These axioms describe how LLVM handles vector values.

```
<simpleTransform>
   extractelement(insertelement(@V:*,@X:*,@I:*),@I) = @X
</simpleTransform>
// Extracting a vector element that was just inserted yields that new element.
```

Aliasing Axioms: In LLVM, since we have pointers, we must deal with the possibility of aliasing. We handle this by introducing some useful annotations that can help propagate information throughout the EPEG. The first of these is `stackPointer(P)`, which encodes the assertion "*P is a pointer to the current call's stack*". Based on information deduced during Equality Saturation, we may equate `stackPointer(P)` with either the true or false node of the EPEG.

The second annotation that we use is `doesNotAlias(P,Q)`, which encodes the assertion "*pointer P and pointer Q do not alias each other*". This can become equivalent to true if we have proof that P and Q can never point to

the same memory location. With these two annotations, we can perform a great deal of alias analysis in order to determine equivalences between pointer operations.

Example:

```
<simpleRule name='non-aliasing stores can swap'>
   @S:store(store(@SIGMA:*,@PTR1:*,@V1:*,@A1:*),
            @PTR2:*,
            @V2:*,
            @A2:*)
   {annotation("doesNotAlias", @PTR1, @PTR2)}
   ==>
   @S = store(store(@SIGMA,@PTR2,@V2,@A2),@PTR1,@V1,@A1)
</simpleRule>
// Adjacent stores to non-aliasing addresses can switch order.
```

Many of the axioms in this category only introduce and propagate the various annotations we created. The major use for these annotations is to trigger one particular axiom, which allows a load to move before a store to a non-aliasing pointer. This axiom combines with another simple axiom which states that a load right after a store to the same pointer yields the stored value. When these two axioms work in concert, it allows a load to traverse down the "sigma path" through any non-aliasing stores that come before it, until it can match up with a store to the same pointer. Then the load can be replaced by the stored value, which simplifies the PEG. This can significantly reduce the number of memory operations in a program, which is very useful for optimization purposes. It can also be very useful in terms of translation validation, if one of the translations performed is to remove memory operations.

5.5 Constant Folding

One of the most common peephole optimizations in any optimizer is constant folding. It amounts to identifying constant expressions in the code and evaluating them statically, then replacing the code with the result expression. For instance, one could replace "4+5*8" with "44", and now the program does not

have to do that computation at runtime. This is possible because we know the operands ahead of time, so we can be sure of the final result of the computation.

In Peggy, we also have a constant folder. It has no special status, but instead runs as yet another equality analysis within the Equality Saturation engine. However, whereas all the axioms above were described in terms of the input languages defined in Chapter 4, the constant folder is written in Peggy's source code and is not extensible.

Since the constant folder is just another equality analysis, we can describe it in terms of a trigger and a response. The trigger for the constant folder looks for a single domain operator that takes at least one parameter. It then checks to make sure that every one of the parameters is a constant value. We define a constant value to be any 0-arity node that is not an input parameter. This is only the structural part of the trigger. The trigger also has a more complicated condition, which essentially checks to see if the operator, when combined with its constant children, can be pre-evaluated and hence folded.

This condition is very complicated to express, and is satisfied by an instance of the `ConstantFolder` class in Peggy, as defined below:

```
public interface ConstantFolder<L> {
   boolean canFold(L root, List<? extends L> children);
   L fold(L root, List<? extends L> children);
}
```

Any object that implements this interface must do two things. Firstly, it must be able to decide whether or not it knows how to fold a given operator with its constant children (the `canFold` method). Secondly, if it has claimed it can fold something, it must be able to perform the folding and return the result (the `fold` method). The final part of the trigger for the constant folding analysis makes a query against an instance of `ConstantFolder`, and if its `canFold` method returns true then the `fold` method is called, and the result is used as the label for a new constant value node.

This design allows us a high degree of modularity in our choice of implementation of the constant folder. For instance, in our current system there are default constant folders that are able to fold constant based around the built-in LLVM

and Java operators and values. However, we could extend the domain of foldable operators to include annotation labels as well. Since annotation labels have no a-priori semantics attached to them, this provides a high degree of flexibility. For instance, in Java, one could make a custom constant folder that can fold calls to the most common String operations on constant Strings. If an expression such as `"abcd".indexOf('a')` was seen, we could write an axiom to equate this with `foldIndexOf("abcd", 'a')`, which uses a custom annotation label. The constant folder would be aware of this label and since the operands are both constant, could fold this expression by calling indexOf inside the folder.

5.6 Domain-Specific Axioms

One of the most important features of the axiom input languages is that they allow for easy extensibility of the Equality Saturation system. The main usage presented so far is describing the semantics of built-in language operators, and the interactions between them. These are generally useful because the semantics of the operators does not change from one program to the next. However, if we consider only the current program we are evaluating, we can make assumptions that allow us to write axioms that are less general-purpose, but still quite useful. These are *domain-specific* axioms, because they apply only to a particular program domain.

Writing axioms must be done with care, because they are required to be accurate regardless of the context they are in (simply because we do not control how and when they are applied during saturation). If, however, there is an overarching assumption about the entire program, then this same assumption can apply to the axiom set as well. For example, if there is a global function named "sqrt" that is used throughout the program, you may make some assumptions about calls to "sqrt" based on its known (and presumably fixed) semantics. One may write axioms about calls to this function to encode some of the details of how it works and interacts with other operators. In this particular case (assuming "sqrt" is a square-root function) we could write axioms like `sqrt(X*X) = abs(X)`, or even add an analysis to do folding if the argument is constant. These axioms would not

be valid outside of this program domain, because any other program might define a different function named "sqrt" with different semantics. But within the context of this program, the axioms are valid.

We have written some domain-specific axioms for the experiments we describe in Chapter 9. These are Java axioms that to optimize a raytracer benchmark for the purposes of evaluating the effectiveness of Peggy as an optimizer. These axioms are designed to encode facts about a particular class named `CVector3D`, which represents a 3-D vector of double-precision floating point numbers. The vector class uses a factory pattern: there is a static method that takes 3 doubles and returns a new `CVector3D` by calling the single private constructor. The constructor also takes 3 doubles and sets the final X, Y, and Z fields to those values, making the vector immutable once it is created. Since the vectors are immutable, every method that manipulates vectors must return a new one. Hence, there is a lot of useless temporary object creation being done. The axioms we define below help to reduce some of that by exposing the semantics of the vector operations.

Examples:

```
<simpleTransform name='cons(A,B,C).X = A'>
   rho_value(getfield(
      *,
      rho_value(invokestatic(
         *,
         method("CVector3D CVector3D.cons(double,double,double)"),
         params(@A:*,@B:*,@C:*))),
      field("double CVector3D.X")))
   =
   @A
</simpleTransform>
// The value of the X field of a newly-constructed vector is equal to the first
    input (similar axioms for Y, Z).

<simpleTransform name='cons(A,B,C).sub(cons(D,E,F)) = cons(A-D, B-E, C
    -F)'>
   invokevirtual(
      @SIGMA:*,
      rho_value(invokestatic(
         *,
         @CONS:method("CVector3D CVector3D.cons(double,double,double)")
             ,
         params(@A:*,@B:*,@C:*))),
      method("CVector3D CVector3D.sub(CVector3D)"),
      params(
         rho_value(invokestatic(
            *,
            @CONS,
            params(@D:*,@E:*,@F:*)))))
   =
   invokestatic(
      @SIGMA,
```

```
      @CONS,
      params(sub(@A,@D), sub(@B,@E), sub(@C,@F)))
</simpleTransform>
// Subtracting two newly-constructed vectors is the same as constructing a new
    vector of the difference of the components (similar method for add).

<simpleTransform name='cons(A,B,C).scaled(D) = cons(A*D,B*D,C*D)'>
   invokevirtual(
      @SIGMA:*,
      rho_value(invokestatic(
         *,
         @CONS:method("CVector3D CVector3D.cons(double,double,double)")
             ,
         params(@A:*,@B:*,@C:*))),
      method("CVector3D CVector3D.scaled(double)"),
      params(@D:*))
   =
   invokestatic(
      @SIGMA,
      @CONS,
      params(mul(@A,@D), mul(@B,@D), mul(@C,@D)))
</simpleTransform>
// Scaling a newly-constructed vector by a scalar is the same as making a new vector
    with scaled components.
```

These axioms allow complicated vector expressions to be simplified down to just the basic arithmetic that is involved. This can avoid creating temporary vector objects for each subexpression, which saves both memory and time.

Summary

In this chapter, we presented examples of the actual axioms that we use when performing Equality Saturation. Some of them are simple and just encode facts about arithmetic operators. Most of them are language-specific in that they refer to operators that only appear in onr particular target language. Other are language-independent and only refer to the nondomain operators. The ones that name specific constants often have to be phrased as analyses, since often the same axiom would apply to constants of many different types. Finally, we talked about the constant folder, which looks for constant expressions and attempts to statically evaluate them so that they do not need to be evaluated at runtime.

In the next chapter, we will discuss an issue the crops up in many different places throughout the design of Peggy. That is the issue of linear types and linear operators in the PEG. Though they can often be treated just like any other value,

in certain situations they can greater increase the complexity of the algorithms involved.

Chapter 6

Side Effects and Linearity

The PEG is a purely functional representation of a program. As such, it suffers from the same major problem that all functional representations suffer from, which is that it becomes difficult to represent operations that produce side effects. While languages like Haskell solve this problem with monads [JL95], our chosen approach is slightly different.

We represent side effects by encapsulating them all within an opaque *effect token*. The effect token is a value that represents the state of the stateful portions of the computing environment, such as the hard drive, I/O streams, and even main memory [TSTL10]. With this abstraction, we can represent side-effecting behavior with functional PEG operators. Any operator that wants to produce side effects must take an effect token as input and can then produce a new one as output. The new effect token opaquely represents the new state of the environment, after the side effects that occurred during the operation.

In the Peggy system, we represent the effect token by σ. Every PEG program that has side effects must take an effect token as input. This allows us to keep track of what side effects the function had as a whole. The input effect token is threaded through the side-effect-producing operators in the PEG, and finally produced as one of the root return values. This is a complete way to represent side effects, and it is still done in a purely functional way.

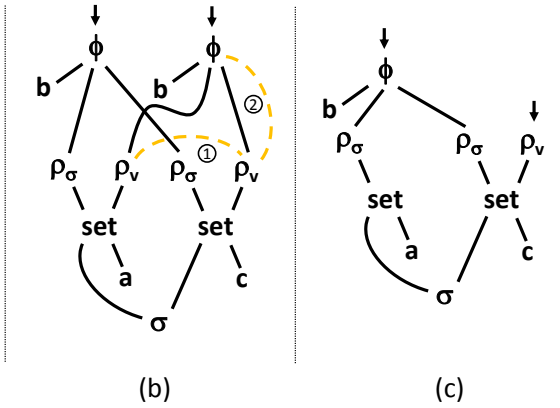
6.1 The problem with effect tokens

While the system we have just described works correctly for PEGs, there is an important detail that we must consider. Since PEGs are functional, everything is treated as a value and every value is referentially transparent. This means that an operator's context does not affect its value; only its parameters do. Hence, there is no reason why we cannot have many different unrelated effect tokens flowing throughout a single PEG. Though the tokens are designed to intuitively represent the continuous state of the program as it changes, there is no mathematical need for this. Tokens could be copied, filtered, and destroyed as long as the semantics of the operators is unchanged.

This freedom becomes a problem when we try to convert a PEG back to an imperative program. As one of the final phases of the optimization pipeline, we are given a saturated EPEG and we must search through it to find an optimal PEG that is equivalent to the original input PEG. We call this the *PEG Selection Problem*, and we describe it in detail in Chapter 8. We then convert this PEG to an imperative CFG, and write the optimized code back to disk. The notion of an effect token makes explicit something that was only implicit in an imperative language. The state of the program environment is not a first-class object in most imperative languages. In cannot be manipulated directly or passed to functions. Moreover, there is no way to create a new one, nor to destroy one. There is a single global instance of the state of the program, and it is updated implicitly by the side-effect-producing operations of the imperative language. Hence, if we wish to convert a PEG into an imperative program, we have a very harsh restriction on the way that effect tokens can be used within the PEG.

One example of how things can go wrong has to do with conditional statements. We represent conditional statements in a PEG using ϕ nodes, and there will be one ϕ node for every program variable that may have two different values based on the condition. Through Equality Saturation, we can get into situations where an effectful operation is both inside and outside of a conditional statement, and hence it is unclear how to revert the code to CFG. Consider the example program shown in Figure 6.1(a). The **foo** method makes calls to the **set** method, which sets

```
foo(a,b,c) {
  if (b)
    return this.set(a);
  else
    return this.set(c);
}
set(v) {
  this.v = v;
  return this.counter++;
}
```



(a) (b) (c)

Figure 6.1: An example of how effect tokens can interact poorly with ϕ nodes. Part (a) shows the original source code, part (b) shows the EPEG for the code during saturation, with 2 axioms applied, and part (c) shows a potential PEG that could be chosen for reversion from the EPEG.

a field value to its parameter and then increments and returns a counter value. The counter essentially keeps track of how many times **set** has been called. The PEG for this code is shown in part (b). The designer of this code could write a domain-specific axiom about the **set** method, knowing its semantics. Since **set** always returns the latest counter value, we know that any two calls to **set** will produce the same return value if they have the same incoming memory state. Hence, we can deduce the equality illustrated by dotted-edge ①. Now we can apply the rule "$\phi(A, B, B) = B$" to establish dotted-edge ②. Starting from this EPEG, when we apply the normal PEG Selection process, one possible solution is the PEG shown in Figure 6.1(c). This PEG has discarded the right-hand ϕ node in favor of its right child, which is now the program's global return value. Since it is the return value, it must be evaluated unconditionally and its value must be returned. Unfortunately, the effect token output of the **set** method is only executed conditionally based on the value of b. So, the result value output of the right-hand call to **set** is used unconditionally but the effect token output is only used conditionally. This becomes a problem when attempting to revert this PEG to a CFG, because the call to **set** cannot be "split up" in the way we need here.

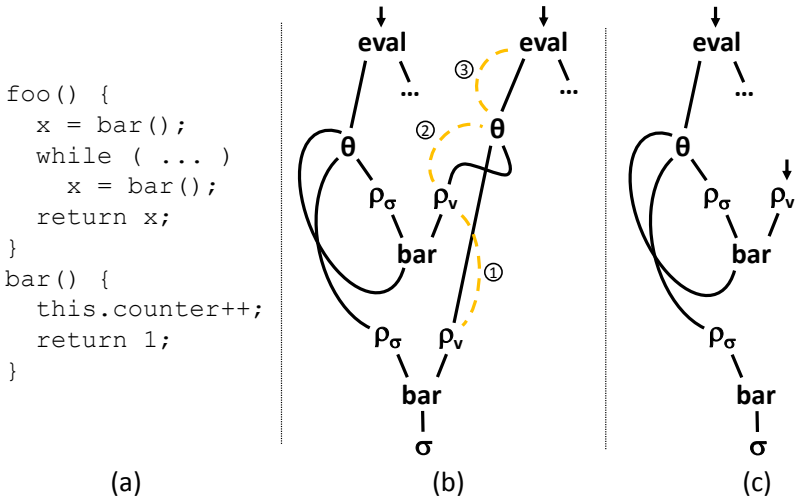


Figure 6.2: An example of how effect tokens can interact poorly with θ nodes. Part (a) shows the original source code, part (b) shows the EPEG for the code during saturation, with 2 axioms applied, and part (c) shows a potential PEG that could be chosen for reversion from the EPEG.

A similar problem is caused by loops. Just as code can be both inside and outside of a conditional statement, we can generate a case when code is both inside and outside of a loop. Consider the source code in Figure 6.2(a). The **foo** method makes calls to the **bar** method, which increments a counter and always returns 1. The PEG for **foo** is in Figure 6.2(b). Since we know the return value of **bar** is always the same, we can equate the results of the two calls, and get dotted-edge ①. Then, by semantics of the θ operator, and since both children of the θ node are equal, the θ node itself is equal to that value, so we get dotted-edge ②. Finally, by the semantics of eval, eval-ing a loop-invariant quantity is equal to the quantity itself, so we get dotted-edge ③. The PEG in Figure 6.2(c) is one that could result from the normal PEG Selection process, given the EPEG in part (b). Here we have removed the θ and eval nodes, and marked the value of the **bar** method inside the loop as the global return value. This means that this node must be evaluated unconditionally, and its value returned. Unfortunately, the effect-token output of that method call is only evaluated inside of the loop. Hence we once again have a

situation where we cannot split up the effectful operation in the way we need in order to produce imperative code.

6.1.1 A Solution: Linear Types

One way to model the restrictions on effect tokens is to consider the effect token as a value having *linear type*. A linear type describes a set of objects such that there must always be exactly one in existence at any given time [Wad90b, Gir98]. They cannot be copied or destroyed, and any operator that takes one as input must produce one as output. There are a few special cases to consider for this rule, specifically branches and loops. For a conditional branch (or even an n-way branching operator like a "switch") all arms of the branch get the same incoming linear value, but since only one will actually execute at runtime, this does not constitute an actual duplication of the value, and a single value is produced as output. For loops, there is a single linear typed value existing at all times throughout the loop. These rules combine to keep exactly one instance of the linear typed value in existence at all times. This kind of behavior is exactly what we want for our effect tokens, when converting back to imperative code.

Linear types provide us with a model for how we should try to arrange our PEGs in order to easily revert them back to imperative code. The problem remains: how do we get our PEG into that form? To answer this problem, it helps to first look at which PEGs we would want to revert in the first place. Specifically, we can ask ourselves at which state in the pipeline do we generate PEGs that we might want to convert to imperative code. For the Peggy system, this only occurs when we use Equality Saturation to perform optimization.

After Equality Saturation, we perform the PEG Selection process described above, and attempt to revert the resulting PEG to imperative code. Hence, the EPEG-to-PEG conversion must produce PEGs that have a linear usage of their effect tokens. There are essentially two possible solutions to this problem, and we discuss them below.

6.1.2 Solution 1: PEG to linear PEG conversion

One solution to this problem would be to have a separate conversion pass that acted on the output of the PEG Selection problem. The PEG that is produced by the PEG Selection process would potentially not have a properly linear usage of its effect tokens. The conversion pass would take this PEG and try to convert it to a new one that differed only in its use of the effect tokens, which would be properly linearized. Essentially, this would amount to an algorithm that could take any PEG as input and return a linearized PEG as output.

In the Peggy system, this was the first solution we actually implemented. The main reason for this is that we did not immediately understand the need for a properly linearized PEG, and so we implemented this pass after the fact. What we discovered is that in general this technique does not work very well. There are essentially two versions of the conversion that one can write. The first is one that will always succeed and produce a properly linearized PEG, but in the worst case will require a great deal of code copying, which bloats the final PEG. The second version is one that does a greedy approach to linearizing the PEG, and makes as few changes as possible to the PEG, but in some cases may not succeed. Since the latter version does not always succeed in producing a linearized PEG, there is a certain percentage of PEGs that we simply cannot convert to imperative code.

In the end, we tried and abandoned both of these approaches. The large amount of code copying from the first version negated most of the positive effects of the optimization that we were attempting in the first place. The failure rate of the second version was high enough that most large, complicated PEGs were not convertible, which prevented any optimization from occurring at all. We concluded that the correct solution lay in a fundamentally different approach to the problem, as described below.

6.1.3 Solution 2: Stateful PEG Selection Problem

The previous approach involved two phases. The first was the PEG Selection process and the second took the result PEG from that process and converted it to a properly linearized PEG. As an alternative, we can combine these two phases

into one. This amounts to reformulating the PEG Selection Problem into a new problem that produces the optimal PEG from the EPEG, subject to the constraint that the result must have linear usage of effect tokens. We call this new problem the *Stateful PEG Selection Problem*, and it is described in detail in Chapter 8. This new approach essentially adds some new restrictions to the original PEG Selection problem, so that the only valid results now must be linearized. Hence every output of the Stateful PEG Selection process is now suitable for conversion back to imperative code.

Summary

In this chapter we introduced the concept of effect tokens, which are a functional representation for operations that produce side effects from imperative code. We can use these effect tokens within a PEG just like any other values, but when converting an effectful PEG back to imperative code, we must be careful about how we handle the effectful operations. Imperative code does not have first-class effect tokens, and implicitly treats the state of the program as a value with linear type: there must be exactly one at all times, and it cannot be copied or destroyed. We must restructure the PEG to treat its effect tokens linearly as well; only then can we convert the PEG back to imperative code correctly. We outlined two approaches to do this, favoring the second which reformulates the PEG Selection problem to incorporate additional constraints that force the resulting PEG to be properly linearized. The details of the actual PEG Selection problem and the subsequent Stateful PEG Selection problem are discussed in detail in Chapter 8.

This chapter focused on a particular aspect of the optimization pipeline. The next chapter describes the entire optimization pipeline algorithm, and how we can use Equality Saturation to perform optimization of programs.

Chapter 7

Optimization

In general, Equality Saturation is used to explore a large space of equivalent programs. The ultimate goal behind this exploration is application-specific, and can be designed independently of the saturation engine itself. Hence, in our Peggy implementation, we have multiple clients that make use of the saturation engine internally. The first client we designed was our intra-procedural optimizer.

Abstractly, program optimization can be thought of as a search problem. Specifically, when we optimize a program P, we are actually just searching for the most efficient program that is semantically equivalent to P. Equality Saturation is well-suited to this task, since it allows us to explore a large portion of the space of programs that are equivalent to the input P. Hence, the saturated EPEG potentially contains a large number of optimized versions of P, and it is our job to choose one. The task of choosing the most-optimized program from the EPEG is the known as the PEG Selection Problem, which we describe in detail in Chapter 8.

Once we have saturated the EPEG and solved the PEG Selection Problem, we are left with a PEG that is equivalent to the original program P, but is optimal over the set of PEGs represented by the EPEG. All that remains is to convert this PEG back to its original program representation. Hence, we can use our Equality Saturation engine as the heart of an effective intra-procedural optimizer.

In the following sections, we present the benefits of our approach as well as some of the drawbacks. Chapter 9 follows with an experimental evaluation of Equality Saturation as a means of optimization.

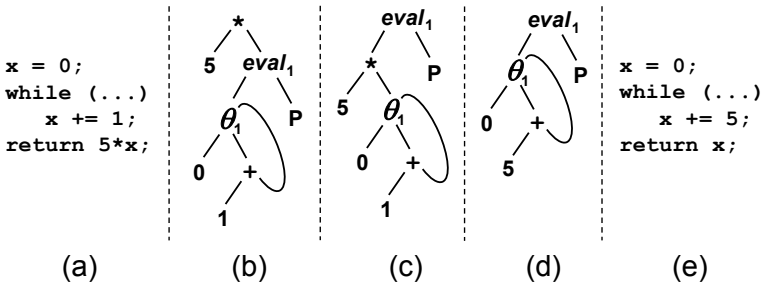


Figure 7.1: An example of loop-based code motion from simple axiom applications; (a) the original source code, (b) the original PEG, (c) the PEG after distributing $*$ through eval_1, (d) the PEG after performing loop-induction-variable strength reduction, (e) the resulting source code.

7.1 Local Changes Have Non-Local Effects

The axioms we apply during our saturation phase tend to be simple and local in nature. It is therefore natural to ask how such axioms can perform anything more than peephole optimizations. In this section, we show additional examples of how Peggy is capable of making significant changes in the program using its purely local reasoning. We particularly emphasize how local changes in the PEG representation can lead to large changes in the CFG of the program. We conclude the section by describing some loop optimizations that we have not fully explored using PEGs, and which could pose additional challenges.

7.1.1 Loop-based code motion

We start with an example showing how Peggy can use simple local axioms to achieve code motion through a loop. Consider the program in Figure 7.1. Part (a) shows the source code for a loop where the counter variable is multiplied by 5 at the end, and part (e) shows equivalent code where the multiplication is removed and the increment has been changed to 5. Essentially, this optimization moves the $(*5)$ from the end of the loop and applies it to the increment and the initial value instead. This constitutes code motion into a loop, and is a non-local transformation in the CFG.

Peggy can perform this optimization using local axiom applications, without requiring any additional non-local reasoning. Figure 7.1(b) shows the PEG for the expression `5*x` in the code from part (a). Parts (c) and (d) show the relevant pieces of the EPEG used to optimize this program. The PEG in part (c) is the result of distributing multiplication through the eval node. The PEG in part (d) is the result of applying loop-induction-variable strength reduction to part (c) (the intermediate steps are omitted for brevity). Finally, the code in part (e) is equivalent to the PEG in part (d).

Our mathematical representation of loops is what makes this optimization so simple. Essentially, when an operator distributes through eval (a local transformation in the PEG), it enters the loop (leading to code motion). Once inside the loop, distributing it through θ makes it apply separately to the initial value and the inductive value. Then, if there are axioms to simplify those two expressions, an optimization may result. This is exactly what happened to the multiply node in the example. In this case, only a simple operation ($*5$) was moved into the loop, but the same set of axioms would allow more complex operations to do the same, using the same local reasoning.

7.1.2 Restructuring the CFG

In addition to allowing non-local optimizations, small changes in the PEG can cause large changes in the program's CFG. Consider the program in Figure 7.2. Parts (a) and (f) show two CFGs that are equivalent but have very different structure. Peggy can use several local axiom applications to achieve this same restructuring. Figure 7.2(b) shows the PEG version of the original CFG, and parts (c)-(e) show the relevant portions of the EPEG used to optimize it. Part (c) results from distributing the multiply operator through the left-hand ϕ node. Similarly, part (d) results from distributing each of the two multiply operators through the bottom ϕ node. Part (e) is simply the result of constant folding, and is equivalent to the CFG in part (f).

By simply using the local reasoning of distributing multiplications through ϕ nodes, we have radically altered the branching structure of the corresponding

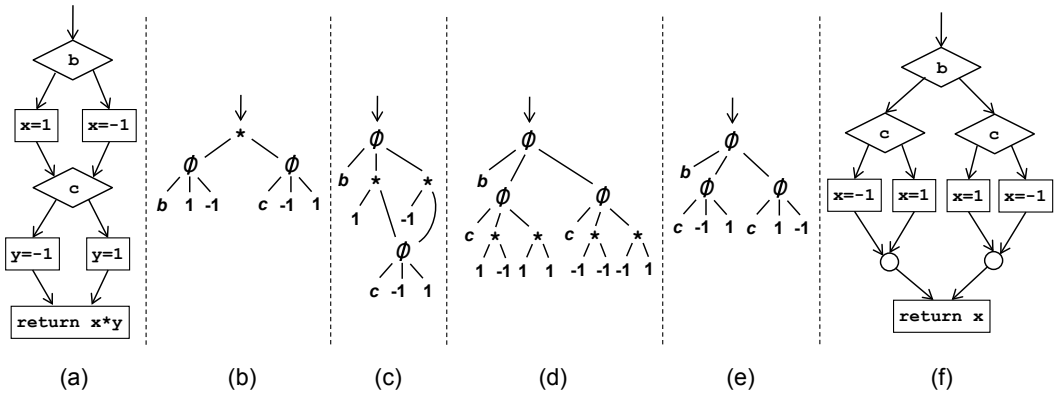


Figure 7.2: An example of how local changes in the PEG can cause large changes in the CFG: (a) the original CFG, (b) the original PEG, (c) the PEG after distributing $*$ through the left-hand ϕ, (d) the PEG after distributing $*$ through the bottom ϕ, (e) the PEG after constant folding, (f) the resulting CFG.

CFG. This illustrates how small, local changes to the PEG representation can have large, far-reaching effects on the program.

7.1.3 Loop Peeling

Here we present an in-depth example to show how loop peeling is achieved using equality saturation. Loop peeling essentially takes the first iteration from a loop and places it before the loop. Using very simple, general-purpose axioms, we can peel a loop of any type and produce code that only executes the peeled loop when the original would have iterated at least once. Furthermore, the peeled loop will also be a candidate for additional peeling.

Consider the source code in Figure 7.3(a). We want to perform a loop peeling on this code, which will result in the code shown in Figure 7.3(i). This can be done through axiom application through the following steps, depicted in Figure 7.3 parts (c) through (h).

Starting from the PEG for the original code, shown in part (b), the first step transforms the pass_1 node using the axiom $\mathsf{pass}_1(C) = \phi(\ \mathsf{eval}_1(C, \mathrm{Z}),\ \mathrm{Z},\ \mathrm{S}(\ \mathsf{pass}_1(\ \mathsf{peel}_1(C)\ )))$ yielding the PEG in part (c). In this axiom, Z is the zero iteration count value, S is a function that takes an iteration count and returns its successor

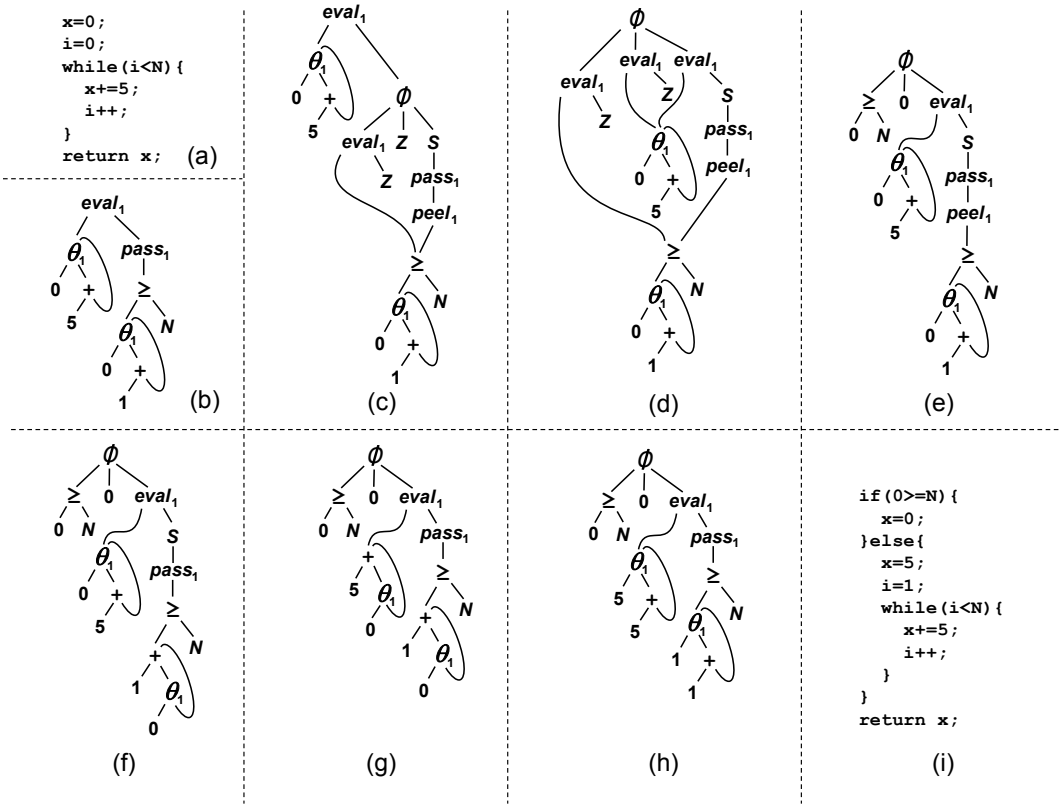


Figure 7.3: An example of axiom-based loop peeling: (a) the original loop, (b) the PEG for part (a), (c)-(h) intermediate steps of the optimization, (i) the final peeled loop, which is equivalent to (h).

(i.e. $S = \lambda x.x + 1$), and peel takes a sequence and strips off the first element (i.e. $\mathsf{peel}(C)[i] = C[i+1]$). This axiom is essentially saying that the iteration where a loop stops is equal to one plus where it would stop if you peeled off the first iteration, but only if the loop was going to run at least one iteration.

The second step, depicted in part (d), involves distributing the topmost eval_1 through the ϕ node using the axiom $op(\phi(A, B, C), D) = \phi(A,\ op(B, D),\ op(C, D))$. Note that op only distributes on the second and third children of the ϕ node, because the first child is the condition.

The third step, shown in part (e), is the result of propagating the two $\mathsf{eval}_1(\cdot, \mathrm{Z})$ expressions downward, using the axiom $\mathsf{eval}_1(op(a_1, \ldots, a_k), \mathrm{Z}) = op($ $\mathsf{eval}_1(a_1, \mathrm{Z}),\ \ldots,\ \mathsf{eval}_1(a_k, \mathrm{Z}))$ when op is a domain operator, such as $+, *$, or S.

When the eval meets a θ, it simplifies using the axiom $\mathsf{eval}_1(\theta_1(A, B), \mathrm{Z}) = A$. Furthermore, we also use the axiom $\mathsf{eval}_1(C, \mathrm{Z}) = C$ for any constant or parameter C, which is why $\mathsf{eval}_1(\mathtt{N}, \mathrm{Z}) = \mathtt{N}$.

The fourth step, shown in part (f), involves propagating the peel_1 operator downward, using the axiom $\mathsf{peel}_1(op(a_1, \ldots, a_k)) = op(\mathsf{peel}_1(a_1), \ldots, \mathsf{peel}_1(a_k))$ when *op* is a domain operator. When the peel operator meets a θ, it simplifies with the axiom $\mathsf{peel}_1(\theta_1(A, B)) = B$. Furthermore, we also use the axiom that $\mathsf{peel}_1(C) = C$ for any constant or parameter C, which is why $\mathsf{peel}_1(\mathtt{N}) = \mathtt{N}$.

The fifth step, shown in part (g), involves removing the S node using the axiom $\mathsf{eval}_1(\theta_1(A, B), \mathrm{S}(C)) = \mathsf{eval}_1(B, C)$.

The final step (which is not strictly necessary, as the peeling is complete at this point) involves distributing the two plus operators through their θ's and doing constant folding afterward, to yield the PEG in part (h). This PEG is equivalent to the final peeled source code in part (i).

It is interesting to see that this version of loop peeling includes the conditional test to make sure that the original loop would iterate at least once, before executing the peeled loop. Another way to implement loop peeling is to exclude this test, opting only to peel when the analysis can determine statically that the loop will always have at least one iteration. This limits peeling to certain types of loops, those with guards that fit a certain pattern. This can both increase the analysis complexity and reduce the applicability of the optimization. In the PEG-based loop peeling, not only do we use the more applicable version of peeling, but the loop guard expression is immaterial to the optimization.

The resulting PEG shown in Figure 7.3(h) is automatically a candidate for another peeling, since the original axiom on pass can apply again. Since we separate our profitability heuristic from the saturation engine, Peggy may attempt any number of peelings. After saturation has completed, the global profitability heuristic will determine which version of the PEG is best, and hence what degree of peeling yields the best result.

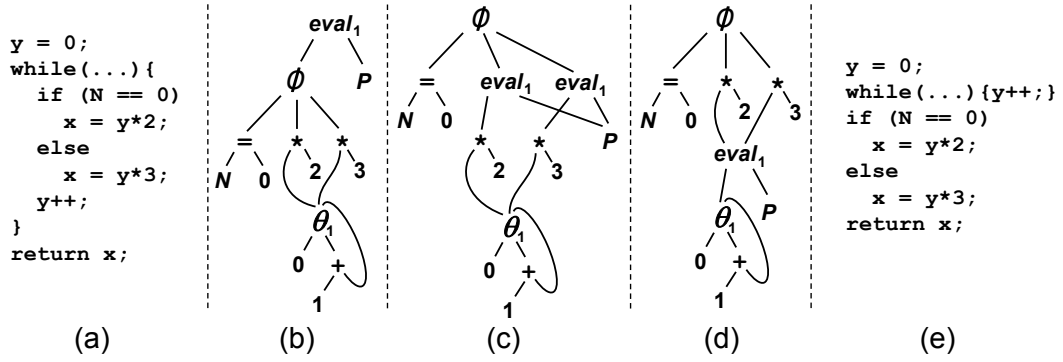


Figure 7.4: An example of branch hoisting: (a) the original program, (b) the PEG for part (a), (c) the PEG after distributing eval through ϕ, (d) the PEG after distributing eval through $*$, (e) the code resulting from (d).

7.1.4 Branch Hoisting

We now examine an example of branch hoisting, where a conditional branch is moved from inside the loop to after the loop. This is possible when the condition of the branch is loop-invariant, and hence is not affected by the loop it's in. This is another example of code motion, and is an optimization because the evaluation of the branch no longer happens multiple times inside the loop, but only once at the end.

Consider the code in Figure 7.4(a). We assume that `N` is a parameter or a variable initialized elsewhere, and is clearly not altered inside the loop. Hence the condition on the if-statement is loop-invariant. Also we see that `x` is never read inside the loop, so the value it holds at the end of the loop can be expressed entirely in terms of the final values of the other variables (i.e. `y`). Hence, this code is equivalent to the code seen in part (e), where the branch is moved outside the loop and `x` is assigned once, using only the final value of `y`.

Our saturation engine can perform this optimization using simple axioms, starting with the PEG shown in part (b) corresponding to the code in part (a). In part (b), we display the pass condition as P, since we never need to reason about it. Parts (c) and (d) depict the relevant intermediate steps in the optimization. Part (c) results from distributing the eval operator through the ϕ operator using

the axiom $op(\phi(A, B, C), D) = \phi(A, op(B, D), op(C, D))$ with $op = \mathsf{eval}_1$. Part (d) comes from distributing the two eval nodes through the multiplication operator, using the axiom $\mathsf{eval}_1(op(A, B), P) = op(\mathsf{eval}_1(A, P), \mathsf{eval}_1(B, P))$ where op is any domain operator. Part (e) is the final code, which is equivalent to the PEG in part (d).

Our semantics for ϕ nodes allows the eval to distribute through them, and hence the loop moves inside the conditional in one axiom. Since we can further factor the $*$'s out of the eval's, all of the loop-based operations are joined at the "bottom" of the PEG, which essentially means that they are at the beginning of the program. Here we again see how a few simple axioms can work together to perform a quite complex optimization that involves radical restructuring of the program.

7.1.5 Limitations of PEGs

The above examples show how local changes to a PEG lead to non-local changes to the CFG. There are however certain kinds of more advanced loop optimizations that we have not yet fully explored. Although we believe that these optimizations could be handled with equality saturation, we have not worked out the full details, and there could be additional challenges in making these optimizations work in practice. One such optimization would be to fuse loops from different nesting levels into a single loop. One option for doing this kind of optimization is to add built-in axioms for fusing these kinds of loops together into one. Another optimization that we have not fully explored is loop unrolling. By adding a few additional higher-level operators to our PEGs, we were able to perform loop unrolling on paper using just equational reasoning. Furthermore, using similar higher-level operators, we believe that we could also perform loop interchange (which changes a loop `for i in R1, for j in R2` into `for j in R2 for i in R1`). However, both of these optimizations do require adding new operators to the PEG, which would require carefully formalizing their semantics and axioms that govern them. Finally, these more sophisticated loop optimizations would also require a more sophisticated cost model. In particular, because our current cost model does not take

into account loop bounds (only loop depth), it has only a coarse approximation of the number of times a loop executes. As a result, it would assign the same cost to the loop before and after interchange, and it would assign a higher cost to an unrolled loop than the original. For our cost model to see these optimizations as profitable, we would have to update it with more precise information about loop bounds, and a more precise modeling of various architectural effects like caching and scheduling. We leave all of these explorations to future work.

7.2 Axiom Sets

One of the largest factors that influences the effectiveness of Peggy is the axiom set it uses. Peggy relies on the axioms to deduce equivalences between the nodes in the EPEG; in fact, this is the only means by which Equality Saturation can do any useful computation. Hence, if any important axioms are missing then there are important deductions that cannot be made. Therefore, the more axioms that are enabled, the more kinds of equalities may be used for optimization.

Conversely, an overly large axiom set can cause problems. For instance, loops within the EPEG can combine with certain axioms to produce a situation where saturation is impossible. Specifically, it may be possible never to reach a state where no axioms can fire without doing redundant work. Figure 7.5 shows a very simple example of how this non-saturation can occur. In part (a), we see a PEG with a θ node in it, which represents a loop-inductive value, call it T. The initial expression sets $T \leftarrow A$, and the recursive expression sets $T \leftarrow op(T, B)$. If B is a constant and op is a domain operator, then we can apply an axiom to distribute the $op(\cdot, B)$ expression through the θ node. This axiom can be written as: "$op(\theta(\mathbf{A}, \mathbf{B}), \mathbf{C}) = \theta(op(\mathbf{A}, \mathbf{C}), op(\mathbf{B}, \mathbf{C}))$, if $\mathbf{C}$ is loop-invariant." The resulting EPEG after applying this axiom is shown in Figure 7.5(b). The axiom creates a new θ node, with an op node as its child. Since the left-hand child of the new op node is equivalent to the new θ node, the same axiom can apply again, this time on the new op node. The axiom does not do redundant work, because the left-hand child of each new θ node is different than the previous one's, and hence the

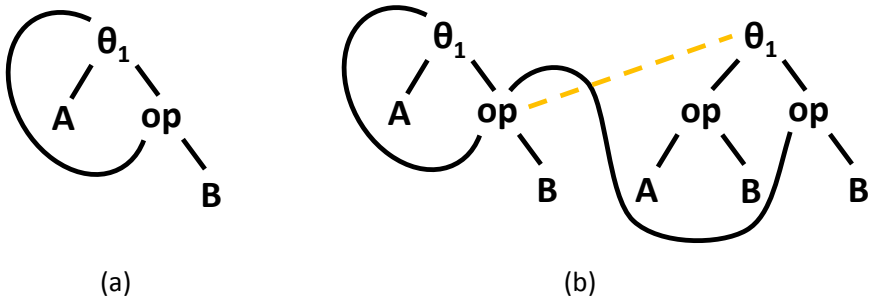


Figure 7.5: Part of an EPEG before and after applying an axiom. The same axiom can apply an infinite number of times, since it creates new nodes that will trigger itself.

axiom is creating unique expressions at each application. Thus, there is nothing to stop this axiom from being applied an unbounded number of times, requiring unbounded execution time and unbounded memory cost. It is clear that in some cases, even a single axiom can cause non-saturation to occur.

Given the previous two paragraphs, one is left with a complicated decision about how many and which axioms should be included in the engine's axiom set. Too few and the engine cannot deduce enough information to perform a useful optimization, too many and we can get unbounded expansion. In our Peggy system, we have implemented a number of practical techniques to mitigate these issues.

First and foremost, we can specify a limit on the maximum number of axioms that the engine will apply. After the specified number of axioms have been applied, the engine halts and saturation does not occur. This is an easy way to prevent the unbounded expansion described above. However, it can prevent useful axioms from applying if they would normally apply late in the saturation process.

In some cases, we see that the same axioms that can cause unbounded expansion are also required for the desired optimization to occur. In this case, disabling the axiom completely is not an option. For these axioms, we can specify a maximum number of times that particular axiom may be applied. This is especially useful for axioms that may potentially be applied an unbounded number of times, like in Figure 7.5 above. Limiting individual axioms can prevent unbounded

expansion while still enabling the other axioms to apply and do useful work.

Finally, we can attempt to weed out axioms that may apply in the EPEG, but never do useful work. For each successful optimization, we can extract a proof of the equivalence of the original program and the optimal one [TSL10]. This proof spells out the exact set of axioms that were used to deduce the equivalence between the programs, and where each axiom applied. Hence, all the axioms in that set were useful to the optimization, and the rest were not. Through examining several proofs, we can begin to see which axioms are used regularly in successful optimizations. The remaining ones are clearly not useful and can be excluded from the saturation process. This technique would be most useful for code that has not changed but is being run through the optimizer again, such as during a nightly automated build process.

These techniques merely ameliorate the problem, they do not solve it. In general, the undirected nature of the saturation process makes it very difficult to determine in advance which axioms should be applied and when. This is an area for future work.

Summary

We have seen how Equality Saturation is used to perform program optimizations by means of first saturating an EPEG and then choosing the best PEG to produce as output. This is effective because of the fact that local changes can produce non-local effects, such as a sequence of simple axioms resulting in loop-based code motion. There are also some problems with using Equality Saturation, such as the fact that several loop-based optimizations are difficult to express without introducing new higher-level operators to the PEG. Also, the cost model is imprecise in certain ways that would prevent some types of optimizations like loop peeling from ever being chosen. Finally, we see that the set of axioms chosen plays a large role in the overall effectiveness of the optimizer, and that the choice of which axioms to include is a difficult one.

In the next chapter, we will focus on an important sub-problem faced when

using Peggy as an optimizer. Specifically, we will discuss the problem of choosing the best PEG from a saturated EPEG. We call this the PEG Selection Problem, and we will see how complicated this problem is, and the various approaches we take to solving it.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER grant CCF-0644306.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Logical Methods in Computer Science 2010*. The dissertation author was the secondary investigator and author of this paper.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. The dissertation author was the secondary investigator and author of this paper.

Chapter 8

The PEG Selection Problem

One of the fundamental problems associated with optimization is converting between the different representations of the program. In the Peggy system, our primary representations are the PEG and EPEG. In order to perform equality saturation on a program, we convert the program from a CFG representation into a PEG, and then use those PEG nodes to build the initial EPEG. We then perform equality saturation on the EPEG to add new nodes and equivalences, and then we choose a single optimal PEG out of the EPEG. Finally, we convert this optimal PEG back into a CFG. Hence, there are 4 conversion problems we must solve: CFG to PEG, PEG to EPEG, EPEG to PEG, and PEG to CFG. Note that the last two of the four are only necessary during optimization. When using Equality Saturation for translation validation, we do not need to find an optimal PEG, nor convert any PEG back to CFG. Hence, the discussion in the rest of this chapter applies only when using Equality Saturation for optimization purposes. We will focus our attention on the middle two conversion phases: PEG to EPEG, and EPEG to PEG.

In the Peggy system, a PEG is represented as a labelled, ordered, directed graph $G = (N, E, \lambda, R)$, where N is the set of nodes, and $E : N \to N^*$ is a function mapping each node to its ordered list of child nodes [TSTL09]. Each node has a label, which is the mathematical function being represented by the node. They are taken from a set of functions F, and hence $\lambda : N \to F$ is the function that maps each node to its label. The children of each node represent the parameters

to the operator named by the node's label. Hence, if a given PEG node p has label $\lambda(p) = \ell$, and the label operation requires n parameters, then $E(p)$ will be an ordered list of n children. Conversely, a given PEG node may have any number of parents, which allows for a sharing of common subexpressions. In addition, a few nodes in each PEG will be marked as "return" nodes, and represent the expression for one of the return values of the function; R is the set of these nodes.

Though we abstractly define an EPEG as a collection of PEGs, the internal representations are quite different. Indeed, since equality saturation relies on the ability to have a single EPEG represent a large number of PEGs, the memory requirements of such a collection put restrictions on how we represent it.

An EPEG must represent the PEG nodes but also keep track of the equivalences between the nodes. We achieve this by defining the nodes of an EPEG differently than those for a PEG. An EPEG is a 5-tuple $G = (N, C, E, \lambda, R)$, where N is the set of nodes. The EPEG nodes still have a label defined by $\lambda : N \to F$, and an ordered list of children defined by E, but the children are sets of EPEG nodes rather than individual nodes. The nodes of an EPEG are divided into equivalence classes in C. Two nodes share a class only if they have been proven equivalent through equality saturation. Hence, every child of an EPEG node will be one of these equivalence classes, and $E : N \to C^*$ is the mapping from nodes to their ordered lists of child classes. Finally, whereas the PEG has particular nodes marked as "return" nodes, the EPEG will have certain equivalence classes marked as "return" classes, so $R \subseteq C$.

One of the immediate advantages of this representation is that we represent many equivalent PEGs implicitly through the equivalence classes. For instance, given an EPEG node $p \in N$ where $E(p) = (C_1, C_2, C_3)$ for some classes $C_1, C_2, C_3 \in C$, we can see that the node p represents $|C_1| \cdot |C_2| \cdot |C_3|$ different parent/child combinations. Hence, this representation of an EPEG can succinctly encode an exponential number of equivalent PEGs.

With these definitions, converting a PEG to an EPEG is quite straightforward. Given a PEG $G = (N, E, \lambda, R)$, we can create an EPEG $G_e = (N_e, C_e, E_e, \lambda_e, R_e)$ as follows. For each $n \in N$, we create $e_n \in N_e$ with the same label. Hence

$\lambda(n) = \lambda_e(e_n)$. Each node gets placed inside a singleton equivalence class, so we have $C_e = \{\{e_n\} \ : \ e_n \in N_e\}$. We also make the return classes of the EPEG be the singleton classes of the return nodes of the PEG: $R_e = \{\{e_n\} \ : \ n \in R\}$. Finally, the edges now point to the class of the child node rather than the node itself, so $E_e(e_n) = (\{e_{n_1}\}, \ldots, \{e_{n_m}\})$, where $E(n) = (n_1, \ldots, n_m)$.

8.1 The PEG Selection Problem

We see that the initial EPEG given to an equality saturation engine is essentially a glorified PEG. However, after saturation has completed, the equivalence classes will be larger because of merges. Furthermore, new nodes will be added to the EPEG, and with each one a new equivalence class. Hence, the EPEG after saturation has completed will be much larger and more complicated than the original. At that point, the task of choosing a single PEG out of that EPEG becomes more nontrivial.

Definition 1. *We define the* ***PEG Selection Problem*** *abstractly as follows. Given an EPEG $G_e = (N_e, C_e, E_e, \lambda_e, R_e)$ and a cost function $\mathbb{F}$ over EPEG nodes, find a valid PEG $G = (N, E, \lambda, R)$ such that:*

(1) Each PEG node $n \in N$ corresponds to one node EPEG node $e_n \in N_e$, and for each $n \in N, \lambda(n) = \lambda_e(e_n)$.

(2) For each PEG node $n \in N$, we choose the children of n from the child classes of e_n. Hence, if $E(n) = (n_1, \ldots, n_m)$, then $E_e(e_n) = (C_1, \ldots, C_m)$, where $e_{n_i} \in C_i, \forall i$.

(3) The return nodes of G must be chosen from the return classes of G_e, exactly one node from each class. Hence if $R_e = \{C_1, \ldots, C_k\}$, then $R = \{r_1, \ldots, r_k\}$, where $r_i \in C_i, \forall i$.

(4) The cost of the chosen PEG must be minimal among all PEGs which satisfy 1-3 above. The cost of a PEG is defined as

$$\sum_{n \in N} \mathbb{F}(e_n)$$

The PEG Selection Problem essentially requires that one choose the optimal "valid" PEG based on the nodes of the EPEG, where optimality is defined by the cost function. One subtle detail is that this definition does not require all chosen nodes to be reachable from a root node. However, since we have optimality, we know that these unreachable nodes must have cost 0 because otherwise they could be removed without affecting the other validity restrictions. Hence any nodes not reachable from a root node can be safely ignored. As it turns out, the definition provided above is too simplistic to deal with a PEG that originated from a program that has stateful operations. But even so, this simpler version of the problem statement is NP-hard, as we show in the following proof.

8.2 The MIN-SAT Problem

We define here the well-known MIN-SAT problem, which we will use in a reduction proof to show that the PEG Selection Problem is NP-Hard.

Definition 2. *We define a* ***clause*** *to be a boolean formula that is a disjunction of* ***literals****, where each literal is either a boolean variable or the negation of a boolean variable.*

Definition 3. *Let* $X = \{x_1, \ldots, x_n\}$ *be a set of boolean variables, and let* D *be a clause over those variables. Then we say a mapping* $A : X \to \{true, false\}$ ***satifies the clause*** D *if* $(\exists x_i \in X, A(x_i) = true$ *and* x_i *appears un-negated in* $D)$ *or* $(\exists x_i \in X, A(x_i) = false$ *and* x_i *appears negated in* $D)$*.*

Definition 4. *Given a set of boolean variables* $X = \{x_1, \ldots, x_n\}$ *and a set of clauses over those variables* $D = \{D_1, \ldots, D_m\}$*, the* ***MIN-SAT Problem*** *is to find a mapping* $A : X \to \{true, false\}$ *such that a minimal number of clauses in* D *are satisfied.*

8.3 NP-Hardness of the PEG Selection Problem

In this section we prove that the PEG Selection Problem is NP-Hard, by reduction from the MIN-SAT Problem.

Proof. Let $X = \{x_1, \ldots, x_n\}$ be a set of boolean variables, and let $D = \{D_1, \ldots, D_m\}$ be a set of clauses over X. We will construct an EPEG $G_e = (N_e, C_e, E_e, \lambda_e, R_e)$ in the following manner:

Nodes: For each clause $D_j \in D$, create a node $d_j \in N_e$. For each variable $x_i \in X$, create nodes $p_i, n_i \in N_e$. Create a node *root* in N_e.

Equivalence Classes: For each $d_j \in N_e$, create equivalence class $Q_j = \{d_j\}$ in C_e. For each pair of $p_i, n_i \in N_e$, create equivalence class $F_i = \{p_i, n_i\}$ in C_e. Create root class $R_e = \{\{root\}\} \subseteq C_e$ (a set containing one equivalence class, with one element).

Edges: Add edges from the root node to the F_i nodes; $E_e(root) = (F_1, \ldots, F_m)$. For each variable x_i, let $P_i = \{d_j \mid x_i \text{ appears un-negated in } D_j\}$ and let $M_i = \{d_j \mid x_i \text{ appears negated in } D_j\}$. Then impose an arbitrary ordering $\{a_i\}$ on P_i and $\{b_i\}$ on M_i and let $E_e(p_i) = (d_{a_1}, \ldots, d_{a_{|P_i|}})$, and $E_e(n_i) = (d_{b_1}, \ldots, d_{b_{|M_i|}})$.

Cost Function: Create a cost function $\mathbb{F} : N \to \mathbb{N}$ as follows: For each $d_j \in N_e$, $\mathbb{F}(d_j) = 1$. For all other nodes n, $\mathbb{F}(n) = 0$.

An example of the encoding is given in Figure 8.1(a). This figure shows the EPEG generated by the boolean expression "$(x_1 \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_3)$". The intuition behind this encoding is that each F_i equivalence class represents the value of the variable x_i in the final mapping A. If p_i is used in the optimal PEG then x_i will map to true (p = "positive") in A, and if node n_i is used then x_i will map to false (n = "negative"). The p_i and n_i nodes point only to the nodes for the clauses that they participate in, as represented by the d_j nodes. For instance, if node n_i is used in the optimal PEG, then the node for every clause that contains $\overline{x_i}$ will also be used in the optimal PEG, representing the fact that those clauses are satisfied. In the example, we have 3 variables and 3 clauses. The first clause contains x_1 un-negated and x_2 negated, so the EPEG has edges from p_1 and n_2 to d_1. Similarly, there are edges from p_2 and n_3 to d_2, and edges from n_1 and p_3 to d_3. When we get the optimal PEG, since only the d_j nodes have non-zero weight,

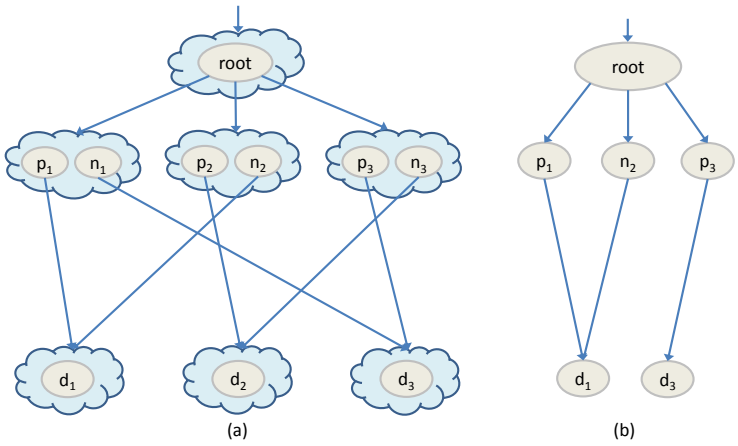


Figure 8.1: An example of the encoding of the expression "$(x_1 \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_3)$". Part (a) shows the EPEG that is produced by the encoding, and part (b) shows the PEG that results from applying the PEG Selection problem to part (a).

we will effectively minimize the number of d_j nodes that are present in the final PEG. This is exactly analogous to minimizing the number of satisfied clauses.

After creating the EPEG G_e from our MIN-SAT problem instance, we can apply the PEG Selection Problem to G_e and F to get the minimal cost PEG, $G = (N, E, \lambda, R)$. A minimal solution to the PEG Selection problem for the EPEG in Figure 8.1(a) is shown in Figure 8.1(b).

From the optimal PEG we can construct an optimal mapping $A : X \rightarrow \{true, false\}$ for the original MIN-SAT problem as follows:

$$A(x_i) = \begin{cases} true, & \text{if } p_i \in N \text{ and is reachable from } root \\ false, & \text{else} \end{cases}$$

We will show that the mapping A is a minimal solution to the original MIN-SAT Problem over X and D, and hence we have given a reduction from the MIN-SAT Problem to the PEG Selection Problem.

Since there are edges in E_e from the *root* node to each of the F_i equivalence classes, conditions (2) and (3) from above tell us that there must be edges in E from *root* to some node in F_i, for each i. This can either be p_i or n_i, or both. In the case when both are chosen, only one will be reachable from the root node. This

is obvious because the F_i class has only one parent, with only one edge sinking in F_i. Hence only one of p_i or n_i can be a descendant of the root node, so we can safely discard the other. The mapping A reflects this.

We can be sure that the discarding step will not affect the cost of the final PEG. Suppose that we have an optimal PEG G where both p_1 and n_1 are used, and we decide to discard n_1 (WLOG). Discarding n_1 will either lower the cost of G or leave it unchanged, since all node weights are non-negative. The only case when discarding n_1 could lower the cost of G is if there exists some clause node d_j such that n_1 is the only node in G that is adjacent to d_j. By optimality of G, we know that this cannot be the case, because the PEG formed by removing n_1 clearly has a lower cost than G and is still valid. Hence, there can be no decrease in the cost of G.

The choice of p_i or n_i represents whether boolean variable x_i is true or false in the assignment A. By construction, each p_i in N has edges to the equivalence class of each of the d_j nodes for which D_j would be satisfied if x_i were true. Similarly, each n_i in N has edges to each d_j for which D_j is satisfied if x_i is false. Since each d_j node is in a singleton equivalence class Q_j, they must be chosen if any p_i or n_i with an edge to Q_j is chosen. Thus, the set of d_j nodes that are in N are exactly those that correspond to clauses in D that would be satisfied by A.

We now consider the cost of N with respect to $\mathbb{F}$. Since $\mathbb{F}$ assigns cost 0 to all nodes except for the d_j nodes, we can see that the cost of N with respect to $\mathbb{F}$ is:

$$\begin{aligned}\sum_{n\in N}\mathbb{F}(n) &= \sum_{n\in N\cap Z}\mathbb{F}(n) + \sum_{n\in N/Z}\mathbb{F}(n)\\ &= \sum_{n\in N\cap Z} 1 + \sum_{n\in N/Z} 0\\ &= |N\cap Z|,\\ &\text{where } Z = \{d_i \mid D_i \in D\}\end{aligned}$$

Hence the cost of G with respect to $\mathbb{F}$ is exactly the number of clauses that are satisfied by A. Since this cost is minimal by assumption, we can see that A

gives an assignment that produces the minimal number of satisfied clauses in D, and hence is a solution to the MIN-SAT Problem over X and D.

Thus we have shown that, given an instance of the MIN-SAT Problem, we can construct an instance of the PEG Selection Problem and use the solution to construct a solution to the original MIN-SAT Problem. Therefore, the MIN-SAT Problem reduces to the PEG Selection Problem. Since the MIN-SAT Problem is NP-Hard, this means that the PEG Selection Problem is also NP-Hard. □

Since we have shown that the PEG Selection Problem is NP-Hard, there is no efficient algorithm to solve it. Thus, we are justified in our use of a *Pseudo-Boolean solver* within the Peggy system. A Pseudo-Boolean problem is a special case of an Integer Linear Programming (ILP) problem, where all of the variables are restricted to be either 0 or 1. Even though it is a simpler problem than Integer Linear Programming, it is still NP-Hard to solve it. We now show how to reduce a PEG Selection Problem instance to a Pseudo-Boolean problem instance.

8.4 Reduction from PEG Selection to Pseudo-Boolean

In addition to the constraints defined in 1, there is an additional constraint we must obey due to the nature of PEG nodes and operators. Our θ node, which is used to describe the value of a loop-varying value, is the only kind of node which should be allowed to create a loop within the graph of the PEG. That is to say, no node should be able to reach itself along any path that does not include the second child of a θ node. We can encode this restriction within the Pseudo-Boolean formulation to ensure that the resulting PEG is valid.

Given an EPEG $G_e = (N_e, C_e, E_e, \lambda_e, R_e)$ and a cost function $\mathbb{F} : N_e \to \mathbb{Z}$, we will construct a Pseudo-Boolean (PB) problem instance as follows.

Variables: The variables used in the constraints and objective function are defined as follows:

(1) For each node $n \in N_e$, add PB variable B_n.

(2) For each class $C \in C_e$, add PB variable B_C.

(3) For each pair of classes $(C_1, C_2) \in C_e \times C_e$, add PB variable $B_{C_1 \to C_2}$.

Constraints: The constraints over the variables are defined as follows:

(1) For each class $C \in C_e$, add constraint:

$$B_C = \sum_{n \in C} B_n$$

(2) Add constraint: $B_{R_e} = 1$.

(3) For each $n \in N_e$, let $E_e(n) = (C_1, \ldots, C_m)$, then add constraints:

$$B_n \implies B_{C_i},\ \forall i$$

(4) For each $n \in N_e$, let $C \in C_e$ be the class of n and let $E_e(n) = (C_1, \ldots, C_m)$, then add constraints:

$$B_n \implies B_{C \to C_i},\ \forall i$$

(exception: if $\lambda_e(n) = \theta$ and $i = 2$, do not add this constraint)

(5) For each class C, let $T_C = \{n \in C \mid \lambda_e(n) = \theta\}$. Add constraint:

$$B_{C \to C} \implies \bigvee_{n \in T_C} B_n$$

(if T_C is empty, this reduces to $B_{C \to C} \implies$ **False**)

(6) For each triple of distinct classes $C_1, C_2, C_3 \in C_e$, add constraint:

$$B_{C_1 \to C_2} \wedge B_{C_2 \to C_3} \implies B_{C_1 \to C_3}$$

Objective Function: The objective function of the PB instance is as follows:

$$\text{minimize: } \sum_{n \in N_e} B_n \cdot \mathbb{F}(n)$$

In the formulation above, we presented the constraints as a mixture of ILP notation and propositional logic notation. This is because it is trivial to convert a propositional logic expression into an ILP inequality, by the following method. Firstly, convert the expression to conjunctive normal form. Once this is done, we can encode each disjunctive clause as a separate ILP inequality. Given a disjunct $p_1 \vee \ldots \vee p_k \vee \overline{n_1} \vee \ldots \overline{n_\ell}$, we can encode this as the PB inequality $p_1 + \ldots + p_k + (1 - n_1) + \ldots + (1 - n_\ell) \geq 1$. This inequality will be true if any of the p_i are 1, or if any of the n_i are 0, and will be false only if all p_i are 0 and all n_i are 1. Since each variable can be only 0 or 1, this exactly encodes the semantics of the original propositional logic expression.

The intuition behind the above formulation is quite straightforward. For each $n \in N_e$, the variable B_n is 1 if and only if node n participates in the resulting PEG. Similarly, for each $C \in C_e$, variable B_C is 1 if and only if some node in class C participates in the resulting PEG (i.e. if C is used). The $B_{C_1 \to C_2}$ variables are designed to prevent invalid loops within the PEG. They enforce the restriction that every loop must include the 2$^{\text{nd}}$ child of a θ node. The intuitive meaning of these variables is "class C_1 can reach class C_2 along some path without traversing a θ node's second edge". Hence the exception in constraint rule (4).

Constraint rule (1) encodes the fact that a class can be used if and only if exactly one of its nodes is used. This is a stricter restriction than mentioned before, but it turns out to be an optimization. Any resulting PEG that uses more than one node from the same class is being wasteful, since the two nodes are equivalent. Hence we can prevent this by allowing exactly one. Constraint rule (2) encodes the fact that the root class must be used. Constraint rule (3) implies that if a node is used, each of its child classes must also be used. Constraint rule (4) encodes the fact that the edge from a node to its child class causes one of the $B_{C_1 \to C_2}$ variables to be true, unless it is the second child of a θ node. Constraint rule (5) encodes the fact that, if a given class C can reach itself via a path with no θ-second-child edge, then the node used in that class can only be a θ node. Finally, constraint rule (6) encodes transitivity rules for the $B_{C_1 \to C_2}$ variables.

The objective function encodes the cost of the PEG, by multiplying each

node's boolean variable by that node's cost. This will be exactly the sum of the costs of the nodes that are used in the resulting PEG. Note that this would allow some 0-cost nodes to be included even if they are not reachable from the root node. However, they can be safely discarded.

We have seen how the PEG Selection Problem as described above can be solved by a reduction to a Pseudo-Boolean problem. However, the PEG Selection problem as described above is incomplete if the programming language being represented by the PEG has stateful operations in it. As we saw in Section 6.1, Stateful operations must be treated specially, since they are instances of linear type operators. The next section details how we redefine the PEG Selection problem to deal with state, and how we solve this problem in practice.

8.5 Stateful PEG Selection Problem

To reformulate the PEG Selection Problem, we must take into account the ways that stateful operations must behave, and the ways in which the heap summary nodes may be manipulated. For traditional linear typed values, there must be exactly one instance of the value at all times [Wad90b, Gir98]. The value cannot be copied or destroyed, except in very particular situations. For example, both branches of an "if" statement will get one copy of the incoming linear value and must produce exactly one as output. Then at runtime only one of the two branches will actually be executed, so the value is indeed treated linearly. The same goes for multi-way conditionals like a "switch" statement. The other complication has to do with loops. When a linear value is used in a loop, we must make sure that it is treated linearly inside the body of the loop. The entrance/exit of the loop is one of the exceptional cases where the linear value may be merged/split.

Aside from the two special cases of loops and conditionals, all other operators that take a linear value as input *must* output one as well, even if no changes are made to the linear value. In the scope of PEGs, determining when these conditions are met can be tricky, because loops and conditionals are split into multiple θ and ϕ nodes. For any given loop in the source code, there can be at most one θ

node that has a linear type (there can be none if the loop does not read or modify the linear value at all). In addition, whenever there is a linear ϕ node (which constitutes the merge of a linear value), there must be a matching descendant operator that both the true and false branches of the ϕ share. This point will be the splitting point of the linear value. In PEGs, unlike in source code, we do not mark the splitting point of a conditional branch explicitly, but it can be calculated.

With these ideas in mind, we can begin to formulate the restrictions for a PEG that properly handes all its linear values. Unlike before, a Pseudo-Boolean formulation is not expressive enough. Instead we will rely on an Integer Linear Programming (ILP) formulation. We will see that this is also not quite good enough, which will lead us to adopt an iterative refinement approach to solving this problem.

8.6 Reduction of Stateful PEG Selection to ILP

In this section we describe how we can formulate the Stateful PEG Selection Problem as an ILP problem. However, the formulation we describe will not be complete. It will still allow certain types of invalid PEGs. Specifically, it will not properly constrain the use of ϕ nodes as they apply to linear values. The reason for this is that the rules to describe these constraints cannot be expressed in an Integer Linear Program; they are too complex. To mitigate this problem, we adopt an iterative refinement approach. We use our ILP formulation to impose all but the ϕ constraints. Whatever solution comes out of the ILP solver may be valid or may have an illegal use of ϕ operators. This fact can be checked quite easily with a Turing-complete programming language (we explore this algorithm in Section 8.7). Hence, we run the ILP solution through a separate validity checker, and if it is valid we accept it as the best solution. If not, we add a new constraint to the old formulation and run it through the ILP solver again. This new constraint essentially says that the previous answer is now invalid. Specifically, we assert that the conjunction of the values of all the variables in the previous solution implies falsehood. This will force the solver to find a new solution, which we will again

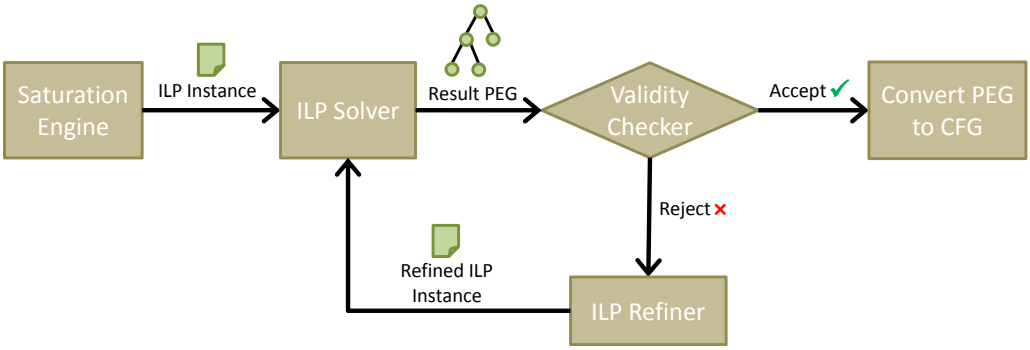


Figure 8.2: A flowchart of the iterative refinement PEG Selection Process.

subject to the validity test. We continue in this iterative manner until the ILP solver has produced a solution that the checker deems valid. Figure 8.2 shows a flowchart of the process.

In the following description, we will refer to a typical EPEG $G = (N, C, E, \lambda, R)$ and a cost function $\mathbb{F}$ over EPEG nodes, as described above. For convenience, we impose an arbitrary ordering on the nodes of the EPEG, and so $N = \{n_1, n_2, \ldots, n_K\}$. The formulation will also need to talk about loop-depth values, so we define V to be the maximal loop-depth index over all loop operators in the EPEG.

Variables. The variables of this formulation are defined as follows.

$\mathbf{N}_a$ For each $n_a \in N$, we define boolean variable $\mathbf{N}_a$. This variable will be true if and only if node n_a is used in the result PEG.

$\mathbf{INV}_a^v$ For each $n_a \in N$ and $1 \leq v \leq V$, we define boolean variable $\mathbf{INV}_a^v$. This variable will be true if and only if node n_a is invariant w.r.t. loop depth v in the result PEG.

$\mathbf{CINV}_a^v[i]$ For each $n_a \in N, 1 \leq v \leq V$, and for each child index i of node n_a, we define boolean variable $\mathbf{CINV}_a^v[i]$. This variable will be true if and only if the i-th child of node n_a is invariant w.r.t. loop depth v.

$\mathbf{IO}_{b,j}^{a,i}$ For each $n_a, n_b \in N$ and for each input index i of n_a and each output index j of n_b, we define boolean variable $\mathbf{IO}_{b,j}^{a,i}$. This variable will be true if and only if input i of n_a is connected to output index j of n_b in the result PEG.

$\mathbf{TI}_{a,i}$ For each node $n_a \in N$ and input index i of n_a, we define integer node $\mathbf{TI}_{a,i}$ to be in the range $[0, M_T]$. This variable will hold the "theta flow" quantity that goes outward from node n_a along input edge i.

$\mathbf{TO}_a$ For each node $n_a \in N$, we define integer variable $\mathbf{TO}_a$ to be in the range $[0, M_T]$. This variable will hold the "theta flow" quantity coming into node n_a from its parents.

$\mathbf{RI}_a$ For each *linear* node $n_a \in N$, we define integer variable $\mathbf{RI}_a$ to be in the range $[0, M_R]$. This variable will hold the current "root flow" that is leaving node n_a out through its child edges.

$\mathbf{RO}_a$ For each *linear* node $n_a \in N$, we define integer variable $\mathbf{RO}_a$ to be in the range $[0, M_R]$. This variable will hold the current "root flow" that is entering node n_a from its parents.

The concept of "theta flow" as described above is used to ensure that all cycles in the result PEG involve the second child edge of some θ node. Essentially, the theta flow is a value that starts at 0 at the roots of the PEG, and increments downward from parent to child. Every node has one outgoing theta flow towards its children and multiple incoming theta flows from its parents. The incoming theta flow value of a given node is the max of the outgoing theta flow values of its parents. The outgoing theta flow value of a node is equal to its incoming theta flow value plus 1. The only exception to this rule is the θ node itself. The outgoing theta flow value of a θ node along its second child edge is always 0. This allows cycles to occur in the PEG, but *only* along the second child of a θ node. No other cycles can occur, because the max of incoming parents' theta flows would not be well defined.

The "root flow" is similar, except it is designed to ensure that all operators that use linear values are reachable from the linear root node. This is important because if some linear operators are not reachable, they constitute a sub-PEG in which the state of the program has been duplicated and separated from the rest of the program. It will also ensure that the linear root dominates all linear operators,

and that the linear input parameter (the input σ node) will post-dominate all linear operators.

In the above description we refer to values M_T and M_R. These are arbitrarily chosen positive integer constants that are meant to impose a practical upper bound on the "theta flow" and the "root flow" values, respectively. Theoretically, these values should be unbounded, but in practice we can place a fairly conservative bound of around 1000 on them.

Constraints. The constraints of the program define what a valid PEG must look like. In some sense, the only "important" variables above are the $\mathbf{N}_a$ and $\mathbf{IO}_{b,j}^{a,i}$ variables, because those are the only ones needed to construct the result PEG. The other ones, along with these constraints, are what prevent invalid PEG from being returned by the ILP solver.

Rule 1: If a node's output is used, then the whole node must be used.

$$\forall a, b, i, j,\ \mathbf{IO}_{b,j}^{a,i} \implies \mathbf{N}_b$$

This rule prevents any of the IO variables from being true without the accompanying N variables being true.

Rule 2: If a node is used, then exactly one child edge is used for each input.

$$\forall a, i,\ \mathbf{N}_a = \sum_{b,j} \mathbf{IO}_{b,j}^{a,i}$$

The IO variables essentially represent the edges of the PEG, so this rule says that if a node is used then its child edges must be used as well. Also, it requires that *exactly one* child edge for each index is used.

Rule 3: The root nodes must be used.

$$\forall r \in R, 1 = \sum_{n_a \in r} n_a$$

This rule says that one node from each root class must be used.

Rule 4: The children of a node define its child invariance values.

$$\forall a, i, b, j, v,\ \mathbf{IO}_{b,j}^{a,i} \implies (\mathbf{CINV}_a^v[i] = \mathbf{INV}_b^v)$$

The CINV variables are really just a convenience to help compute the values of the INV variables. Each node will learn its "child invariance" from its children and store that in the CINV variables. The next rule we present uses the child invariance to compute the invariance for the node itself.

Rule 5: The child invariance of a node determines its invariance.

This rule is divided into several cases based on what the operator of the node is. We present these cases below. For a given node n_a:

(1) For θ_i nodes:

θ_i varies in loop depth i: $\mathbf{N}_a \implies \mathbf{INV}_a^i$.

θ_i's first child must not vary in i: $\mathbf{N}_a \implies \mathbf{CINV}_a^i[0]$.

θ_i's second child must vary in i: $\mathbf{N}_a \implies \mathbf{CINV}_a^i[1]$.

(2) For pass_i nodes:

pass_i does not vary in i: $\mathbf{N}_a \implies \mathbf{INV}_a^i$.

pass_i's child must vary in i: $\mathbf{N}_a \implies \mathbf{CINV}_a^i[0]$.

(3) For eval_i nodes:

eval_i varies in i iff its 2^{nd} child does: $\mathbf{N}_a \implies \big(\mathbf{INV}_a^i = \mathbf{CINV}_a^i[1]\big)$.

eval_i's first child must vary in i: $\mathbf{N}_a \implies \mathbf{CINV}_a^i[0]$.

(4) For all other nodes:

A node is invariant in v if and only if all of its children are also invariant in v:

$$\forall v,\ \mathbf{N}_a \implies (\mathbf{CINV}_a^v[0] \wedge \ldots \wedge \mathbf{CINV}_a^v[m] \iff \mathbf{INV}_a^v)$$

In the above description, we only describe how a loop operator defines its invariance in terms of its own loop depth i. For other loop depths $v \neq i$, these operators exhibit the same behavior as in rule (4) above.

Rule 6: Root nodes have total invariance.

$$\forall r \in R, \forall n_a \in r, \forall v,\ \mathbf{N}_a \implies \mathbf{INV}_a^v$$

The root of a PEG must not be inside any loop.

Rule 7: For linear domain operators, any linear child must vary at least as much as any nonlinear child.

$$\forall a, \forall v, \forall \text{linear } i, \forall \text{nonlinear } j,\ \mathbf{CINV}_a^v[i] \implies \mathbf{CINV}_a^v[j]$$

If any nonlinear child of a domain operator varies in some loop depth i, then all linear children must do so as well. This is because the node will have to be inside some loop at depth i. Hence, the node operator will be evaluated at every loop iteration. If the linear child is invariant, then the operator clearly expects to get the same state value at each iteration. However, just by evaluating the operator we change the state implicitly. Hence, the linear input to the operator *must* vary since it will do so implicitly anyway.

Rule 8: A θ node cannot be its own second child.

$$\forall a,\ \mathbf{IO}_{a,0}^{a,1} = \text{false},\ \text{if } n_a \text{ is a } \theta \text{ node}$$

Rule 9: All linear nodes must be reachable from the linear root.

This rule consists of several subrules that describe the "root flow" mentioned above. Every linear node has both an incoming and outgoing root flow value. The incoming root flow $\mathbf{RI}_a$ comes from the parents of the node, and the outgoing root flow $\mathbf{RO}_a$ goes out to the children of the node.

(1) The linear root's flow starts at 1.

$$\mathbf{RI}_r = 1, \mathbf{RO}_r = 2,\ \text{for linear root node } n_r$$

(2) A linear node can only be used if it has a nonzero root flow.

$$\forall \text{linear } n_a \in N,\ \mathbf{N}_a \implies (\mathbf{RI}_a > 0)$$

(3) The outgoing root flow is 1 more than the incoming root flow.

$$\forall a,\ \mathbf{RO}_a = \mathbf{RI}_a + 1$$

(4) The incoming root flow is equal to the min of the outgoing root flow over all parent nodes.

$$\forall a,\ \mathbf{N}_a \implies \left(\mathbf{RI}_a = \min\left\{\mathbf{RO}_b \mid \mathbf{IO}_{a,j}^{b,i} \text{ is true and linear}\right\}\right)$$

These rules constrain the linear operators in such a way that no linear operator can be unreachable from the linear root node. Rule 2 states that all linear nodes must have a nonzero root flow, and the other rules state that no node may have a nonzero root flow unless there is an increasing path of root flows reaching it from the linear root node.

Rule 10: No cycles may exist in the PEG unless they use the second child edge of a θ node.

This rule is divided into a few subrules that make use of the idea of "theta flow" described above. These rules use the TI and TO variables to ensure that the only cycles allowed in a PEG must use the second child edge of a θ node.

(1) The theta flow input of a node is greater than the theta flow output of all its parents.

$$\forall a, b, i, j,\ \mathbf{IO}_{b,j}^{a,i} \implies (\mathbf{TO}_b \geq \mathbf{TI}_{a,i})$$

(2) The theta flow output of a node is related to its theta flow input.

For a θ node n_a: $\mathbf{TI}_{a,0} = \mathbf{TO}_a + 1,\ \mathbf{TI}_{a,1} = 0.$

For a root node n_a: $\forall i,\ \mathbf{TI}_{a,i} = 0,\ \mathbf{TO}_a = 0.$

For all other nodes n_a: $\forall i,\ \mathbf{TI}_{a,i} = \mathbf{TO}_a + 1.$

The "theta flow" is a value that increases from parent to child within the PEG. The only exception is that the second child of a θ node always gets 0 as its incoming theta flow. This allows a cycle to form without having an ever-increasing theta flow value. No other cycles are possible.

These constraints define exactly how a valid PEG must look when being chosen from an EPEG. The set of N and IO variables that are true in the solution to this problem will describe exactly the shape of the result PEG. In order to find the best PEG, we must include the objective function to encode the cost model over the nodes.

Objective Function. The objective function of the ILP formulation is slightly more complicated than the one for the Pseudo-Boolean formulation. This is because we explicitly keep track of the invariance of the nodes in this formulation, rather than relying on the invariance info from the EPEG itself. We wish to make the cost of a node higher if it is inside of a loop. Hence, we add negative weight to the INV variables for each node. This way, if a node varies in a particular loop depth, the cost will be higher. We define a variance multipler **VM** and define the objective function of the formulation as follows:

$$\sum_{a} \left(\mathbb{F}(n_a) \cdot \mathbf{N}_a - \sum_{i=1}^{V} \mathbf{VM}^i \cdot \mathbf{INV}_a^i \right)$$

This objective function weights each PEG node according to its cost, and adds weight proportional to the variance of the node. With this objective function, we can find the optimal PEG in the EPEG according to the cost function $\mathbb{F}$.

8.7 The PEG Validity Checker

As we have stated above, the Stateful PEG Selection problem is incomplete. It relies on a separate checker to determine whether or not the chosen PEG is properly linearized with respect to ϕ nodes. In this section we describe the algorithm for implementing this check.

Recall from Section 6.1 that the real problem with linearity with respect to ϕ nodes is that some operations have both linear and non-linear outputs, and we can encounter cases where one of the outputs is executed "more conditionally" than another. That is, one of the outputs is always used and another is only used if a certain condition is true. If the conditional output is the linear output, then it is unclear how to convert the operation back to imperative code. In the

imperative version we cannot split up the operation into multiple parts. Hence, if we were to execute the operation unconditionally to get one value out, we would inevitably cause the side effects to occur, and hence evaluate the linear output unconditionally as well, which is not the correct semantics. Note that it is allowable for the nonlinear output to be executed more conditionally than the linear output. This would amount to something like saving the return value of a function call, but only using that value under certain conditions. As long as we unconditionally want the side effects to occur, the operation can be executed safely.

The solution embodied by the validity checker algorithm is to disallow PEGs that have this pattern of ϕ nodes. In order to do this, we must simply identify every operator in the PEG that has both linear and non-linear outputs, and for each one ensure that its outputs have compatible ϕ profiles. The ϕ profile of an output of an operator is the set of paths of nested ϕ nodes that can reach the output. Each ϕ node in a path is represented as a triple $(cond, case, var)$, where *cond* is the PEG node that represents the ϕ's condition, *case* is either T or F telling which child of the ϕ reaches the output in question, and *var* is the maximum loop index in which the ϕ node varies. In fact, we do not need the sets of profiles to be identical for all outputs. The profiles for linear outputs just need to be prefixes of the profiles for nonlinear outputs. We formalize these notions below.

Definition 5. *We define a* ϕ***-path*** *to be a sequence of triples representing ϕ nodes in a PEG: $(cond_1, case_1, var_1) \rightarrow \ldots \rightarrow (cond_n, case_n, var_n)$. The empty path is a valid path, and is denoted by $\emptyset$. We call a set of ϕ-paths a* ϕ***-path profile****.*

A single ϕ-path represents one path of nested ϕ's in the PEG that reaches a particular output of a particular operator. Triples that occur later in a path are nested deeper than those that occured earlier. The empty path effectively means unconditional execution of the output.

Definition 6. *Given ϕ-paths p_1, p_2, p_3, we say that p_3* ***subsumes*** *p_1* ***and*** *p_2 if there exist subpaths A and B such that $p_1 = A \rightarrow (cond_i, T, var_i) \rightarrow B$ and $p_2 = A \rightarrow (cond_i, F, var_i) \rightarrow B$ and $p_3 = A \rightarrow B$. In this case, we call p_1 and p_2 a* ***redundant pair****.*

The intuition behind this definition is that if the same output occurs on both sides of a single ϕ, then it is being executed in either case, and hence it is not actually conditional upon that ϕ node. When we collect sets of paths, we will always want to remove redundant pairs and replace them with the paths that subsume them.

Definition 7. *Given a ϕ-path profile S, we define the process of **pruning** S as follows:*

while** $\exists$ redundant pair $p_1, p_2 \in S$ **do

Let p_3 be the path that subsumes p_1 and p_2.

$S \leftarrow S \backslash \{p_1, p_2\} \cup \{p_3\}$

end while

*We call the result of this process a **pruned profile**.*

Definition 8. *Given two ϕ-path profiles A and B, we say that A **implies** B if*

$$\forall p_a \in A,\ \exists p_b \in B,\ \exists p_c,\ p_b = p_c \rightarrow p_a.$$

Intuitively, a profile A implies another profile B if every path in A is "more conditional" than some path in B. We can see that in general we want the profile for a nonlinear output to imply the profile for a linear output of the same operator. With these definitions, we are now ready to express the validity condition for PEGs.

Definition 9. *Let G be a PEG. Let L be the set of nodes in G whose operators have both linear and nonlinear outputs. Then we say that G **is valid with respect to conditionals** if for every node $\ell \in L$, the pruned profiles of every nonlinear output of ℓ implies the pruned profile of the linear output of ℓ.*

For any PEG that is valid with respect to conditionals, we have made sure that there are no cases when a ϕ node causes unconditional execution of a nonlinear output with conditional execution of a linear output. This means that the PEG can be reverted to a CFG in a fairly straightforward manner, and the semantics of the PEG will be preserved.

In the example from Figure 6.1(c), the profile for the right-hand call to **set** is $A = \{(\mathbf{b}, F, 0)\}$ for the linear output and $B = \{\emptyset\}$ for the nonlinear output.

Hence A does not imply B, and so this PEG is not valid. Notice that for the left-hand call to **set**, the profile for the nonlinear output is the empty set, which trivially implies all other profiles, and hence is still valid.

Summary

In this chapter we describe an important subproblem that we encounter when optimizing using Equality Saturation. That is finding the optimal PEG within the saturated EPEG. We proved formally that, given a per-node cost function, the task of finding the optimal PEG is NP-Hard. We presented a naive first attempt at solving the problem by reducing it to a Pseudo-Boolean problem instance. We then discovered that this is insufficient for PEGs with linearly typed operations, and reformulated the problem to handle the linear values. Finally, we describe an Integer Linear Programming solution to the problem, that involves an iterative refinement approach.

In the next chapter, we perform an experimental evaluation of Peggy as a tool for optimization. We show that Peggy can perform many different types of classical optimizations, as well as some that are difficult to do in a classical optimzer. We also show that Equality Saturation is a useful tool for performing optimizations on real programs.

Chapter 9

Evaluation: Optimization

In this chapter we use our Peggy implementation to validate two hypotheses about our approach for structuring optimizers: our approach is practical both in terms of space and time (Section 9.1), and it is effective at discovering both simple and intricate optimization opportunities (Section 9.2).

9.1 Time and space overhead

To evaluate the running time of the various Peggy components, we compiled SpecJVM, which comprises 2,461 methods. We used the Pueblo pseudo-boolean solver to solve the PEG selection problem, as described in Chapter 8. For 1% of these methods, Pueblo exceeded a one minute timeout we imposed on it, in which case we just ran the conversion to PEG and back. We imposed this timeout because in some rare cases, Pueblo runs too long to be practical.

The following table shows the average time in milliseconds taken per method for the 4 main Peggy phases (for Pueblo, a timeout counts as 60 seconds).

	CFG to PEG	Saturation	Pueblo	PEG to CFG
Time	13.9 ms	87.4 ms	1,499 ms	52.8 ms

All phases combined take slightly over 1.5 seconds. An end-to-end run of Peggy is on average 6 times slower than Soot with all of its intraprocedural

(a) EQ Analyses	Description
1. Built-in EPEG ops	Axioms about primitive PEG nodes (ϕ, θ, eval, pass)
2. Basic Arithmetic	Axioms about arithmetic operators like $+$, $-$, $*$, /, <<, >>
3. Constant Folding	Equates a constant expression with its constant value
4. Java-specific	Axioms about Java operators like field/array accesses
5. TRE	Replaces the body of a tail-recursive procedure with a loop
6. Method Inlining	Inlining based on intraprocedural class analysis
7. Domain-specific	User-provided axioms about application domains

(b) Optimizations	Description
8. Constant Prop/Fold	Traditional Constant Propagation and Folding
9. Simplify Algebraic	Various forms of traditional algebraic simplifications
10. Peephole SR	Various forms of traditional peephole optimizations
11. Array Copy Prop	Replace read of array element by last expression written
12. CSE for Arrays	Remove redundant array accesses
13. Loop Peeling	Pulls the first iteration of a loop outside of the loop
14. LIVSR	Loop-induction-variable Strength Reduction
15. Interloop SR	Reduce strength of nested loop computation
16. Entire-loop SR	Entire loop becomes one op, *e.g.* n incrs becomes "plus n"
17. Loop-op Factoring	Factor op out of a loop, *e.g.* multiplication
18. Loop-op Distributing	Distribute op into loop, where it cancels out with another
19. Partial Inlining	Inlines part of method in caller, but keeps the call
20. Polynomial Factoring	Evaluates a polynomial in a more efficient manner

(c) DS Opts	Description
21. DS LIVSR	LIVSR on domain ops like matrix addition and multiply
22. DS Code Hoisting	Code hoisting based on domain-specific invariance axioms
23. DS Remove Redundant	Removes redundant computations based on domain axioms
24. Temp. Object Removal	Remove temp objects made by calls to, *e.g.*, matrix libraries
25. Math Lib Specializing	Specialize matrix algs based on, *e.g.*, the size of the matrix
26. Design-pattern Opts	Remove overhead of common design patterns
27. Method Outlining	Replace code by method call performing same computation
28. Specialized Redirect	Replace call with more efficient call based on calling context

Figure 9.1: Optimizations performed by Peggy. Throughout this table we use the following abbreviations: EQ means "equality", DS means "domain-specific", TRE means "tail-recursion elimination", SR means "strength reduction"

optimizations turned on. Nearly all of our time is spent in the pseudo-boolean solver. We have not focused our efforts on compile-time, and we conjecture there is significant room for improvement, such as better pseudo-boolean encodings, or other kinds of profitability heuristics that run faster.

Since Peggy is implemented in Java, to evaluate memory footprint, we limited the JVM to a heap size of 200 MB, and observed that Peggy was able to compile all the benchmarks without running out of memory.

In 84% of compiled methods, the engine ran to complete saturation, without imposing bounds. For the remaining cases, the engine limit of 500 was reached, meaning that the engine ran until fully processing 500 expressions in the EPEG, along with all the equalities they triggered. In these cases, we cannot provide a completeness guarantee, but we can give an estimate of the size of the explored state space. In particular, using just 200 MB of heap, our EPEGs represented more than 2^{103} versions of the input program (using geometric average).

9.2 Implementing optimizations

The main goal of our evaluation is to demonstrate that common, as well as unanticipated, optimizations result in a natural way from our approach. To achieve this, we implemented a set of basic equality analyses, listed in Figure 9.1(a). We then manually browsed through the code that Peggy generates on a variety of benchmarks (including SpecJVM) and made a list of the optimizations that we observed. Figure 9.1(b) shows the optimizations that we observed fall out from our approach using equality analyses 1 through 6, and Figure 9.1(c) shows optimizations that we observed fall out from our approach using equality analyses 1 through 7. Based on the optimizations we observed, we designed some micro-benchmarks that exemplify these optimizations. We then ran Peggy on each of these micro-benchmarks to show how much these optimizations improve the code when isolated from the rest of the program.

Figure 9.2 shows our experimental results for the runtimes of the micro-benchmarks listed in Figure 9.1(b) and (c). The y-axis shows run-time normalized

to the runtime of the unoptimized code. Each number along the x-axis is a micro-benchmark exemplifying the optimization from the corresponding row number in Figure 9.1. The "rt" and "sp" columns correspond to our larger raytracer benchmark and SpecJVM, respectively. The value reported for SpecJVM is the average ratio over all benchmarks within SpecJVM. Our experiments with Soot involve running it with all intra-procedural optimizations turned on, which include: common sub-expression elimination, lazy code motion, copy propagation, constant propagation, constant folding, conditional branch folding, dead assignment elimination, and unreachable code elimination. Soot can also perform interprocedural optimizations, such as class-hierarchy-analysis, pointer-analysis, and method-specialization. We did not enable these optimizations when performing our comparison against Soot, because we have not yet attempted to express any interprocedural optimizations in Peggy. In terms of runtime improvement, Peggy performed very well on the micro-benchmarks, optimizing all of them by at least 10%, and in many cases much more. Conversely, Soot gives almost no runtime improvements, and in some cases makes the program run slower. For the larger raytracer benchmark, Peggy is able to achieve a 7% speedup, while Soot does not improve performance. On the SpecJVM benchmarks both Peggy and Soot had no positive effect, and Peggy on average made the code run slightly slower. This leads us to believe that traditional intraprocedural optimizations on Java bytecode generally produce only small gains, and in this case there were few or no opportunities for improvement.

With effort similar to what would be required for a compiler writer to implement the optimizations from part (a), our approach enables the more advanced optimizations from parts (b) and (c). Peggy performs some optimizations (for example 15 through 20) that are quite complex given the simplicity of its equality analyses. To implement such optimizations in a traditional compiler, the compiler writer would have to explicitly design a pattern that is specific to those optimizations. In contrast, with our approach these optimizations fall out from the interaction of basic equality analyses without any additional developer effort, and without specifying an order in which to run them. Essentially, Peggy finds the right sequence of equality analyses to apply for producing the effect of these

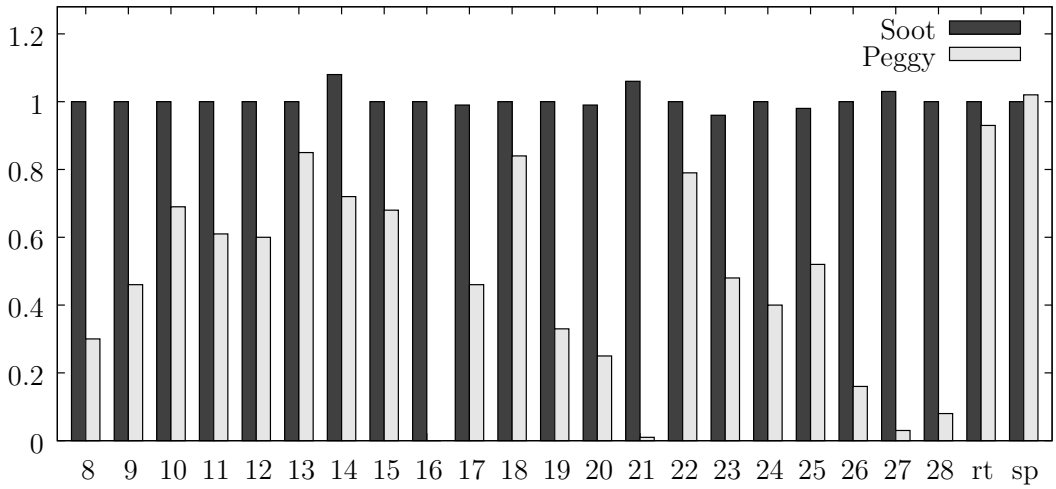


Figure 9.2: Runtimes of generated code from Soot and Peggy, normalized to the runtime of the unoptimized code. The x-axis denotes the optimization number from Figure 9.1, where "rt" is our raytracer benchmark and "sp" is the average over the SpecJVM benchmarks.

complex optimizations.

With the addition of domain-specific axioms, our approach enables even more optimizations, as shown in part (c). To give a flavor for these domain-specific optimizations, we describe two examples.

The first is a ray tracer application (5 KLOCs) that one of the authors had previously developed. To make the implementation clean and easy to understand, the author used immutable vector objects in a functional programming style. This approach however introduces many intermediate objects. With a few simple vector axioms, Peggy is able to remove the overhead of these temporary objects, thus performing a kind of deforestation optimization. This makes the application 7% faster, and reduces the number of allocated objects by 40%. Soot is not able to recover any of the overhead, even with interprocedural optimizations turned on. This is an instance of a more general technique where user-defined axioms allow Peggy to remove temporary objects (optimization 24 in Figure 9.1).

Our second example targets a common programming idiom involving `List`s, which consists of checking that a `List` contains an element e, and if it does, fetching

and using the index of the element. If written cleanly, this pattern would be implemented with a branch whose guard is `contains`(e) and a call to `indexOf`(e) on the true side of the branch. Unfortunately, `contains` and `indexOf` would perform the same linear search, which makes this clean way of writing the code inefficient. Using the equality axiom l.`contains`$(e) = (l$.`indexOf`$(e) \neq -1)$, Peggy can convert the clean code into the hand-optimized code that programmers typically write, which stores `indexOf`(e) into a temporary, and then branches if the temporary is not -1. An extensible rewrite system would not be able to provide the same easy solution: although a rewrite of l.`contains`(e) to $(l$.`indexOf`$(e) \neq -1)$ would remove the redundancy mentioned above, it could also degrade performance in the case where the list implements an efficient hash-based `contains`. In our approach, the equality simply adds information to the EPEG, and the profitability heuristic can decide after saturation which option is best, taking the entire context into account. In this way our approach transforms `contains` to `indexOf`, but only if `indexOf` would have been called anyway.

These two examples illustrate the benefits of user-defined axioms. In particular, the clean, readable, and maintainable way of writing code can sometimes incur performance overheads. User-defined axioms allow the programmer to reduce these overheads while keeping the code base clean of performance-related hacks. Our approach makes domain-specific axioms easier to add for the end-user programmer, because the programmer does not need to worry about what order the user-defined axioms should be run in, or how they will interact with the compiler's internal optimizations. The set of axioms used in the programs from Figure 9.1 is presented in Appendix D.1.

Summary

This chapter presented an experimental evaluation of the Peggy optimizer on several real benchmark programs. We showed the time and space overhead of the running the optimizer, as well as a set of specific optimizations that we were able to perform. Finally, we presented our results in optimizing several micro-

benchmarks as well as two real-world code examples.

In the next chapter we will move away from optimization and talk instead about another major application for Equality Saturation. Specifically, we will talk about how Equality Saturation can be used to perform translation validation, which is a technique to prove that some other program transformer is semantically valid. This is useful for program verification, and it is a natural extension of the existing Equality Saturation engine.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER grant CCF-0644306.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Logical Methods in Computer Science 2010*. The dissertation author was the secondary investigator and author of this paper.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. The dissertation author was the secondary investigator and author of this paper.

Chapter 10

Translation Validation

Until now, we have only presented Peggy as a tool for performing optimizations. In this section, we will examine a new use for the Equality Saturation technique.

When an optimizer transforms a program, there is a specific goal in mind; namely, that the resulting program should be more efficient in some way, be it runtime, memory usage, etc. This goal is related to the utility of the optimizer. In addition, the optimizer is under the restriction that it must preserve the semantics of the program being transformed. That is, the program after optimization must perform the same computation as the program before optimization. This restriction is related to the correctness of the optimizer.

10.1 Translation Validation

Both of these factors are important, but we argue that the latter is moreso. It does little good to have an optimizer that doubles the speed of your program if it no longer produces the same output, or computes the same value. Hence, ensuring compiler correctness is of paramount importance. This has led to the creation of *Translation Validation* [PSS98b], which is a process whereby one attempts to prove the correctness of a program transformer such as an optimizer.

There are multiple ways to approach translation validation, but we will focus on one in particular. Our method involves looking not at the optimizer

itself, but instead at its output. Given a program P, we can run an optimizer on it to produce a new program, P'. Our translation validation tool will then examine the pair of P and P' to test for semantic equivalence. This test can produce one of three answers: Equivalent, Not Equivalent, or Unknown. If the result is Equivalent, then we have shown that the particular transformations performed on P to produce P' are valid, and preserve correctness. If the result is Not Equivalent, then we have found a correctness bug in the optimizer, and have found a concrete instance of a program that triggers it. If the result is Unknown, then the tool was unable to prove anything, and no conclusions can be drawn.

The setup as described above is not a decision procedure for validating an optimizer's correctness, but is instead more of a confidence builder. It cannot detect the absence of correctness bugs, only their presence. If we can validate the translation of a large corpus of programs by a particular optimizer, we have still not proven it correct, but we have increased our confidence in its correctness.

10.2 Translation Validation in Peggy

We have extended the Peggy system to include a translation validation engine for both Java and LLVM. We achieve this by using the same Equality Saturation engine that we use for optimization. Equality Saturation is inherently designed to find equivalences between programs, so it is a perfect tool for this job. The validation process proceeds as follows. Firstly, we convert both the before and after programs into our PEG representation. Secondly, we merge the two PEGs into the same PEG-space, so that they share as many nodes as possible. Now that both programs are in the same PEG-space, we can begin to describe equivalences between them. As we did for optimization, we input the merged PEG into an initial EPEG and perform Equality Saturation on it. If, at any time, the corresponding pairs of root nodes of the two PEGs become equivalent, then we have shown that the two programs are equivalent.

Example 1. Consider the original code in Figure 10.1(a) and the optimized code in Figure 10.1(b). There are two optimizations that LLVM applied here.

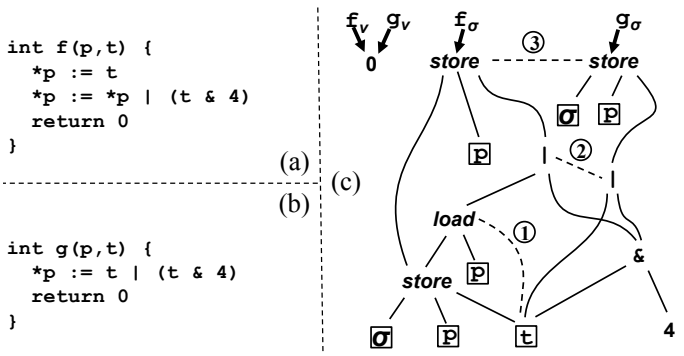


Figure 10.1: (a) Original code (b) Optimized code (c) Combined EPEG.

First, LLVM performed copy propagation through the location `*p`, thus replacing `*p` with `t`. Second, LLVM removed the now-useless store `*p := t`.

Figure 10.1(c) shows the PEGs for `f` and `g`. The labels $\mathtt{f}_v$ and $\mathtt{f}_\sigma$ point to the value and heap returned by `f` respectively, and likewise for `g`. The PEG for `f` takes a heap σ as an input parameter (in addition to `p` and `t`), and produces a new heap, labeled $\mathtt{f}_\sigma$, and a return value, labeled $\mathtt{f}_v$.

Our approach to translation validation builds the PEGs for both the original and the optimized programs in the same PEG space, meaning that nodes are reused when possible. In particular note how `t & 4` is shared. Once this combined PEG has been constructed, we apply equality saturation. If through this process Peggy infers that node $\mathtt{f}_\sigma$ is equal to node $\mathtt{g}_\sigma$ and that node $\mathtt{f}_v$ is equal to node $\mathtt{g}_v$, then Peggy has shown that the original and optimized functions are equivalent. In the diagrams we use dashed lines to represent PEG node equality (in the implementation, we store equivalence classes of nodes using Tarjan's union-find data structure).

Peggy proves the equivalence of `f` and `g` in the following three steps:

Peggy adds equality ① using axiom: $load(store(\sigma, p, v), p) = v$

Peggy adds equality ② by congruence closure: $a = b \Rightarrow f(a) = f(b)$

Peggy adds equality ③ by axiom: $store(store(\sigma, p, v_1), p, v_2) = store(\sigma, p, v_2)$

By equality ③, Peggy has shown that `f` and `g` return the same heap, and are therefore equivalent since they are already known to return the same value 0.

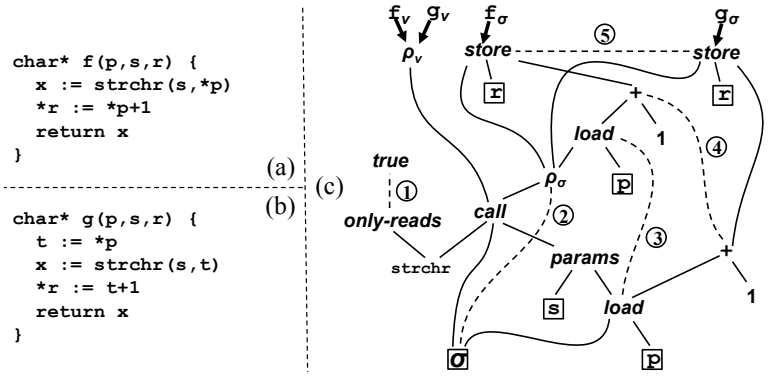


Figure 10.2: (a) Original code (b) Optimized code (c) Combined EPEG.

Example 2. As a second example, consider the original function `f` in Figure 10.2(a) and the optimized version `g` in Figure 10.2(b). `p` is a pointer to an `int`, `s` is a pointer to a `char`, and `r` is a pointer to an `int`. The function `strchr` is part of the standard C library, and works as follows: given a string `s` (i.e., a pointer to a `char`), and an integer `c` representing a character[1], `strchr(s,c)` returns a pointer to the first occurrence of the character `c` in the string, or `null` otherwise. The optimization is correct because LLVM knows that `strchr` does not modify the heap, and the second load `*p` is redundant.

The combined PEGs are shown in Figure 10.2(c). The call to `strchr` is represented using a *call* node, which has three children: the name of the function, the incoming heap, and the parameters (which are passed as a tuple created by the *params* node). A *call* node returns a pair consisting of the return value and the resulting heap. We use projection operators ρ_v and ρ_σ to extract the return value and the heap from the pair returned by a *call* node.

To give Peggy the knowledge that standard library functions like `strchr` do not modify the heap, we have annotated such standard library functions with an *only-reads* annotation. When Peggy sees a call to a function foo annotated with *only-reads*, it adds the equality $\textit{only-reads}(foo) = \textit{true}$ in the PEG. Equality ① in Figure 10.2(c) is added in this way.

Peggy adds equality ② using: $\textit{only-reads}(n) = \textit{true} \Rightarrow \rho_\sigma(\textit{call}(n, \sigma, p)) = \sigma$. This axiom encodes the fact that a read-only function call does not modify the

[1]It may seem odd that `c` is not declared a `char`, but this is indeed the interface.

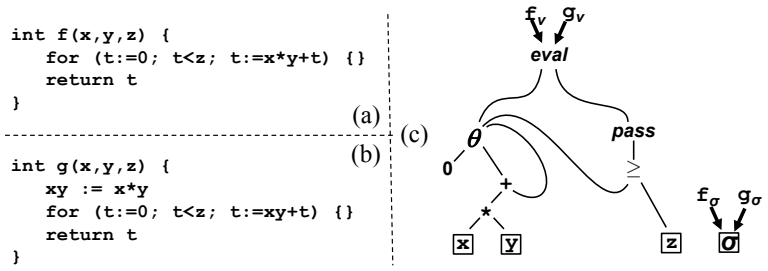


Figure 10.3: (a) Original code (b) Optimized code (c) Combined EPEG.

heap. Equalities ③, ④, and ⑤ are added by congruence closure.

In these 5 steps, Peggy has identified that the heaps $\mathtt{f}_\sigma$ and $\mathtt{g}_\sigma$ are equal, and since the returned values $\mathtt{f}_v$ and $\mathtt{g}_v$ are trivially equal, Peggy has shown that the original and optimized functions are equivalent.

Example 3. As a third example, consider the original code in Figure 10.3(a) and the optimized code in Figure 10.3(b). LLVM has pulled the loop-invariant code `x*y` outside of the loop. The combined PEG for the original function `f` and optimized function `g` is shown in Figure 10.3. As it turns out, `f` and `g` will produce the exact same PEG, so let us focus on understanding the PEG itself. The θ node represents the *sequence* of values that `t` takes throughout the loop. The $\geq$ node is a lifting of $\geq$ to sequences, and so it represents the sequence of values that `t` $\geq$ `z` takes throughout the loop. The eval/pass pair is used to extract the value of `t` after the loop. Therefore, eval returns the value of `t` after the last iteration of the loop

As Figure 10.3 shows, the PEG for the optimized function `g` is the same as the original function `f`. Peggy has validated this example just by converting to PEGs, without even running equality saturation. One of the key advantages of PEGs is that they are agnostic to code-placement details, and so Peggy can validate code placement optimizations such as loop-invariant code motion, lazy code motion, and scheduling by just converting to PEGs and checking for syntactic equality.

10.3 Evaluation

In this section we evaluate Equality Saturation as a mechanism for performing translation validation. We performed two experiments to evaluate translation validation on Java and LLVM separately. These experiments were performed at different stages of Peggy's development, and hence do not follow the same experimental methodology. However, they both show that Equality Saturation is an effective tool for performing translation validation of imperative programs.

10.3.1 Translation Validation in Java

We used Peggy to perform translation validation for the Soot optimizer [VRHS+99]. In particular, we used Soot to optimize a set of benchmarks with all of its intraprocedural optimizations turned on. The benchmarks included SpecJVM [spe], along with other programs, comprising a total of 3,416 methods. After Soot finished compiling, for each method we asked Peggy's saturation engine to show that the original method was equivalent to the corresponding method that Soot produced. The engine was able to show that 98% of methods were compiled correctly.

Among the cases that Peggy was unable to validate, we found three methods that Soot optimized *incorrectly*. In particular, Soot incorrectly pulled statements outside of an intricate loop, transforming a terminating loop into an infinite loop. It is a testament to the power of our approach that it is able not only to perform optimizations, but also to validate a large fraction of Soot runs, and that in doing so it exposed a bug in Soot. Furthermore, because most false negatives are a consequence of our coarse heap model (single σ node), a finer-grained model can increase the effectiveness of translation validation, and it would also enable more optimizations.

10.3.2 Translation Validation in LLVM

We used our updated Peggy tool to perform translation validation for LLVM on SPEC CPU 2006 integer C benchmarks. We used LLVM version 2.3, and we

Benchmark	#func	#instr	%succ	To PEG	Engine Time	
					succ	fail
400.perlbench	1,868	317,016	84.9%	0.878s	1.592s	23s
401.bzip2	103	18,820	74.0%	0.943s	0.614s	16s
403.gcc	5,619	982,168	84.8%	0.829s	0.670s	24s
429.mcf	24	2,865	70.8%	0.308s	0.907s	15s
433.milc	235	24,457	83.3%	0.631s	0.249s	12s
456.hmmer	539	64,677	84.2%	0.700s	1.089s	17s
458.sjeng	144	29,767	77.1%	0.648s	0.688s	27s
462.libquantum	115	7,939	79.1%	0.343s	0.748s	15s
464.h264ref	590	142,521	78.7%	1.147s	0.885s	26s
470.lbm	19	3,628	73.7%	0.349s	0.088s	5s
482.sphinx3	370	32,103	88.1%	0.331s	0.588s	20s

Figure 10.4: Results of running Peggy's translation validator on SPEC 2006 benchmarks. The times listed in the "Engine Time" columns are averages.

enabled the following optimizations in our experiment: dead code elimination, global value numbering, sparse conditional constant propagation, loop-invariant code motion, loop deletion, loop unswitching, dead store elimination, constant propagation, conditional propagation, and basic block placement.

Figure 10.4 shows the results: "Benchmark" is the benchmark name; "#func" is the total number of functions in the benchmark; "#instr" is the total number of instructions; "%succ" is the percentage of functions whose compilation Peggy validated (in measuring this and subsequent columns, we ignored functions that LLVM did not perform any optimizations on – about 0.16% of all functions); "To PEG" is the average time per function taken to convert the CFG into a PEG; "Engine Time" is the average time per function taken by the Peggy equality saturation engine ("succ" is the average over successful runs, and "fail" is the average over failed runs).

For these experiments, we found that the greatest difficulty Peggy had in validating was in dealing with pointers and aliasing. For instance, there were many

cases where the validation relied on using the following axiom: `load(store(`σ`,` P`,` V`),` P`) =` V. This axiom encodes the fact that a load from a pointer P after a store of value V to the same pointer P will yield the value V that was stored, if no other heap-modifying operations happen in between. This axiom is straightforward, but unfortunately the code is often not shaped in this way. More often, there are several heap operations in between a store to P and the next load from P, so the axiom will not fire.

In some cases, we can use alias information to get around this problem. For instance, if the code contained a store to P, then a store to Q, then a load from P, then this pattern will not immediately trigger the above axiom. However, if we can prove with alias information that Q and P definitely do not alias, then we can safely move the load of P before the store to Q. Now the load of P immediately follows the store to P, so the axiom applies.

This pattern occurs far more frequently in real code. With the proper alias information, we are still able to get the results we want. However, we have seen that before we can apply the critical axiom, we must apply other helper axioms first. In our example only one was necessary, but in general many such axioms might be needed *for each load/store pair*. Also, gathering the alias information itself requires additional axiom applications, and if the right paths are not followed then there is not enough information to allow the critical axioms to fire. In general, we are left with many situations where Peggy *could* validate the function pair, but it requires the application of the right axioms in the right places to do so. Hence we see false negatives because of our heuristic cutoff mechanism, not because Peggy is incapable of performing the validation.

Summary

We have shown how Equality Saturation can be used to perform translation validation of pairs of programs. Since Equality Saturation is primarily designed to find equivalences between program fragments, it is ideally suited to this task. We have seen multiple examples of how Peggy performs translation validation by

applying axioms to both of the input programs simultaneously. We also showed experimental evidence that Equality Saturation is an effective device for performing translation validation on real programs. In the Java setting, we validated 98% of the runs of the Soot optimizer on the SpecJVM benchmark, and in the process discovered a bug in Soot. In the LLVM setting, we were able to validate LLVM's optimizer on the SPEC CPU 2006 benchmarks, getting results that are consistent with the current state of the art [TGM11].

In the next chapter, we examine the body of related work. We will look at some of the issues related to Equality Saturation. We will also examine other approaches that have been taken, and how Peggy relates to them.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER grant CCF-0644306.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Logical Methods in Computer Science 2010*. The dissertation author was the secondary investigator and author of this paper.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. The dissertation author was the secondary investigator and author of this paper.

This chapter contains material taken from "Equality-Based Translation Validator for LLVM", by Michael Stepp, Ross Tate, and Sorin Lerner, which appears in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*. The dissertation author was the primary investigator and author of this paper.

Chapter 11

Related Work

Superoptimizers. Our approach of computing a set of programs and then choosing from this set is related to the approach taken by super-optimizers [Mas87, GK92, BA06, FHP92]. Superoptimizers strive to produce optimal code, rather than simply improve programs. Although super-optimizers can generate (near) optimal code, they have so far scaled only to small code sizes, mostly straight line code. Our approach, on the other hand, is meant as a general purpose paradigm that can optimize branches and loops.

Our approach was inspired by Denali [JNR02], a super-optimizer for finding near-optimal ways of computing a given basic block. Denali represents the computations performed in the basic block as an expression graph, and applies axioms to create an E-graph data structure representing the various ways of computing the values in the basic block. It then uses repeated calls to a SAT solver to find the best way of computing the basic block given the equalities stored in the E-graph. The biggest difference between our work and Denali is that our approach can perform intricate optimizations involving branches and loops. On the other hand, the Denali cost model is more precise than ours because it assigns costs to entire sequences of operations, and so it can take into account the effects of scheduling and register allocation.

Extensible Optimizers. There are many existing extensible optimization systems that allow the user to define additional optimizations or analyses. The Gen-

esis system [WS97b] by Whitfield and Soffa takes specifications written in the Gospel language and examines them to determine if they have any important interactions. The user must specify the relevant pre- and post- conditions for their optimization in the Gospel specification, and then the Genesis engine can use that data to determine a logical ordering in which the optimizations can run, without having any harmful interactions. Since our system makes no destructive updates, we can make a similar guarantee of no harmful interactions without doing any costly analysis step. However, the individual rewrite rules in our system tend to be smaller in scope and we instead rely on multiple interacting rules to get things done.

The Broadway compilation engine [GL05] performs library-level optimization by reading annotation files that describe the properties of library functions. The user can specify lattice types that the library manipulates, and then describe how each function affects elements of those lattices. Optimization occurs by describing circumstances when a particular function call may be replaced with a more efficient one. Our system can similarly be used to optimize library calls by specifying the usage patterns within axioms. In addition, Peggy is not limited to optimizations related to library calls, nor limited to optimizations on procedure calls in general.

Rhodium [LMRC05] is a system for defining propagation and transformation rules that can then be proved correct and semantics-preserving automatically. The user specifies a set of propagation and transformations rules by manipulating explicit dataflow facts in Rhodium, and then the engine proves that the transformations are semantics-preserving by discharging a simple proof obligation to a theorem prover for each rule. Our system does not perform the same formal proof of soundness of the transformations specified, but at the same time it is not limited by the scope of the theorem prover. In addition, we do not require the user to define complex dataflow propagation rules for each analysis.

Rewrite-Based Optimizers. Axioms or rewrite-rules have been used in many compilation systems, for example TAMPR [BHW97], ASF+SDF [vdBHKO02], the ML compilation system of Visser *et al.* [VBT98], and Stratego [BKVV08]. These

systems, however, perform transformations in sequence, with each axiom or rewrite rule destructively updating the IR. Typically, such compilers also provide a mechanism for controlling the application of rewrites through built-in or user-defined *strategies*. Our approach, in contrast, does not use strategies – we instead simultaneously explore all possible optimization orderings, while avoiding redundant work. Furthermore, even with no strategies, we can perform a variety of intricate optimizations.

Intermediate Representations. Our main contribution is an approach for structuring optimizers based on equality saturation. However, to make our approach effective, we have also designed the EPEG representation. There has been a long line of work on developing IRs that make analysis and optimizations easier to perform [CFR+89, AWZ88, TP95, Hav93, FOW87, WCES94, Cli95, SKR90, PBJ91]. The key distinguishing feature of EPEGs is that a single EPEG can represent many optimized versions of the input program, which allows us to use global profitability heuristics and to perform translation validation.

We now compare the PEG component of our IR with previous IRs. PEGs are related to SSA [CFR+89], gated SSA [TP95] and thinned-gated SSA [Hav93]. The μ function from gated SSA is similar to our θ function, and the η function is similar to our eval/pass pair. However, in all these variants of SSA, the SSA nodes are inserted *into* the CFG, whereas we do not keep the CFG around. The fact that PEGs are not tied to a CFG imposes fewer placement constraints on IR nodes, allowing us to implicitly restructure the CFG simply by manipulating the PEG. Furthermore, the conversion from any of the SSA representations back to imperative code is extremely simple since the CFG is already there. It suffices for each assignment $x := \phi(a, b)$ to simply insert the assignments $x := a$ and $x := b$ at the end of the two predecessors CFG basic blocks. The fact that our PEG representation is not tied to a CFG makes the conversion from PEGs back to a CFG-like representation much more challenging, since it requires reconstructing explicit control information.

The Program Dependence Graph [FOW87] (PDG) represents control information by grouping together operations that execute in the same control region.

The representation, however, is still statement-based. Also, even though the PDG makes many analyses and optimizations easier to implement, each one has to be developed independently. In our representation, analyses and optimizations fall out from a single unified reasoning mechanism.

The Program Dependence Web [OBM90] (PDW) combines the PDG with gated SSA. Our conversion algorithms have some similarities with the ones from the PDW. The PDW however still maintains explicit PDG control edges, whereas we do not have such explicit control edges, making converting back to a CFG-like structure more complex.

Dependence Flow Graphs [PBJ91] (DFGs) are a complete and executable representation of programs based on dependencies. However, DFGs employ a side-effecting storage model with an imperative *store* operation, whereas our representation is entirely functional, making equational reasoning more natural.

Like PEGs, the Value Dependence Graph [WCES94] (VDG) is a complete functional representation. VDGs use λ nodes (i.e. regular function abstraction) to represent loops, whereas we use specialized θ, eval and pass nodes. Using λs as a key component in an IR is problematic for the equality saturation process. In order to effectively reason about λs one must particularly be able to reason about substitution. While this is possible to do during equality saturation, it is not efficient. The reason is that equality saturation is also being done to the body of the λ expression (essentially optimizing the body of the loop in the case of VDGs), so when the substitution needs to be applied, it needs to be applied to all versions of the body and even all future versions of the body as more axioms are applied. Furthermore, one has to determine when to perform λ abstraction on an expression, that is to say, turn e into $(\lambda x.e_{body})(e_{arg})$, which essentially amounts to pulling e_{arg} out of e. Not only can it be challenging to determine when to perform this transformation, but one also has to take particular care to perform the transformation in a way that applies to *all* equivalent forms of e and e_{arg}.

The problem with λ expressions stems in fact from a more fundamental problem: λ expressions use *intermediate variables* (the parameters of the λs), and the level of indirection introduced by these intermediate variables adds reasoning

overhead. In particular, as was explained above for VDGs, the added level of indirection requires reasoning about substitution, which in the face of equality saturation is cumbersome and inefficient. An important property of PEGs is that they have no intermediate variables. The overhead of using intermediate variables is also why we chose to represent effects with an effect token rather than using the techniques from the functional languages community such as monads [Wad90a, Wad95, Wad98] or continuation-passing style [App91, Ken07, HD94, FSDF93, AJ97], both of which introduce indirection through intermediate variables. It is also why we used recursive expressions rather than using syntactic fixpoint operators.

Dataflow Languages. Our PEG intermediate representation is related to the broad area of dataflow languages [JHM04]. The most closely related is the Lucid programming language [AW77], in which variables are maps from iteration counts to possibly undefined values, as in our PEGs. Lucid's **first**/**next** operators are similar to our θ nodes, and Lucid's **as soon as** operator is similar to our eval/pass pair. However, Lucid and PEGs differ in their intended use and application. Lucid is a programming language designed to make formal proofs of correctness easier to do, whereas Peggy uses equivalences of PEG nodes to optimize code expressed in existing imperative languages.

Optimization Ordering. Many research projects have been aimed at mitigating the phase ordering problem, including automated assistance for exploring enabling and disabling properties of optimizations [WS90, WS97a], automated techniques for generating good sequences [CSD99, ACG+04, KDC02], manual techniques for combining analyses and optimizations [CC95], and automated techniques for the same purpose [LGC02]. However, we tackle the problem from a different perspective than previous approaches, in particular, by simultaneously exploring all possible sequences of optimizations, up to some bound. Our approach can do well even if every part of the input program requires a different ordering.

Translation Validation. Although previous approaches to translation validation have been explored [PSS98a, Nec00, ZPFG03], our approach has the advantage that it can perform translation validation by using the same technique as for program optimization, with no significant changes to the underlying Equality Sat-

uration engine. Another approach performs translation validation by attempting to put both programs into the same normal form, by applying several heuristically-ordered normalization passes [TGM11]. Our approach subsumes this one since we do similar transformations but with no need to worry about the order in which they are applied.

Execution Indices. Execution indices identify the state of progress of an execution [Dij68, XSZ08]. The call stack typically acts as the interprocedural portion, and the loop iteration counts in our semantics can act as the intraprocedural portion. As a result, one of the benefits of PEGs is that they make intraprocedural execution indices explicit.

Theorem Proving. Because our reasoning mechanism uses axioms, our work is related to the broad area of automated theorem proving. The theorem prover that most inspired our work is the Simplify theorem prover, with its E-graph data structure for representing equalities [NO79, NO80, DNS05]. Our EPEGs are in essence specialized E-graphs for reasoning about PEGs. The process of applying axioms to EPEGs uses a conceptually similar notion to the matching heuristic used in Simplify for instantiating universal quantifiers. However, the implementation of our axiom engine is different from Simplify's, in that we use the Rete algorithm.

SAT-based Optimizers. There has been a variety of work on using SAT solvers or Integer Linear Program solvers to compute optimal or near optimal solutions to various compiler optimization problems, for example data layout [BKK94], register allocation [GW96], and instruction scheduling [WLH00]. The work that is most closely related to ours in this area is the Denali super-optimizer [JNR02], a system for finding near-optimal ways of computing a given basic block. Denali represents the computation in the basic block as an expression graph, and applies axioms to create an E-graph representing the various ways of computing the values in the basic block. It then uses repeated calls to a SAT solver to find the best way of computing the basic block given the equalities stored in the E-graph. The biggest difference between our work and Denali is that we represent an entire procedure, including branches and loops. Also, our work uses an Integer Linear Program solver rather than iterated calls to a SAT solver. On the other hand, the Denali

cost model is more precise than ours because it assigns costs entire sequences of operations, and so it can take into account the effects of scheduling and register allocation.

Linear Types. Since our PEG representation is inherently functional, we represent operations with side effects by using effect tokens. This is related to the area of linear types [Wad90b] and linear logic [Gir98]. A linearly typed value is one that cannot be duplicated or destroyed by any operation. Any operation that takes in a value of linear type must produce one as output. Hence there will always be a single object of any linear type in existence at any given time. In Peggy, we use linear types to opaquely represent the global program state, which may be modified but cannot be copied or destroyed.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER grant CCF-0644306.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Logical Methods in Computer Science 2010*. The dissertation author was the secondary investigator and author of this paper.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. The dissertation author was the secondary investigator and author of this paper.

Chapter 12

Conclusion

This dissertation has described the Equality Saturation technique for performing optimization and translation validation of programs, and the Peggy framework which implements it. The major focus has been on the engineering challenges and applications of Equality Saturation, in order to make it into a viable technique for performing optimization and translation validation.

One of the first requirements for using Equality Saturation on real programs is the ability to represent real programs in a way that the engine can manipulate. To this end, I designed and implemented two distinct front-ends for Peggy; one for Java bytecode and one for LLVM bitcode. This required defining conversions from every Java/LLVM operator to a functional form that can be used in the PEG. Acting upon the compiled code rather than source allows Peggy to target a wide range of languages, since both Java and LLVM are compilation targets for many other languages.

The Equality Saturation engine acts upon the EPEG through a set of equality analyses. The more analyses it has, the more opportunities there are for optimization. To this end, I designed a system for describing and implementing equality analyses based on a simple XML-based text format. There are many levels of complexity for these analyses, and as such the input language has different kinds of specifications. The common case of a less complex analysis (or axiom) is much simpler to define than one that has arbitrary complexity, but both are possible.

Using the equality analysis definition language, I created a large corpus of analyses to be used within Peggy, for both the Java and LLVM contexts. These axioms encode many different aspects of the operators and values used within these languages, such as arithmetic facts, language-specific facts, pointer and aliasing facts, and even domain-specific facts. The inclusion of these analyses makes Peggy a much more potent system than it would be otherwise.

When converting from an imperative language to a functional language, there are some fundamental differences that must be overcome. Specifically, we must be careful in how we deal with imperative operations that cause side-effects. The imperative-to-functional conversion is fairly straightforward, but the reverse conversion can be quite tricky, since imperative code implicitly has more restrictions than functional code. To this end, I formalized the notion of what properties need to be maintained when converting to and from a PEG, and designed a system to maintain the correctness of the imperative code that is produced from a saturated EPEG.

Equality Saturation is simply a technique to discover equalities within programs. The engine by itself performs neither optimization nor translation validation, but is merely the mechanism by which these are achieved. I designed a system utilizing the main Equality Saturation engine which performs optimization of both Java and LLVM programs. The basic pipeline for optimization is: parse input program, convert program to PEG, add equality analyses to engine, insert PEG into engine, run Equality Saturation, choose optimal PEG from saturated EPEG, convert PEG back to imperative form, write imperative code back to disk. Each of these phases is a complex sub-problem with its own engineering challenges.

One of the most important sub-problems in the optimization pipeline is finding the optimal PEG in a saturated EPEG. This problem deserves special attention because its effectiveness speaks directly to the utility of the optimization performed. I formalized the PEG Selection problem, which describes the process of finding the optimal PEG within a saturated EPEG. I proved that this problem is NP-hard, and designed an algorithm to solve it by reducing the problem to Integer Linear Programming.

To show that Equality Saturation and Peggy in particular are effective at performing optimization, I performed an experimental evaluation of the optimizer. These experiments illustrated the fact that Peggy can perform many classical optimizations, and is particularly suited to domain-specific optimizations. Also, we found that Peggy exhibits instances of *emergent optimizations*, where equality analyses can combine in unexpected ways to perform optimizations that the user did not anticipate. I also showed that Equality Saturation can show significant performance improvements on a large real-world Java benchmark.

In addition to the optimization framework, I designed and implemented a framework for using Equality Saturation to perform translation validation of Java and LLVM programs. The basic pipeline for translation validation is: parse input programs, convert both programs into a single merged PEG, add equality analyses to engine, insert merged PEG into engine, run Equality Saturation, check that all pairs of program roots are equivalent in the EPEG. I also performed an experimental evaluation of the translation validator, for both Java and LLVM programs. In the Java setting, we validated 98% of the runs of the Soot optimizer on the SpecJVM benchmark. In the LLVM setting, we validated around 80% of the functions from the SPECCPU2006 integer benchmarks, after being optimized by the LLVM compiler. In both cases, we only evaluated intraprocedural optimizations.

Our implementation of the Peggy system is available for download at the following url: `http://cseweb.ucsd.edu/~mstepp/peggy/`

There are several avenues for future work which we have considered for Equality Saturation and the Peggy system, which we describe below.

Linear Types. One direction involves addressing our heap linearizing issues when reverting a PEG to a CFG. Our current approach relies on only allowing a particular subset of PEGs to be reverted, specifically those that are "properly linearized". This restricts the manner in which stateful operators can be used, so that they correspond more closely to the implicitly stateful operations of a CFG. Another approach to this problem is to reformulate our PEG representation itself to use *string diagrams* [BS09, Cur08]. Expressions are an excellent theory for non-linear values; string diagrams are a similar theory, but for linear values. A

string diagram is comprised of nodes with many inputs and many outputs along with strings which connect outputs of nodes to inputs of other nodes. By default these strings cannot be forked, capturing the linear quality of the values carried by the strings; however, strings for non-linear types are privileged with the ability to fork. In addition to using string diagrams to encode linearity in our PEGs, we could also re-express all of our axioms in terms of string diagrams, thus preserving the linear qualities of any strings involved. This prevents the saturation engine from producing PEGs which cannot be linearized. Also, string diagrams can be used to preserve well-formedness of PEGs.

Expressiveness of Rules. Our implementation of the Equality Saturation engine relies on the Rete pattern-matching algorithm [For82]. The α and β nodes in the engine's Rete currently only check structural properties of the EPEG, such as the fact the node A has n children, or that node A is the second child of node B. It also does basic node label equality tests. We could easily extend this to allow more general predicates in the α and β nodes of the Rete network. For instance, we could make an α predicate to check the result type of the operator for a particular node. Or, we could make a β predicate to compare the operators of two different nodes to see if they are binary operators that commute. These types of tests could be used to make more powerful and expressive axioms to be used in the engine. In addition, the Rete nodes are designed to check only local properties of the nodes in the EPEG. They cannot check properties that have to do with arbitrarily long paths through the EPEG. For instance, there is no way to write an axiom to test for "node A is a descendant of node B". Designing an efficient method for defining and checking such properties is an area for future work.

Interprocedural Optimization. Currently our optimization framework operates on one function at a time, and hence is purely intra-prodecural. However, it would useful to explore how Equality Saturation could be used to perform inter-procedural optimizations as well. This would have impact on the effectiveness of the optimizer, as well as the potency of the translation validator. As mentioned in Section 4.2, Equality Saturation is in fact well-suited to perform one very specific type of inter-procedural optimization, which is inlining. This is achieved by writing

an axiom that replaces a function call with the body of the called function. Identifying additional types of interprocedural optimizations that Peggy can perform is an area for future work.

Acknowledgements

Portions of this research were funded by the US National Science Foundation under NSF CAREER grant CCF-0644306.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Logical Methods in Computer Science 2010*. The dissertation author was the secondary investigator and author of this paper.

This chapter contains material taken from "Equality Saturation: a New Approach to Optimization", by Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner, which appears in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. The dissertation author was the secondary investigator and author of this paper.

Appendix A

Java/Soot/PEG conversion

In this section we describe in detail the conversion from Java bytecode to PEG operators. For each Java bytecode, we show how it maps to a Soot `Value` or `Stmt`, and then how the Soot objects are represented in terms of groups of PEG nodes. Some of the Java bytecode instructions have no direct analog to Soot or PEG operators. For example, since Soot is based on Jimple 3-address code, the instructions that only manipulate the stack such as `POP` and `SWAP` are translated away. Hence they are not represented in the PEG either.

For the Soot rows of the table, most of the bytecode instructions translate to instances of subclasses of the `Value` class. In addition to these values, there are `JimpleVariable` values, which represent local variables. Since Jimple is designed to resemble 3-address code, the expressions that take other values as parameters may only take constants or `JimpleVariable`'s as their parameters. Hence, rather than building up arbitrarily complex composite expressions, assignment statements are inserted of the form "`JAssignStmt`(`V` value, `RHS` value) stmt", where `V` is a `JimpleVariable` and `RHS` is an expression. In the table, whenever a bytecode instruction uses a parameter off the operand stack, Soot would have a `JimpleVariable` associated with that stack location, and would use that variable as the parameter to a new expression. Hence, in our notation we treat the stack operands as if they were Soot values. These are shown in the PEG rows in bold italics. All the node labels that appear in all capitals are instances of the SimpleJavaLabel class, which is used for labels that don't require any additional

information.

Table A.1: Translation between bytecode, Soot, and PEG nodes.

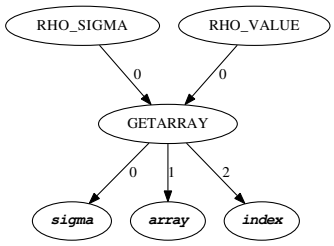
BC: `AALOAD`, `BALOAD`, `CALOAD`, `DALOAD`, `FALOAD`, `IALOAD`, `LALOAD`, `SALOAD`.

Desc: Load element from array.

Stack: $\ldots, array, index \implies \ldots, element$

Soot: `JArrayRef`(`array` value, `index` value) value

PEG:



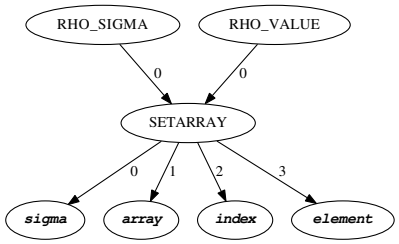
BC: `AASTORE`, `BASTORE`, `CASTORE`, `DASTORE`, `FASTORE`, `IASTORE`, `LASTORE`, `SASTORE`.

Desc: Store element into array.

Stack: $\ldots, array, index, element \implies \ldots$

Soot: `JAssignStmt`(LHS, RHS) stmt, where
LHS = `JArrayRef`(`array` value, `index` value) value
RHS = `element` value

PEG:



BC: `ACONST_NULL`.

Desc: Push a `null` reference onto the stack.

Stack: $\ldots \implies \ldots, null$

Soot: `NullConstant` value

PEG:

BC: `ALOAD` *index*,
`ILOAD` *index*,
`FLOAD` *index*,
`DLOAD` *index*,
`LLOAD` *index*.

Desc: Load from local variable slot `index`.

Stack: $\ldots \implies \ldots, val$

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

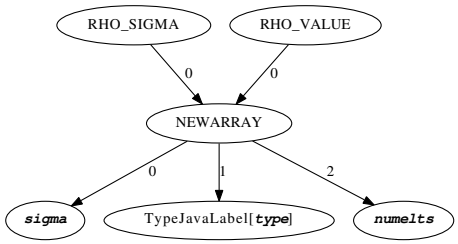
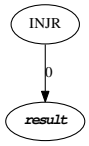
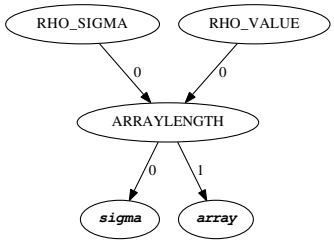
Soot:	Represented by a `JimpleVariable` value, whose value was assigned by a previous `JAssignStmt`.
PEG:	Represented by the sub-PEG for the Soot variable's value.
BC:	`ANEWARRAY` *type*, `NEWARRAY` *type*.
Desc:	Create a new array of object/primitive type given by `type`.
Stack:	$\ldots, numelts \implies \ldots, array$
Soot:	`JNewArray`(`type` type, `numelts` value) value
PEG:	
BC:	`ARETURN, IRETURN, FRETURN, LRETURN, DRETURN`.
Desc:	Return value and end method.
Stack:	$\ldots, result \implies [empty]$
Soot:	`JReturnStmt`(`result` value) stmt
PEG:	
BC:	`ARRAYLENGTH`.
Desc:	Get length of array.
Stack:	$\ldots, array \implies \ldots, length$
Soot:	`JLengthExpr`(`array` value) value
PEG:	
BC:	`ASTORE` *index*, `ISTORE` *index*, `DSTORE` *index*, `FSTORE` *index*, `LSTORE` *index*.

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

Desc: Store value into local variable.

Stack: $\ldots, val \implies \ldots$

Soot: `JAssignStmt`(`LHS` value, `val` value) stmt, where `LHS` is the `JimpleVariable` value for the given local variable slot.

PEG: Store is not explicitly represented, but will be referenced by the sub-PEG for `LHS`.

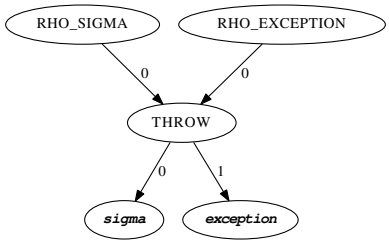
BC: `ATHROW`.

Desc: Throw an exception.

Stack: $\ldots, exception \implies exception$

Soot: `JThrowStmt`(`exception` value) stmt

PEG:



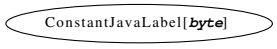
BC: `BIPUSH` *byte*.

Desc: Push constant byte onto stack.

Stack: $\ldots \implies \ldots, byte$

Soot: `IntConstant` value

PEG:



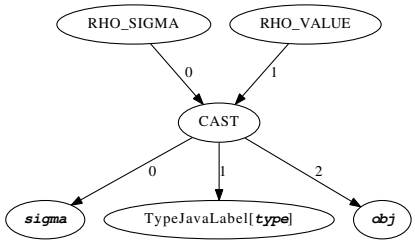
BC: `CHECKCAST` *type*.

Desc: Perform dynamic type cast.

Stack: $\ldots, obj \implies \ldots, obj$

Soot: `JCastExpr`(`obj` value, `type` type) value

PEG:



BC: `D2F, D2I, D2L`

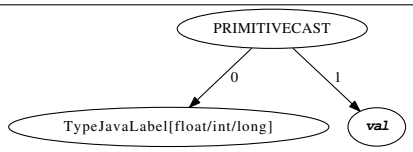
Desc: Convert double to float/int/long.

Stack: $\ldots, val \implies \ldots, cval$

Soot: `JCastExpr`(`val` value, `float/int/long` type) value

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

PEG:



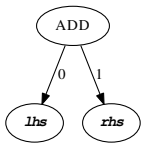
BC: `DADD, FADD, IADD, LADD`

Desc: Add two doubles/floats/ints/longs.

Stack: $\ldots, lhs, rhs \implies \ldots, sum$

Soot: `JAddExpr`(`lhs` value, `rhs` value) value

PEG:



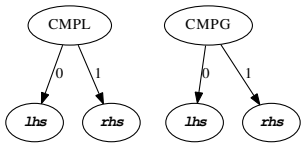
BC: `DCMPG, DCMPL, FCMPG, FCMPL`

Desc: Compare two doubles/floats.

Stack: $\ldots, lhs, rhs \implies \ldots, cmp$

Soot: `JCmpgExpr`(`lhs` value, `rhs` value) value, or
`JCmplExpr`(`lhs` value, `rhs` value) value

PEG:



BC: `DCONST_0, DCONST_1`

Desc: Push double constant 0.0 or 1.0.

Stack: $\ldots \implies \ldots, val$

Soot: `DoubleConstant` value

PEG:

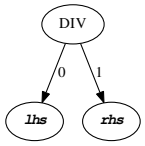
BC: `DDIV, FDIV, IDIV, LDIV`

Desc: Divide two doubles/floats/ints/longs.

Stack: $\ldots, lhs, rhs \implies \ldots, quotient$

Soot: `JDivExpr`(`lhs` value, `rhs` value) value

PEG:



BC: `DMUL, FMUL, IMUL, LMUL`

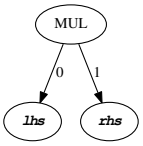
Desc: Multiply two doubles/floats/ints/longs.

Stack: $\ldots, lhs, rhs \implies \ldots, product$

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

Soot: `JMulExpr`(`lhs` value, `rhs` value) value

PEG:



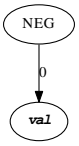
BC: `DNEG, FNEG, INEG, LNEG`

Desc: Negate a double/float/int/long.

Stack: $\ldots, val \implies \ldots, nval$

Soot: `JNegExpr`(`val` value) value

PEG:



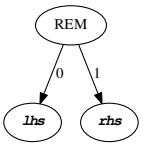
BC: `DREM, FREM, IREM, LREM`

Desc: Compute modulus of two doubles/floats/ints/longs.

Stack: $\ldots, lhs, rhs \implies \ldots, mod$

Soot: `JRemExpr`(`lhs` value, `rhs` value) value

PEG:



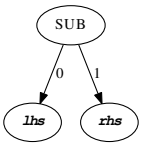
BC: `DSUB, FSUB, ISUB, LSUB`

Desc: Subtract two doubles/floats/ints/longs.

Stack: $\ldots, lhs, rhs \implies \ldots, diff$

Soot: `JSubExpr`(`lhs` value, `rhs` value) value

PEG:



BC: `DUP, DUP_X1, DUP_X2, DUP2, DUP2_X1, DUP2_X2`

Desc: Duplicate items on the stack.

Stack: $\ldots, v_1 \implies \ldots, v_1, v_1$

$\ldots, v_2, v_1 \implies \ldots, v_1, v_2, v_1$

$\ldots, v_3, v_2, v_1 \implies \ldots, v_1, v_3, v_2, v_1$

$\ldots, v_2, v_1 \implies \ldots, v_2, v_1, v_2, v_1$

$\ldots, v_3, v_2, v_1 \implies \ldots, v_2, v_1, v_3, v_2, v_1$

$\ldots, v_4, v_3, v_2, v_1 \implies \ldots, v_2, v_1, v_4, v_3, v_2, v_1$

Soot: Not represented in Soot.

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

PEG: Not represented in PEG.

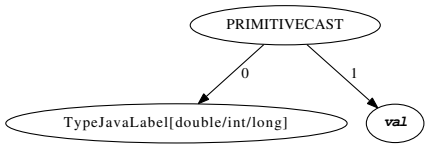
BC: `F2D, F2I, F2L`

Desc: Convert float to double/int/long.

Stack: $\ldots, val \implies \ldots, cval$

Soot: `JCastExpr`(`val` value, `double/int/long` type) value

PEG:



BC: `FCONST_0, FCONST_1`

Desc: Push float value 0.0f or 1.0f into stack.

Stack: $\ldots \implies \ldots, val$

Soot: `FloatConstant` value

PEG:

BC: `GETFIELD` *field*,
`GETSTATIC` *field*.

Desc: Get the value of a non-static/static field.

Stack: $\ldots, obj \implies \ldots, val$,
$\ldots \implies \ldots, val$

Soot: `JInstanceFieldRef`(`obj` value, `field` field) value,
`StaticFieldRef`(`field` field) value

PEG:

BC: `GOTO` *target*
`GOTO_W` *target*

Desc: Unconditional branch.

Stack: No change.

Soot: `JGotoStmt`(`target` stmt) stmt

PEG: Not represented in PEG.

BC: `I2B, I2C, I2D, I2F, I2L, I2S`

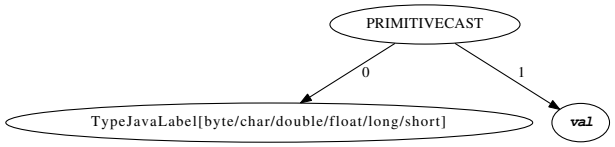
Desc: Convert int to byte/char/double/float/long/short.

Stack: $\ldots, val \implies \ldots, cval$

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

Soot: `JCastExpr`(`val` value, `byte/char/double/float/long/short` type) value

PEG:



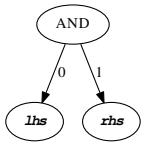
BC: `IAND, LAND`

Desc: Bitwise AND of two ints/longs.

Stack: $\ldots, lhs, rhs \implies \ldots, and$

Soot: `JAndExpr`(`lhs` value, `rhs` value) value

PEG:



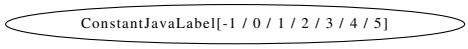
BC: `ICONST_M1, ICONST_0, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5`

Desc: Push the int value -1/0/1/2/3/4/5 into the stack.

Stack: $\ldots \implies \ldots, val$

Soot: `IntConstant` value

PEG:



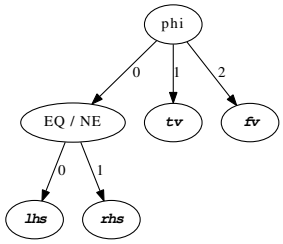
BC: `IF_ACMPEQ` *target*,
`IF_ACMPNE` *target*

Desc: Conditional branch, comparing two objects for reference equality.

Stack: $\ldots, lhs, rhs \implies \ldots$

Soot: `JIfStmt`(`JEqExpr`(`lhs` value, `rhs` value) value, `target` stmt) stmt
`JIfStmt`(`JNeExpr`(`lhs` value, `rhs` value) value, `target` stmt) stmt

PEG:



BC: `IF_ICMPEQ` *target*,
`IF_ICMPNE` *target*,
`IF_ICMPGE` *target*,
`IF_ICMPGT` *target*,
`IF_ICMPLE` *target*,
`IF_ICMPLT` *target*

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

Desc: Conditional branch, comparing two int values.

Stack: $\ldots, lhs, rhs \implies \ldots$

Soot: `JIfStmt`(`JEqExpr`(`lhs` value, `rhs` value) value, `target` stmt) stmt

`JIfStmt`(`JNeExpr`(`lhs` value, `rhs` value) value, `target` stmt) stmt

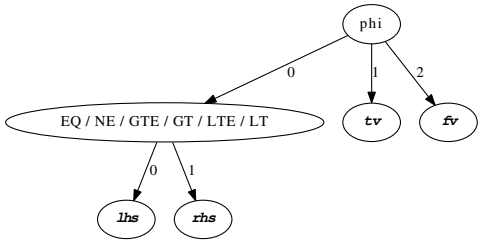
`JIfStmt`(`JGeExpr`(`lhs` value, `rhs` value) value, `target` stmt) stmt

`JIfStmt`(`JGtExpr`(`lhs` value, `rhs` value) value, `target` stmt) stmt

`JIfStmt`(`JLeExpr`(`lhs` value, `rhs` value) value, `target` stmt) stmt

`JIfStmt`(`JLtExpr`(`lhs` value, `rhs` value) value, `target` stmt) stmt

PEG:



BC: `IFEQ` *target*,

`IFNE` *target*,

`IFGE` *target*,

`IFGT` *target*,

`IFLE` *target*,

`IFLT` *target*

Desc: Conditional branch, comparing int value against 0.

Stack: $\ldots, val \implies \ldots$

Soot: `JIfStmt`(`JEqExpr`(`val` value, 0 value) value, `target` stmt) stmt,

`JIfStmt`(`JNeExpr`(`val` value, 0 value) value, `target` stmt) stmt,

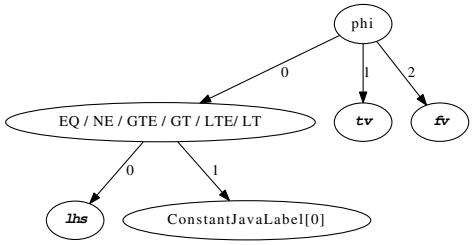
`JIfStmt`(`JGeExpr`(`val` value, 0 value) value, `target` stmt) stmt,

`JIfStmt`(`JGtExpr`(`val` value, 0 value) value, `target` stmt) stmt,

`JIfStmt`(`JLeExpr`(`val` value, 0 value) value, `target` stmt) stmt,

`JIfStmt`(`JLtExpr`(`val` value, 0 value) value, `target` stmt) stmt

PEG:



BC: `IFNULL` *target*,

`IFNONNULL` *target*

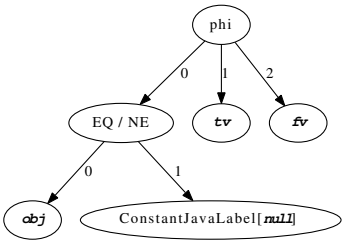
Desc: Conditional branch, compare object against null reference.

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

Stack: $\ldots, obj \implies \ldots$

Soot: `JIfStmt`(`JEqExpr`(`obj` value, `null` value) value, `target` stmt) stmt,
`JIfStmt`(`JNeExpr`(`obj` value, `null` value) value, `target` stmt) stmt

PEG:



BC: `IINC` *index, inc*

Desc: Increment integer local variable slot *index* by amount *inc*.

Stack: No change.

Soot: `JAssignStmt`(`var` value, `JAddExpr`(`var` value, `inc` value) value) stmt

PEG: Represented same as addition.

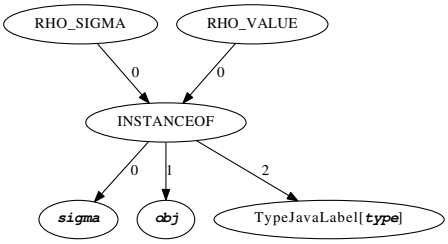
BC: `INSTANCEOF` *type*

Desc: Checks if an object is of the given type.

Stack: $\ldots, obj \implies \ldots, bool$

Soot: `JInstanceOfExpr`(`obj` value, `type` type) value

PEG:



BC: `INVOKEINTERFACE` *method*,
`INVOKEVIRTUAL` *method*,
`INVOKESPECIAL` *method*

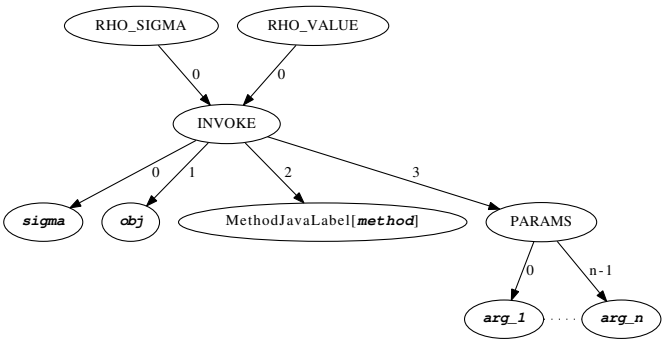
Desc: Non-static method invocation of interface/virtual/constructor method.

Stack: $\ldots, obj, arg_1, \ldots, arg_n \implies \ldots, result$ or
$\ldots, obj, arg_1, \ldots, arg_n \implies \ldots$ if void.

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

Soot: If void, a `JInvokeStmt`(`invoke` value) stmt,
otherwise a `JAssignStmt`(`var` value, `invoke` value) stmt, where
`var` is a `JimpleVariable` for the return value, and where `invoke` is one of:
`JInterfaceInvokeExpr`(`obj` value, `method` method, $\texttt{arg}_1$ value, ..., $\texttt{arg}_n$ value) value,
`JVirtualInvokeExpr`(`obj` value, `method` method, $\texttt{arg}_1$ value, ..., $\texttt{arg}_n$ value) value,
`JSpecialInvokeExpr`(`obj` value, `method` method, $\texttt{arg}_1$ value, ..., $\texttt{arg}_n$ value) value.

PEG:



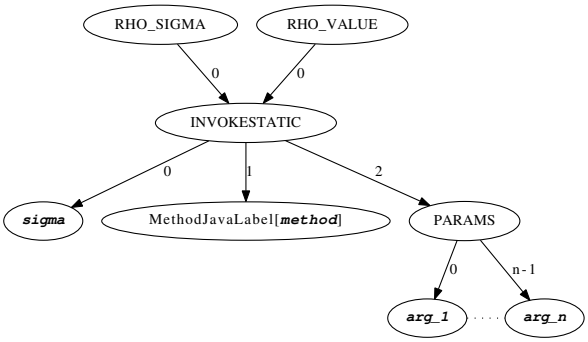
BC: `INVOKESTATIC` *method*

Desc: Static method invocation.

Stack: $\ldots, arg_1, \ldots, arg_n \implies \ldots, result$ or
$\ldots, arg_1, \ldots, arg_n \implies \ldots$ if void.

Soot: If void, a `JInvokeStmt`(`invoke` value) stmt,
otherwise a `JAssignStmt`(`var` value, `invoke` value) stmt, where
`var` is a `JimpleVariable` for the return value, and where `invoke` is:
`JStaticInvokeExpr`(`method` method, $\texttt{arg}_1$ value, ..., $\texttt{arg}_n$ value) value

PEG:



BC: `IOR, LOR`.

Desc: Bitwise OR of two ints/longs.

Stack: $\ldots, lhs, rhs \implies \ldots, or$

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

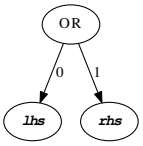
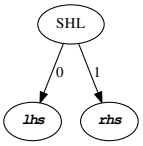
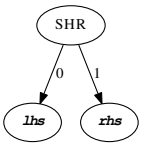
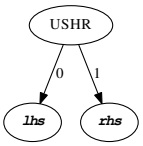
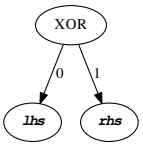
Soot:	`JOrExpr`(`lhs` value, `rhs` value) value
PEG:	
BC:	`ISHL, LSHL`.
Desc:	Left shift of int/long.
Stack:	$\ldots, lhs, rhs \implies \ldots, result$
Soot:	`JShlExpr`(`lhs` value, `rhs` value) value
PEG:	
BC:	`ISHR, LSHR`
Desc:	Arithmetic shift right of int/long.
Stack:	$\ldots, lhs, rhs, \implies \ldots, result$
Soot:	`JShrExpr`(`lhs` value, `rhs` value) value
PEG:	
BC:	`IUSHR, LUSHR`
Desc:	Logical shift right of int/long.
Stack:	$\ldots, lhs, rhs, \implies \ldots, result$
Soot:	`JUshrExpr`(`lhs` value, `rhs` value) value
PEG:	
BC:	`IXOR, LXOR`
Desc:	Bitwise XOR of two ints/longs.
Stack:	$\ldots, lhs, rhs \implies \ldots, xor$
Soot:	`JXorExpr`(`lhs` value, `rhs` value) value
PEG:	

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

BC: `JSR` *target*,
`JSR_W` *target*

Desc: Jump-to-subroutine: branch and push (opaque) address of following instruction on stack.

Stack: $\ldots \implies \ldots, addr$

Soot: These are restructured into GOTO's before translation to Soot.

PEG: Not represented.

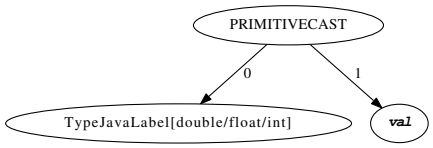
BC: `L2D, L2F, L2I`

Desc: Convert long to double/float/int.

Stack: $\ldots val \implies \ldots, cval$

Soot: `JCastExpr`(`val` value, `double/float/int` type) value

PEG:



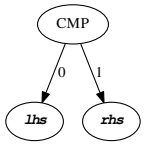
BC: `LCMP`

Desc: Compare two longs.

Stack: $\ldots, lhs, rhs \implies cmp$

Soot: `JCmpExpr`(`lhs` value, `rhs` value) value

PEG:



BC: `LCONST_0, LCONST_1`

Desc: Push the long value 0 or 1 onto the stack.

Stack: $\ldots \implies \ldots, val$

Soot: `LongConstant` value.

PEG:

BC: `LDC` *constant*,
`LDC_W` *constant*,
`LDC2_W` *constant*

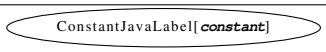
Desc: Load an int, float, double, long, or string constant from the constant pool onto the stack.

Stack: $\ldots \implies \ldots, constant$

Soot: `IntConstant` value, `FloatConstant` value, `DoubleConstant` value, `LongConstant` value, or `StringConstant` value.

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

PEG:



BC: `LOOKUPSWITCH` *default*, *n*, *match*$_1$, *target*$_1$, ..., *match*$_n$, *target*$_n$.

Desc: Switch statement, with default target *default*, and *n* match-target pairs.

Stack: ..., *key* $\Longrightarrow$ ...

Soot: `JLookupSwitchStmt`(`key` value, `default` stmt, `match`$_1$ value, `target`$_1$ value, ..., `match`$_n$ value, `target`$_n$ value) stmt

PEG: First converted to chain of if-stmts, then those are translated in normal way.

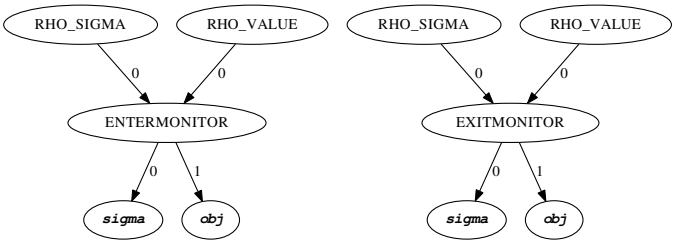
BC: `MONITORENTER, MONITOREXIT`

Desc: Acquire/release an object's monitor condition variable.

Stack: ..., *obj* $\Longrightarrow$ ...

Soot: `JEnterMonitorStmt`(`obj` value) stmt

PEG:



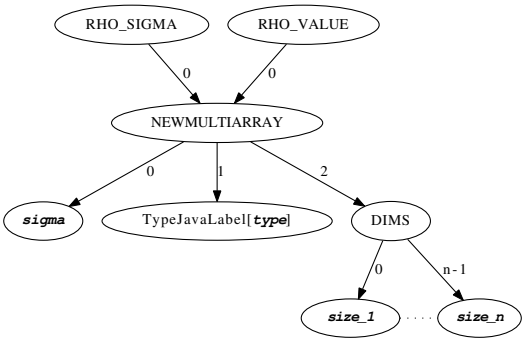
BC: `MULTIANEWARRAY` *type, n*

Desc: Create a new *n*-dimensional array of type *type*.

Stack: ..., *size*$_1$, ..., *size*$_n$ $\Longrightarrow$ ..., *array*

Soot: `JNewMultiArrayExpr`(`type` type, `size`$_1$ value, ..., `size`$_n$ value) value

PEG:



BC: `NEW` *type*

Desc: Create new object of type *type* (does not call constructor).

Stack: ... $\Longrightarrow$ ..., *obj*

Soot: `JNewExpr`(`type` type) value

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

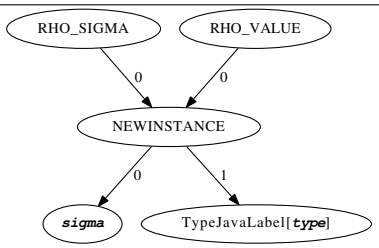
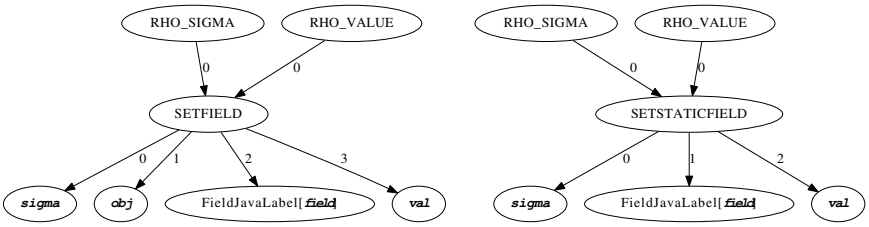
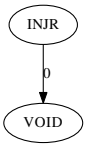
PEG:	
BC:	`NOP`
Desc:	Do nothing (NO-OP).
Stack:	No change.
Soot:	`JNopStmt` stmt.
PEG:	Not represented.
BC:	`POP, POP2`
Desc:	Pop 1 or 2 items off the stack.
Stack:	$\ldots, v_1 \implies \ldots,$ $\ldots, v_2, v_1 \implies \ldots$
Soot:	Not directly represented.
PEG:	Not directly represented.
BC:	`PUTFIELD` *field*, `PUTSTATIC` *field*.
Desc:	Set value of non-static/static field.
Stack:	$\ldots, obj, val \implies \ldots,$ $\ldots, val \implies \ldots$
Soot:	`JAssignStmt`(`JInstanceFieldRef`(`obj` value, `field` field) value, `val` value) stmt, `JAssignStmt`(`StaticFieldRef`(`field` field) value, `val` value) stmt
PEG:	
BC:	`RET` *index*
Desc:	Jump to opaque address value stored in local variable slot *index*.
Stack:	No change.
Soot:	JSR/RET instructions are converted to GOTO's before translation to Soot.
PEG:	Not directly represented.
BC:	`RETURN`

Table A.1: Translation between bytecode, Soot, and PEG nodes, Continued

Desc:	Void return from method.
Stack:	$\ldots \implies [empty]$
Soot:	`JReturnVoidStmt` stmt
PEG:	
BC:	`SIPUSH` *short*
Desc:	Push constant short onto stack.
Stack:	$\ldots \implies \ldots, short$
Soot:	`IntConstant` value
PEG:	
BC:	`SWAP`
Desc:	Exchange top 2 items on stack.
Stack:	$\ldots, v_2, v_1 \implies \ldots, v_1, v_2$
Soot:	Not directly represented.
PEG:	Not directly represented.
BC:	`TABLESWITCH` *default, lo, hi,* $target_{lo}$*, ...,* $target_{hi}$
Desc:	Switch statement, with default target *default* and targets to match *lo* through *hi*.
Stack:	$\ldots, key \implies \ldots$
Soot:	`JTableSwitchStmt(default` stmt, `lo` value, `hi` value, $\texttt{target}_{lo}$ stmt, ..., $\texttt{target}_{hi}$ stmt) stmt
PEG:	Converted to chain of if-stmts, then those are translated in normal way.

Appendix B

LLVM/PEG conversion

Table B.1 describes how the various LLVM values are encoded as PEG nodes. Each value in this table is represented as a single node with no children, so we do not draw the PEG, but simply describe the operator label for the single PEG node. Several of the labels are simply wrappers around the values themselves. For these examples, we give the value an explicit name and then refer to it in the PEG label.

In addition to the values listed in the table, there are values defined in terms of instructions acting on constant operands. These values are most easily expressed in terms of instructions, but since they operate on constant operands they may be computed at compile-time, rather than runtime. The only instructions that may be used for this purpose are `Cast`, `GetElementPtr`, `Select`, `ExtractElement`, `InsertElement`, `ShuffleVector`, `ICmp`, `FCmp`, and `Binop`. In the PEG, we replace these constant instructions with their non-constant equivalents, as shown in Table B.2.

Table B.1: Translation between LLVM values and PEG nodes.

LLVM:	`Alias`(*type*, *aliasee*, *linkage*, *visibility*)
Desc:	Alias value for another global value (one of global/function/alias), where *type* is a pointer type, *aliasee* is the value being aliased, *linkage* is a linkage id, and *visibility* is a visibility id.

Table B.1: Translation between LLVM values and PEG nodes, Continued

PEG:	`AliasLLVMLabel`[*name, type*], where *name* is the name of the alias taken from the module's value symbol table.
LLVM:	`Argument`(*parent*, *index*, *type*)
Desc:	Function formal parameter value, where *parent* is the parent function value, *index* is the 0-based formal parameter index, and *type* is the first-class type of the parameter.
PEG:	`ArgumentLLVMParameter`[*parent, index, type*]
LLVM:	`BlockAddress`(*func*, *index*)
Desc:	Address value for a particular basic block of a particular function, where *func* is the function value, and *index* is the index of the basic block.
PEG:	Cannot be represented.
LLVM:	$C =$ `ConstantExplicitArray`($type, elt_1, \ldots, elt_n$)
Desc:	An explicitly-defined constant array value, where *type* is an array type, and $elt_1, \ldots, elt_n$ are the constant element values.
PEG:	`ConstantLLVMLabel`[C]
LLVM:	$C =$ `ConstantNullArray`(*type*)
Desc:	A constant array value, where the elements are implicitly null (and not listed explicitly), where *type* is the array type.
PEG:	`ConstantLLVMLabel`[C]
LLVM:	$C =$ `InlineASM`(*hasSideEffects*, *type*, *ASM*, *constraints*)
Desc:	A value which represents a block of assembly code, which can be called as a function, where *hasSideEffects* is a boolean describing if the assembly code has side effects, *type* is a pointer to function type, *ASM* is a string of the actual assembly code, and *constraints* is a string spelling out some constraints on the assembly code.
PEG:	`ConstantLLVMLabel`[C]
LLVM:	$C =$ `NullPointer`(*type*)
Desc:	A constant null pointer value, where *type* is the pointer type.
PEG:	`ConstantLLVMLabel`[C]

Table B.1: Translation between LLVM values and PEG nodes, Continued

LLVM:	$C =$ `ConstantStructure`$(type, field_1, \ldots, field_n)$
Desc:	A constant structure value, where *type* is the struct type, and $field_1$, $\ldots$, $field_n$ are the constant field values.
PEG:	`ConstantLLVMLabel`$[C]$
LLVM:	$C =$ `ConstantExplicitVector`$(type, elt_1, \ldots, elt_n)$
Desc:	An explicitly-defined constant vector value, where *type* is the vector type, and elt_1, $\ldots$, elt_n are the constant element values.
PEG:	`ConstantLLVMLabel`$[C]$
LLVM:	$C =$ `ConstantNullVector`$(type)$
Desc:	A constant vector value, where the elements are implicitly null (and not listed explicitly), where *type* is the vector type.
PEG:	`ConstantLLVMLabel`$[C]$
LLVM:	`FloatingPoint`$(type, bits)$
Desc:	A literal floating point value, where *type* is the float type, and *bits* is the bit-string representation of the value.
PEG:	`ConstantLLVMLabel`$[C]$
LLVM:	`Function`$(type,\ cc,\ attrs,\ align,\ section,\ visibility,\ collector)$
Desc:	A function descriptor value, where *type* is a pointer to function type, *cc* is a calling convention id, *attrs* is a bitmask of parameter attributes, *align* is the function's byte alignment, *section* is a section id, *visibility* is a visibility id, and *collector* is a garbage collector id.
PEG:	`FunctionLLVMLabel`$[name,\ type]$, where *name* is the name of the function taken from the module's symbol table.
LLVM:	`Global`$(type,\ isConst,\ init,\ linkage,\ align,\ section,\ visibility,\ isTL)$
Desc:	A global variable value, where *type* is a pointer type, *isConst* is a boolean specifying whether the global is constant, *init* is the optional initial value, *linkage* is a linkage id, *align* is the global's byte alignment, *section* is a section id, *visibility* is a visibility id, and *isTL* is a boolean specifying whether the global is thread-local.

Table B.1: Translation between LLVM values and PEG nodes, Continued

PEG:	`GlobalLLVMLabel`[*name, type*], where *name* is the name of the global taken from the module's symbol table.
LLVM:	$C =$ `Integer`(*width*, *bits*)
Desc:	A literal integer value, where *width* is the bit-width of the 2's complement integer value, and *bits* is the bit-string of the value.
PEG:	`ConstantLLVMLabel`[C]
LLVM:	`Label`(*index*)
Desc:	A basic block label value, where *index* is the index of the basic block. This value can only appear inside a function body, so the parent function of the label is implicit.
PEG:	Cannot be represented.
LLVM:	$C =$ `MetadataNode`(*isFuncLocal*, $val_1, \ldots, val_n$)
Desc:	A metadata node value, where *isFuncLocal* is a boolean telling whether the metadata node contains any function-local values, and $val_1, \ldots, val_n$ are the values inside the node.
PEG:	`ConstantLLVMLabel`[C]
LLVM:	$C =$ `MetadataString`(*str*)
Desc:	A metadata string value, where *str* is the string contents of the metadata.
PEG:	`ConstantLLVMLabel`[C]
LLVM:	$C =$ `Undef`(*type*)
Desc:	An undefined value, where *type* is the type of the value.
PEG:	`ConstantLLVMLabel`[C]
LLVM:	`VirtualRegister`(*type*)
Desc:	A virtual register value where the results of instructions are stored, where *type* is the type of the value in the register.
PEG:	Not represented.

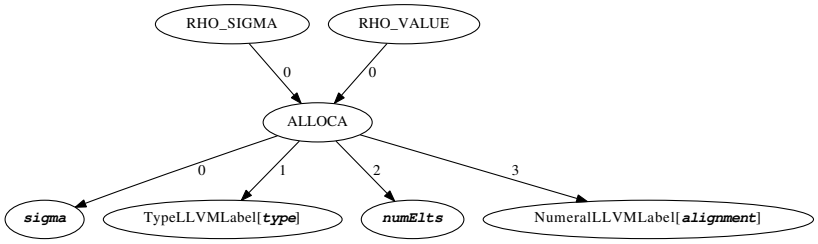
Table B.2 decribes how each LLVM instruction is encoded as a group of PEG nodes. In cases when the instruction takes a sub-value, these are represented in the PEG in bold, and implicitly mean that the sub-PEG for that value would appear in its place. All the node labels that appear in all capitals are instances of the SimpleLLVMLabel class, which is used for labels that don't require any additional information.

Table B.2: Translation between LLVM instructions and PEG nodes.

LLVM: `Alloca`(*type*, *numElts*, *alignment*)

Desc: Dynamically allocate stack space and return a pointer to it, where *type* is the pointee type, *numElts* is an `i32` value specifying the number of elements, and *alignment* is the desired byte alignment of the pointer.

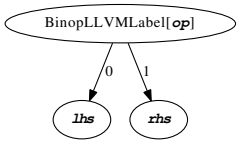
PEG:



LLVM: `Binop`(*op*, *lhs*, *rhs*)

Desc: Compute binary operation, where *op* describes which binop to compute, which is one of: {`add`, `sub`, `mul`, `unsigned-div`, `signed-div`, `float-div`, `unsigned-mod`, `signed-mod`, `float-mod`, `shift-left`, `arithmetic-shift-right`, `logical-shift-right`, `and`, `or`, `xor`}, *lhs* is the left-hand operand, and *rhs* is the right-hand operand.

PEG:

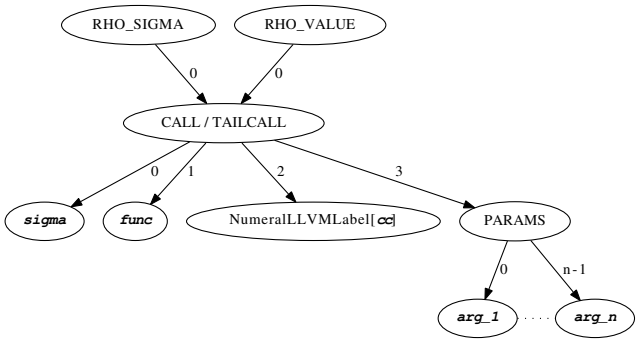


LLVM: `Call`(*isTailCall*, *cc*, *func*, *attrs*, arg_1, ..., arg_n)

Table B.2: Translation between LLVM instructions and PEG nodes, Continued

Desc: Call a function with no unwind handling, where *isTailCall* is a boolean specifying whether this is a tail call, *cc* is a calling convention id, *func* is the function value, *attrs* is a bitmask of callsite parameter attributes, and $arg_1, \ldots, arg_n$ are the actual parameters to the function.

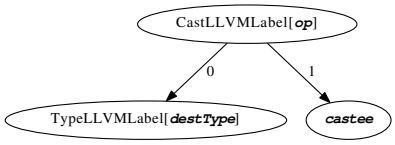
PEG:



LLVM: `Cast`(*op*, *destType*, *castee*)

Desc: Do a primitive type coercion, where *op* decribes the kind of coercion, which is one of {`truncate`, `zero-extend`, `sign-extend`, `float-to-unsigned-int`, `float-to-signed-int`, `unsigned-int-to-float`, `signed-int-to-float`, `float-truncate`, `float-extend`, `pointer-to-int`, `int-to-pointer`, `bitcast`}, *destType* is the new type of the coerced value, and *castee* is the value being coerced.

PEG:

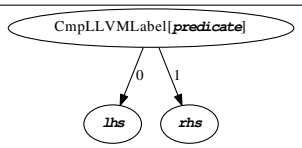


LLVM: `ICmp`(*predicate*, *lhs*, *rhs*)

Desc: Do an integer value comparison, where *predicate* describes the comparison being done and is one of {`eq`, `ne`, `unsigned-gt`, `unsigned-ge`, `unsigned-lt`, `unsigned-le`, `signed-gt`, `signed-ge`, `signed-lt`, `signed-le`}, *lhs* is the left-hand operand value, and *rhs* is the right-hand operand value.

Table B.2: Translation between LLVM instructions and PEG nodes, Continued

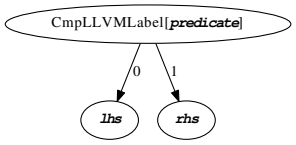
PEG:



LLVM: `FCmp`(*predicate*, *lhs*, *rhs*)

Desc: Do a floating-point value comparison, where *predicate* describes the comparison being done and is one of {`true`, `false`, `ordered`, `unordered`, `ordered-eq`, `ordered-ne`, `ordered-gt`, `ordered-ge`, `ordered-lt`, `ordered-le`, `unordered-eq`, `unordered-ne`, `unordered-gt`, `unordered-ge`, `unordered-lt`, `unordered-le`}, *lhs* is the left-hand operand value, and *rhs* is the right-hand operand value.

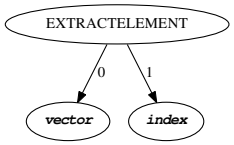
PEG:



LLVM: `ExtractElement`(*vector*, *index*)

Desc: Extract an element from a vector value, where *vector* is the vector value, and *index* is an `i32` index value.

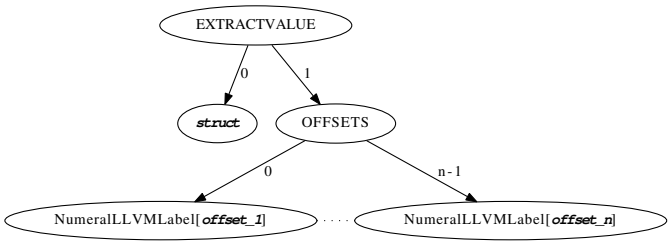
PEG:



LLVM: `ExtractValue`(*struct*, $offset_1$, ..., $offset_n$)

Desc: Extract an inner value from a structure value, where *struct* is the structure value, and $offset_1$, ..., $offset_n$ are offsets into the nested structure value.

PEG:

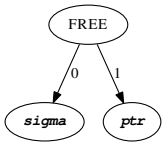


LLVM: `Free`(*ptr*)

Table B.2: Translation between LLVM instructions and PEG nodes, Continued

Desc: Frees a pointer value that was previously malloc'ed, where *ptr* is the pointer value.

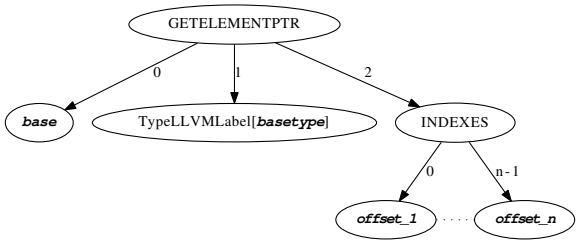
PEG:



LLVM: `GetElementPointer`(*basePtr*, $offset_1$, ..., $offset_n$)

Desc: Do pointer arithmetic to compute the location of a composite value's inner value, where *basePtr* is a pointer to the composite value, and $offset_1$, ..., $offset_n$ are the offsets into the nested composite values.

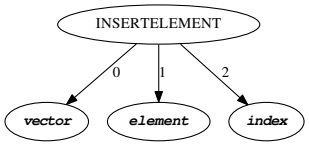
PEG:



LLVM: `InsertElement`(*vector*, *element*, *index*)

Desc: Create a new vector resulting from replacing an element of a given vector, where *vector* is the vector value, *element* is the replacement element value, and *index* is an `i32` value specifying the index to replace.

PEG:

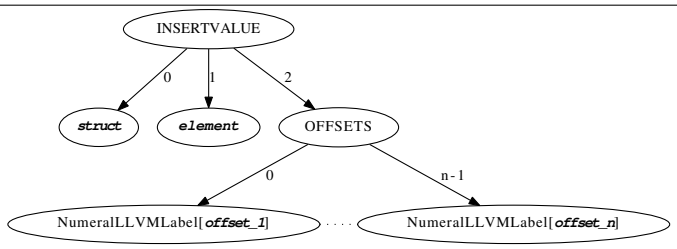


LLVM: `InsertValue`(*struct*, *element*, $offset_1$, ..., $offset_n$)

Desc: Create a new structure value by replacing one of its element values, where *struct* is the structure value, *element* is the new element value, and $offset_1$, ..., $offset_n$ are the offsets into the structure for the element to be replaced.

Table B.2: Translation between LLVM instructions and PEG nodes, Continued

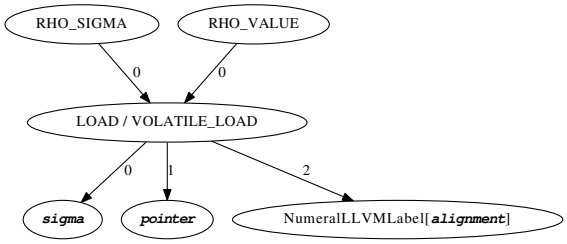
PEG:



LLVM: `Load`(*pointer*, *alignment*, *isVolatile*)

Desc: Load a value from a pointer, where *pointer* is the pointer value to load from, *alignment* is the byte-alignment of the pointer value, and *isVolatile* is a boolean specifying whether the load is volatile.

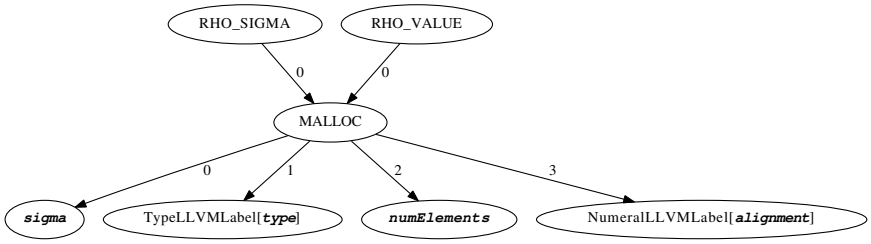
PEG:



LLVM: `Malloc`(*type*, *numElements*, *alignment*)

Desc: Dynamically allocate heap memory, where *type* is the pointee type, *numElements* is a `i32` value indicating the number of items to allocate, and *alignment* is the desired byte-alignment of the result pointer.

PEG:



LLVM: `Phi`(*type*, $\langle value_1, bb_1 \rangle$, ..., $\langle value_n, bb_n \rangle$)

Desc: Executable SSA ϕ-operator, where *type* is the type of the result, and $\langle value_1, bb_1 \rangle$, ..., $\langle value_n, bb_n \rangle$ are the value-block pairs that tell which value to use based on the dynamically preceding basic block.

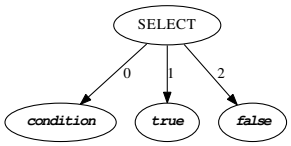
PEG: Not represented.

Table B.2: Translation between LLVM instructions and PEG nodes, Continued

LLVM: `Select`(*condition*, *true*, *false*)

Desc: Returns one of two values based on a boolean condition, where *condition* is the boolean condition value, *true* is the value to return if true, and *false* is the value to return if false.

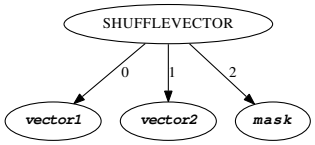
PEG:



LLVM: `ShuffleVector`(*vector*$_1$, *vector*$_2$, *mask*)

Desc: Create new vector based on rearranging elements from two others, where *vector*$_1$ is the first input vector, *vector*$_2$ is the second input vector, and *mask* is a "shuffle vector" that describes which elements of the input vectors to use in the result vector.

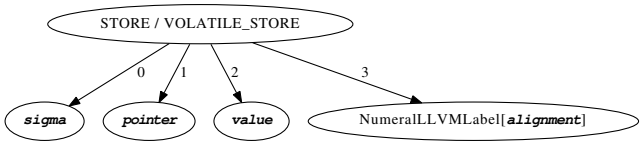
PEG:



LLVM: `Store`(*pointer*, *value*, *alignment*, *isVolatile*)

Desc: Store a value to a pointer, where *pointer* is the pointer value, *value* is the value being stored, *alignment* is the byte-alignment of the pointer, and *isVolatile* is a boolean specifying if the store is volatile.

PEG:



LLVM: `Branch`(*condition*, bb_{true}, bb_{false})

`Branch`(*target*)

Desc: Conditional or unconditional branch, where *condition* is the boolean condition value to test, bb_{true} is the block to visit if true, bb_{false} is the block to visit if false, and *target* is the sole branch target for an unconditional branch.

PEG: Conditional branch represented with ϕ nodes. Unconditional branch not represented.

Table B.2: Translation between LLVM instructions and PEG nodes, Continued

LLVM: `IndirectBranch`($blockAddress$, $block_1$, ..., $block_n$)

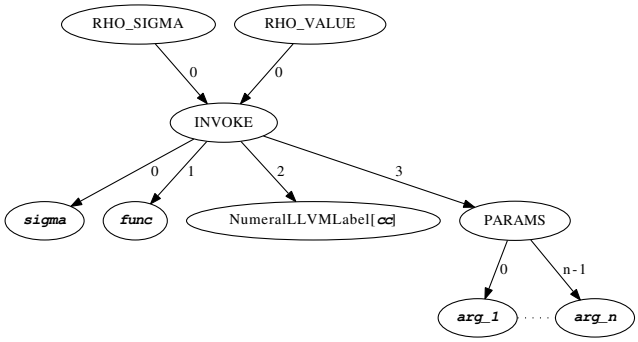
Desc: Indirect branch to an opaque basic block address, where *blockAddress* is the block address value, and $block_1$, ..., $block_n$ are all possible blocks where the branch might land.

PEG: Cannot be represented.

LLVM: `Invoke`(cc, $func$, $attrs$, bb_{return}, bb_{unwind}, arg_1, ..., arg_n)

Desc: Call a function with unwind handling, where cc is a calling convention id, $func$ is the function value to call, $attrs$ is a bitmask of callsite parameter attributes, bb_{return} is the basic block to jump to if the function returns normally, bb_{unwind} is the basic block to jump to if the function unwinds, and arg_1, ..., arg_n are the actual parameter values.

PEG:



LLVM: `Ret`(arg_1, ..., arg_n)

Desc: Cause the current function to terminate, returning 0 or more values, where arg_1, ..., arg_n are the values to return. If there are 0 values then it is a void return, if there is 1 value it is a normal non-void return, and if there are multiple values then they are implicitly returned in the fields of a new structure value.

PEG:

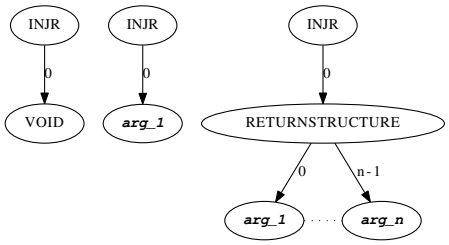


Table B.2: Translation between LLVM instructions and PEG nodes, Continued

LLVM: `Switch`(key, $bb_{default}$, $\langle value_1, bb_1 \rangle$, ..., $\langle value_n, bb_n \rangle$)

Desc: Perform a multi-way branch based on an integer key value, where key is the integer key value, $bb_{default}$ is the block to visit if no value/block pair matches, and $\langle value_1, bb_1 \rangle$, ..., $\langle value_n, bb_n \rangle$ are a set of value/block pairs that determine where to jump to based on the key's value.

PEG: Converted to chain of conditional branches.

LLVM: `Unreachable`()

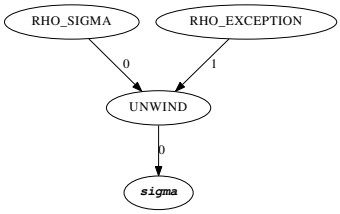
Desc: Unreachable instruction; execution should never reach this instruction and if it does the behavior is undefined.

PEG: Represented same as an `Unwind`.

LLVM: `Unwind`()

Desc: Unwind the call stack until the dynamically closest occurence of an `Invoke` instruction is found. This instruction is used for basic exception handling.

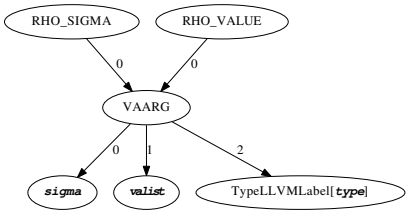
PEG:



LLVM: `Vaarg`($valist$, $type$)

Desc: Read a value from the variable-argument list of a function call, where $valist$ is a pointer to the variable-argument list, and $type$ is the type of the value to be read and returned.

PEG:



Appendix C

Axioms and Analyses Used in Peggy

In this appendix we list the axioms and analyses used within Peggy to perform Equality Saturation. This section is an elaboration of Chapter 5.

C.1 Arithmetic Axioms

Commutativity. These axioms encode the fact that certain operators commute.

```
<simpleTransform name='A+B = B+A'>
   add(!!A:*,@B:*) = add(@B,@A)
</simpleTransform>

<simpleTransform name='A*B = B*A'>
   mul(@A:*,@B:*) = mul(@B,@A)
</simpleTransform>

<simpleTransform name='A^B = B^A'>
   xor(@A:*,@B:*) = xor(@B,@A)
</simpleTransform>

<simpleTransform name='A|B = B|A'>
   or(@A:*,@B:*) = or(@B,@A)
</simpleTransform>

<simpleTransform name='A&B = B&A'>
   and(@A:*,@B:*) = and(@B,@A)
</simpleTransform>
```

Distributivity. These axioms encode the fact that some operators distribute over others.

```
<simpleTransform name='A*(B+C) = A*B + A*C'>
   mul(@A:*,add(@B:*,@C:*)) = add(mul(@A,@B),mul(@A,@C))
</simpleTransform>

<simpleTransform name='A*(B-C) = A*B - A*C'>
   mul(@A:*,sub(@B:*,@C:*)) = sub(mul(@A,@B),mul(@A,@C))
</simpleTransform>

<simpleTransform name='A&(B|C) = A&B | A&C'>
   and(@A:*,or(@B:*,@C:*)) = or(and(@A,@B),and(@A,@C))
</simpleTransform>

<simpleTransform name='A|(B&C) = (A|B) & (A|C)'>
   or(@A:*,and(@B:*,@C:*)) = and(or(@A,@B),or(@A,@C))
</simpleTransform>
```

Associativity. These axioms encode the fact that some operators are associative.

```
<simpleTransform name='A+(B+C) = (A+B)+C'>
   add(@A:*,add(@B:*,@C:*)) = add(add(@A,@B),@C)
</simpleTransform>

<simpleTransform name='A*(B*C) = (A*B)*C'>
   mul(@A:*,mul(@B:*,@C:*)) = mul(mul(@A,@B),@C)
</simpleTransform>

<simpleTransform name='A&(B&C) = (A&B)&C'>
   and(@A:*,and(@B:*,@C:*)) = and(and(@A,@B),@C)
</simpleTransform>

<simpleTransform name='A|(B|C) = (A|B)|C'>
   or(@A:*,or(@B:*,@C:*)) = or(or(@A,@B),@C)
</simpleTransform>

<simpleTransform name='A^(B^C) = (A^B)^C'>
   xor(@A:*,xor(@B:*,@C:*)) = xor(xor(@A,@B),@C)
</simpleTransform>
```

Relational Equivalence. These axioms encode the fact that many comparison operators are related.

```
<simpleTransform name='(A < B) == (B > A)'>
   lt(@A:*,@B:*) = gt(@B,@A)
</simpleTransform>
```

```
<simpleTransform name='(A > B) == (B < A)'>
   gt(@A:*,@B:*) = lt(@B,@A)
</simpleTransform>

<simpleTransform name='(A <= B) == (B >= A)'>
   lte(@A:*,@B:*) = gte(@B,@A)
</simpleTransform>

<simpleTransform name='(A >= B) == (B <= A)'>
   gte(@A:*,@B:*) = lte(@B,@A)
</simpleTransform>
```

Implications. These axioms encode implications about relational operators.

```
<simpleRule name='(A > B) => (A >= B)'>
  {gt(@A:*,@B:*)}
  ==>
  {gte(@A,@B)}
</simpleRule>

<simpleRule name='(A < B) => (A <= B)'>
  {lt(@A:*,@B:*)}
  ==>
  {lte(@A,@B)}
</simpleRule>

<simpleRule name='(A < A) => false'>
   @TOP:lt(@A:*,@A)
   ==>
   !{@TOP}!
</simpleRule>

<simpleRule name='(A > A) => false'>
   @TOP:gt(@A:*,@A)
   ==>
   !{@TOP}!
</simpleRule>

<simpleRule name='(A != A) => false'>
   @TOP:ne(@A:*,@A)
   ==>
   !{@TOP}!
</simpleRule>

<simpleRule name='(A == A) => true'>
   @TOP:eq(@A:*,@A)
   ==>
   {@TOP}
```

```
</simpleRule>

<simpleRule name='(A >= A) => true'>
   @TOP:gte(@A:*,@A)
   ==>
   {@TOP}
</simpleRule>

<simpleRule name='(A <= A) => true'>
   @TOP:lte(@A:*,@A)
   ==>
   {@TOP}
</simpleRule>
```

Miscellaneous. These axioms encode miscellaneous mathematical facts.

```
<simpleTransform name='(X*C1)<<C2 = X*(C1<<C2)'>
   shl(mul(@X:*,@C1:*),@C2:*) = mul(@X,shl(@C1,@C2))
</simpleTransform>

<simpleTransform name='(X<<C1)<<C2 = X<<(C1+C2)'>
   shl(shl(@X:*,@C1:*),@C2:*) = shl(@X,add(@C1,@C2))
</simpleTransform>

<simpleTransform name='(X>>C1)>>C2 = X>>(C1+C2)'>
   shr(shr(@X:*,@C1:*),@C2:*) = shr(@X,add(@C1,@C2))
</simpleTransform>

<simpleTransform name='(Y+(X>>>C)&CC)<<C = (X&(CC<<C))+(Y<<C)'>
   shl(add(@Y:*,and(ushr(@X:*,@C:*),@CC:*)),@C)
   = add(and(@X,shl(@CC,@C)),shl(@Y,@C))
</simpleTransform>

<simpleTransform name='(X<<C1)*C2 = X*(C2<<C1)'>
   mul(shl(@X:*,@C1:*),@C2:*) = mul(@X,shl(@C2,@C1))
</simpleTransform>

<simpleTransform name='(X/A)/B = X/(A*B)'>
   div(div(@X:*,@A:*),@B:*) = div(@X,mul(@A,@B))
</simpleTransform>

<simpleTransform name='A-(B+C) = (A-B)-C'>
   sub(@A:*,add(@B:*,@C:*)) = sub(sub(@A,@B),@C)
</simpleTransform>

<simpleTransform name='A-(B-C) = (A-B)+C'>
   sub(@A:*,sub(@B:*,@C:*)) = add(sub(@A,@B),@C)
</simpleTransform>
```

C.2 Nondomain Axioms

Boolean Axioms. These are axioms that encode facts about boolean logic.

```
<simpleRule name='A && false = false'>
   @TOP:%and(@A:*, @B:*)
   !{@B}!
   ==>
   !{@TOP}!
</simpleRule>

<simpleRule name='A && true = true'>
   @TOP:%and(@A:*, @B:*)
   {@B}
   ==>
   @TOP = @A
</simpleRule>

<simpleRule name='A || false = A'>
   @TOP:%or(@A:*, @B:*)
   !{@B}!
   ==>
   @TOP = @A
</simpleRule>

<simpleRule name='A || true = true'>
   @TOP:%or(@A:*, @B:*)
   {@B}
   ==>
   {@TOP}
</simpleRule>

<simpleTransform name='A && B = B && A'>
   %and(@A:*, @B:*) = %and(@B,@A)
</simpleTransform>

<simpleTransform name='A || B = B || A'>
   %or(@A:*, @B:*) = %or(@B,@A)
</simpleTransform>

<simpleRule name='negate(true) = false'>
   @TOP:%negate(@A:*)
   {@A}
   ==>
   !{@TOP}!
</simpleRule>
```

```
<simpleRule name='negate(false) = true'>
   @TOP:%negate(@A:*)
   !{@A}!
   ==>
   {@TOP}
</simpleRule>

<simpleTransform name='negate(negate(A)) = A'>
   %negate(%negate(@A:*)) = @A
</simpleTransform>
```

Phi Axioms. These axioms encode facts about the ϕ operator.

```
<simpleRule name='phi(true,B,C) = B'>
   @TOP:%phi(@A:*,@B:*,*)
   {@A}
   ==>
   @TOP = @B
</simpleRule>
// If the condition is true, the result is the first child.

<simpleRule name='phi(false,B,C) = C'>
   @TOP:%phi(@A:*,*,@C:*)
   !{@A}!
   ==>
   @TOP = @C
</simpleRule>
// If the condition is false, the result is the second child.

<simpleTransform name='phi(negate(A),B,C) = phi(A,C,B)'>
   %phi(%negate(@A:*),@B:*,@C:*) = %phi(@A,@C,@B)
</simpleTransform>
// Negating the condition is the same as swapping the other two children.

<simpleTransform name='phi(A,B,B) = B'>
   %phi(@A:*,@B:*,@B) = @B
</simpleTransform>
// If both cases are equal, that value is the result.

<simpleRule name='phi(A,true,false) = A'>
   @TOP:%phi(@A:*,@B:*,@C:*)
   {@B}
   !{@C}!
   ==>
   @TOP = @A
</simpleRule>
// If the result matches the condition in both cases, then just use the condition.
```

```
<simpleRule name='phi(A,false,true) = negate(A)'>
   @TOP:%phi(@A:*,@B:*,@C:*)
   !{@B}!
   {@C}
   ==>
   @TOP = %negate(@A)
</simpleRule>
// If the result is the negation of the condition in both cases, just use the negated condition.
```

Loop Axioms. These axioms encode facts about loop operators.

```
<simpleRule name='eval(theta(A,B),zero) = A, if A invariant'>
   @TOP:%eval-1(%theta-1(@A:*,@B:*),%zero)
   ~1{@A}
   ==>
   @TOP = @A
</simpleRule>
// Evaluating a loop-varying value at the 0-th iteration gives the initial value.

<simpleRule name='eval(A,B) = A, if A invariant'>
   @TOP:%eval-1(@A:*,*)
   ~1{@A}
   ==>
   @TOP = @A
</simpleRule>
// Evaluating a loop-invariant value at any loop iteration yields the same value.

<simpleTransform name='shift(theta(A,B)) = B'>
   %shift-1(%theta-1(*,@B:*)) = @B
</simpleTransform>
// Peel an iteration of a loop-varying value yields the recursive case.

<simpleRule name='shift(A) = A, if A invariant'>
   @TOP:%shift-1(@A:*)
   ~1{@A}
   ==>
   @TOP = @A
</simpleRule>
// Peeling a loop-invariant value has no effect.

<simpleRule name='theta(A,A) = A, if A invariant'>
   @TOP:%theta-1(@A:*,@A)
   ~1{@A}
   ==>
   @TOP = @A
</simpleRule>
// A loop-varying value that does not change equals its initial value.
```

```
<simpleRule name='pass(theta(true,*)) = 0'>
   @P:%pass-1(%theta-1(@A:*,*))
   {@A}
   ==>
   @P = %zero
</simpleRule>
// The first true iteration of a loop that starts with true is 0.

<simpleRule name='pass(true) = 0'>
   @P:%pass-1(@A:*)
   {@A}
   ==>
   @P = %zero
</simpleRule>
// The first true iteration of true itself is 0.
```

Loop Operator Factoring Axioms. These axioms encode how domain operators can distribute through loop operators.

```
<analysis name='op distribute through theta'>
   <trigger>
      <exists>
         <wild value='1' id='OP'>
            <theta index='1' id='THETA'>
               <variable id='A'/>
               <variable id='B'/>
            </theta>
            <variable id='C'/>
         </wild>
      </exists>
      <match>
         function match() {
            var op = $("OP");
            return op.getOp().isDomain();
         }
      </match>
   </trigger>
   <response>
      function build() {
         var opop = $("OP").getOp();
         var index = $("THETA").getOp().getLoopDepth();
         var A = copySource($("THETA"), 0);
         var B = copySource($("THETA"), 1);
         var C = copySource($("OP"), 1);
         // Build:  theta(op(A,eval(B,zero)),op(B,shift(C)))
         var result = futureNode(FlowValue.createTheta(index),
                        futureSource(opop,
                           A,
```

```
                        futureSource(FlowValue.createEval(index),
                            C,
                            futureSource(FlowValue.createZero()))),
                    futureSource(opop,
                        B,
                        futureSource(FlowValue.createShift(index), C)));
            makeEqual(result, $("OP"));
        }
    </response>
</analysis>
```

This axiom states that any domain operator can distribute through a θ node, but its children must be evaluated specially. For the initial value of the new θ node, the new children of the operator must be evaluated at the 0-th loop iteration. This is equivalent to passing the operator the initial values of all of its parameters, as would happen at the start of the loop. In the recursive case of the new θ node, the children of the operator must be peeled by one iteration.

The above axiom can generalize to the case when the operator has any number of parameters. Each parameter that is not a θ gets the eval/shift operators on top of its old value, as with C in the above axiom.

```
<analysis name='op distributes through eval'>
    <trigger>
        <exists>
            <wild value='1' id='OP'>
                <eval index='1' id='EVAL'>
                    <variable id='A'/>
                    <variable id='B'/>
                </theta>
                <variable id='C'/>
            </wild>
        </exists>
        <match>
            function match() {return $("OP").getOp().isDomain();}
        </match>
    </trigger>
    <response>
        function build() {
            var opop = $("OP").getOp();
            var index = $("EVAL").getOp().getLoopDepth();
            // Build:  eval(op(A,C),B)
            var result = futureNode(FlowValue.createEval(index),
```

```
                        futureSource(opop,
                           copySource($("EVAL"), 0),
                           copySource($("OP"), 1)),
                        copySource($("EVAL"), 1));
            makeEqual(result, $("OP"));
        }
    </response>
</analysis>
```

This axiom states that a domain operator can distribute through an eval node as long as the other children of the operator are invariant w.r.t. the same loop as the eval. This is important because when you move the operator under the eval, its children will now be evaluated according to a new loop iteration number. We can only be sure of the equivalence of these two expressions if the children are already invariant of the loop iteration they are evaluated at.

The above axiom can also generalize to the case when the operator has any number of parameters. As long as the children of the domain operator node are invariant then an analogous axiom can apply. We make special cases of this axiom for different arities.

C.3 Language-Specific Axioms

C.3.1 Java-specific Axioms

Field Access Axioms. These axioms encode facts about how Java accesses and modifies fields of classes.

```
<simpleTransform name='get(set(T,F,V),T,F) = V'>
    rho_value(getfield(
        rho_sigma(setfield(*,@T:*,@F:*,@V:*)),
        @T,
        @F))
    = @V
</simpleTransform>

<simpleRule name='get(set(Q,F,V),T,G) = get(T,F), if F!=G'>
    @V1:rho_value(getfield(
            rho_sigma(setfield(@SIGMA:*,@T:*,@F1:*,*)),
            @T,
            @F2:*))
```

```
   !{%equals(@F1,@F2)}!
   ==>
   @V1 = rho_value(getfield(@SIGMA,@T,@F2))
</simpleRule>

<simpleTransform name='set(set(T,F,X),T,F,Y) = set(T,F,Y)'>
   rho_sigma(setfield(
      rho_sigma(setfield(@SIGMA:*,@T:*,@F:*,@V1:*)),
      @T,
      @F,
      @V2:*))
   =
   rho_sigma(setfield(@SIGMA,@T,@F,@V2))
</simpleTransform>
```

The first axiom above states that if you set a field's value and then immediately fetch it, it will equal the value stored. The second axiom states that a get of a field after a set of a different field can be moved before the set. This relies on seeing that the field descriptors are explicitly non-equal. The third axiom states that a set of a field after another set to the same field makes the first set irrelevant.

```
<simpleTransform name='getstatic(setstatic(F,V),F)= V'>
   rho_value(getstaticfield(
      rho_sigma(setstaticfield(@SIGMA:*,@F:*,@V:*)),
      @F))
   = @V
</simpleTransform>

<simpleRule
   name='getstatic(setstatic(G,V),F) = getstatic(F),if F!=G'>
   @V1:rho_value(getstaticfield(
          rho_sigma(setstaticfield(@SIGMA:*,@F1:*,*)),
          @F2:*))
   !{%equals(@F1,@F2)}!
   ==>
   @V1 = rho_value(getstaticfield(@SIGMA,@F2))
</simpleRule>

<simpleTransform
   name='setstatic(setstatic(F,X),F,Y) = setstatic(F,Y)'>
   rho_sigma(setstaticfield(
      rho_sigma(setstaticfield(@SIGMA:*,@F:*,*)),
      @F,
      @V2:*))
```

```
   =
   rho_sigma(setstaticfield(@SIGMA,@F,@V2))
</simpleTransform>
```

These 3 axioms are analogous to the first 3, but for static fields.

Array Access Axioms. These axioms encode facts about how Java accesses and modifies arrays and array elements.

```
<simpleTransform name='get(set(A,I,V),A,I) = V'>
   rho_value(getarray(
      rho_sigma(setarray(@SIGMA:*,@A:*,@I:*,@V:*)),
      @A,
      @I))
   = @V
</simpleTransform>

<simpleRule name='get(set(A,J,V),A,I) == get(A,I), if I!=J'>
   @V1:rho_value(getarray(
          rho_sigma(setarray(@SIGMA:*,@A:*,@I:*,*)),
          @A,
          @J:*))
   !{%equals(@I,@J)}!
   ==>
   @V1 = rho_value(getarray(@SIGMA,@A,@J))
</simpleRule>

<simpleTransform name='set(set(A,I,V1),A,I,V2) = set(A,I,V2)'>
   rho_sigma(setarray(
      rho_sigma(setarray(@SIGMA:*,@A:*,@I:*,*)),
      @A,
      @I,
      @V2:*))
   = rho_sigma(setarray(@SIGMA,@A,@I,@V2))
</simpleTransform>
```

These 3 axioms are analogous to the field axioms above.

C.3.2 LLVM-specific Axioms

Pointer Axioms. These axioms have to do with pointers to memory in LLVM, and the operations that manipulate them.

```
<simpleTransform name='(store P (load P)) = no-op'>
   store(@S:*,@P:*,rho_value(load(@S,@P,@N:*)),@N) = @S
</simpleTransform>
// Storing the value that was just loaded from the same pointer is a no-op.

<simpleTransform name='load after store = stored value'>
   rho_value(load(store(*,@PTR:*,@V:*,@A:*),@PTR,@A)) = @V
</simpleTransform>
// Loading the value that was just stored in a pointer is equal to the stored value.

<simpleTransform name='store after store => kill bottom store'>
   store(store(@SIGMA:*,@PTR:*,*,*),@PTR,@V:*,@A:*) =
   store(@SIGMA,@PTR,@V,@A)
</simpleTransform>
// Storing to the same pointer twice in a row makes the first store useless.

<simpleTransform name='load is sigma-invariant'>
   rho_sigma(load(@S:*,*,*)) = @S
</simpleTransform>
// Loading from a pointer does not modify the heap.

<simpleTransform name='gep(B,0) = B'>
   getelementptr(@B:*,@T:*, indexes(int("32","0"))) = @B
</simpleTransform>

<simpleTransform name='gep(B,0) = B'>
   getelementptr(@B:*,@T:*, indexes(int("64","0"))) = @B
</simpleTransform>
// Pointer arithmetic with offset 0 gives the original pointer.
```

These last 2 axioms are actually constant axioms, but they relate to the GETELEMENTPTR operator, which performs opaque pointer arithmetic. Essentially, these axioms state that if your pointer arithmetic only adds 0 to the base pointer, then you are left with the original pointer value. Since the indexes to the GETELEMENTPTR operator can only be i32 or i64 values, we can simply make two versions of this axiom to cover both cases.

Vector Axioms. These axioms describe how LLVM handles vector values.

```
<simpleTransform>
   extractelement(insertelement(@V:*,@X:*,@I:*),@I) = @X
</simpleTransform>
// Extracting a vector element that was just inserted yields that new element.
```

```
<simpleRule>
   @TOP:extractelement(insertelement(@V:*,*,@I:*),@J:*)
   {%negate(%equals(@I,@J))}
   ==>
   @TOP = extractelement(@V,@J)
</simpleRule>
// Extracting an element after inserting is the same as extracting before inserting, if you know
    the vector indices are different.

<simpleTransform>
   insertelement(insertelement(@V:*,@X1:*,@I:*),@X2:*,@I)
   =
   insertelement(@V,@X2,@I)
</simpleTransform>
// Inserting the same element twice into a vector is the same as only inserting the later one.
```

Aliasing Axioms. These axioms encode facts about pointers and aliasing in LLVM, and make use of the `stackPointer` and `doesNotAlias` annotation labels defined in Section 5.4.2.

```
<simpleRule name='non-aliasing stores can swap'>
   @S:store(store(@SIGMA:*,@PTR1:*,@V1:*,@A1:*),
            @PTR2:*,
            @V2:*,
            @A2:*)
   {annotation("doesNotAlias", @PTR1, @PTR2)}
   ==>
   @S = store(store(@SIGMA,@PTR2,@V2,@A2),@PTR1,@V1,@A1)
</simpleRule>
// Adjacent stores to non-aliasing addresses can switch order.

<simpleRule name='sp(phi(A,B,C)) = sp(B) = sp(C)'>
   @A1:annotation("stackPointer", %phi(@A:*,@B:*,@C:*))
   ==>
   @A1 = annotation("stackPointer", @B)
   @A1 = annotation("stackPointer", @C)
</simpleRule>
// If a φ is known to be a stack pointer, then so must be both its result cases.

<simpleRule name='alloca is a stackPointer'>
   @TOP:rho_value(alloca(*,*,*,*))
   ==>
   {annotation("stackPointer",@TOP)}
</simpleRule>
// The alloca instruction always yields a stack pointer.
```

```
<simpleRule name='GEP preserves stackPointer-ness'>
   @GEP:getelementptr(@PTR:*,*,*)
   @A:annotation("stackPointer",@PTR)
   ==>
   @A = annotation("stackPointer",@GEP)
</simpleRule>
// If a pointer comes from the stack, then so does any arithmetic on top of it.

<simpleRule name='malloc is not a stackPointer'>
   @TOP:rho_value(malloc(*,*,*,*))
   ==>
   !{annotation("stackPointer",@TOP)}!
</simpleRule>
// The malloc instruction always yields a non-stack pointer.

<simpleRule name='sp and non-sp do not alias'>
   {annotation("stackPointer", @A:*)}
   !{annotation("stackPointer", @B:*)}!
   ==>
   {annotation("doesNotAlias", @A, @B)}
</simpleRule>
// If one pointer is from the stack and another is not, they cannot alias.

<simpleRule name='load may skip over non-aliasing alloca'>
   @L:rho_value(load(
      rho_sigma(@A:alloca(@SIGMA:*,*,*,*)),
      @PTR1:*,
      @ALIGN:*))
   {annotation("doesNotAlias", @PTR1, rho_value(@A))}
   ==>
   @L = rho_value(load(@SIGMA,@PTR1,@ALIGN))
</simpleRule>
// A load may skip over a non-aliasing stack allocation.

<simpleRule name='load may skip over a non-aliasing store'>
   @TOP:rho_value(load(
      store(@SIGMA:*,@PTR1:*,*,*),
      @PTR2:*,
      @ALIGN:*))
   {annotation("doesNotAlias",@PTR1,@PTR2)}
   ==>
   @TOP = rho_value(load(@SIGMA,@PTR2,@ALIGN))
</simpleRule>
// A load may skip over a store to a non-aliasing pointer.
```

In addition to the axioms presented above, we also use some analyses to help with the alias analysis. These are presented below.

```
<analysis name='bitcast to pointer preserves stackPointer'>
   <trigger>
      <exists>
         <cast id='B' type='bitcast'>
            <wild id='TYPE' value='1'/>
            <variable id='PTR'/>
         </cast>
         <annotation id='S' value='stackPointer'>
            <ref id='PTR'/>
         </annotation>
      </exists>
      <match>
         function match() {
            var d, t, typeop = $("TYPE").getOp();
            return typeop.isDomain() &&
                   (d=typeop.getDomain()).isType() &&
                   (t=d.getTypeSelf().getType()).isComposite() &&
                   t.getCompositeSelf().isPointer();
         }
      </match>
   </trigger>
   <response>
      function build() {
         var result = futureNode(
            new StringAnnotationLLVMLabel("stackPointer"),
            concreteSource($("B")));
         makeEqual(result, $("S"));
      }
   </response>
</analysis>
// A pointer-to-pointer bitcast preserves stack-pointer-ness.
```

This axiom states that if you have a pointer with a `stackPointer` annotation on it, then doing a pointer-to-pointer bitcast should result in a new pointer that has the same stack-pointer-ness as the original. We can express this by equating the `stackPointer` annotation on the original pointer with a new `stackPointer` annotation on the bitcast. Even if we have not yet determined that the original pointer was definitely a stack pointer or definitely not a stack pointer (by equating the annotation to true or false), we can still say that the stack-pointer-ness of the two pointers will be equal.

```
<analysis name='null is not a stackPointer'>
   <trigger>
      <exists>
         <wild id='null' value='1'/>
      </exists>
      <match>
        function match() {
           var d, n = $("null").getOp();
           return n.isDomain() &&
                  (d=n.getDomain()).isConstantValue() &&
                  d.getConstantValueSelf().getValue().isConstantNullPointer();
        }
      </match>
   </trigger>
   <response>
      function build() {
         var n = $("null");
         addArityProperty(n);
         addOpProperty(n);
         var result = node(
            new StringAnnotationLLVMLabel("stackPointer"),
            concOld(n));
         makeEqual(result, getFalse());
      }
   </response>
</analysis>
// The null pointer (of any type) is not a stack pointer.
```

This analysis simply marks all occurrences of the null pointer as not being a stack pointer. This must be done in an analysis instead of a simple axiom, because the type of the null pointer can be any valid pointer type. An axiom would have to spell out the pointer's type explicitly in order to match, but with an analysis we can just check that the given type is a pointer without naming it explicitly.

```
<analysis name='alloca/malloc does not alias null'>
   <trigger>
      <exists>
         <op id='RHO_VALUE' value='rho_value'>
            <wild id='ALLOC' value='1'>
               <variable/>
               <variable/>
               <variable/>
               <variable/>
            </wild>
```

```
            </op>
            <wild id='NULL' value='2'/>
        </exists>
    </trigger>
    <match>
        function match() {
            var ALLOCALABEL = LLVMOperator.ALLOCA;
            var MALLOCLABEL = LLVMOperator.MALLOC;
            var NULL = $("NULL").getOp();
            var ALLOC = $("ALLOC").getOp();
            if (NULL.isDomain() &&
                (d=NULL.getDomain()).isConstantValue() &&
                d.getConstantValueSelf().getValue().isConstantNullPointer() &&
                ALLOC.isDomain() &&
                (d=ALLOC.getDomain()).isSimple()) {
                var op = d.getSimpleSelf().getOperator();
                return op.equals(ALLOCALABEL) ||
                       op.equals(MALLOCLABEL);
            }
            return false;
        }
    </match>
    <response>
        function build() {
            var NULL = $("NULL");
            addArityProperty(NULL);
            addOpProperty(NULL);
            var result = futureNode(
               new StringAnnotationLLVMLabel("doesNotAlias"),
               concreteSource($("RHO_VALUE")),
               concreteSource(NULL));
            makeEqual(result, getTrue());
        }
    </response>
</analysis>
// The result of an alloca or malloc does not alias NULL.
```

This analysis establishes new instances of the `doesNotAlias` annotation, between any null pointer objects and any allocation operators (i.e. `alloca` or `malloc`). The allocation operators never return null, so we know that the result of these operators cannot alias the null pointer.

C.4 Domain-Specific Axioms

In this section we provide more of the domain-specific axioms used in the raytracer example described in Section 5.6.

```
<simpleRule name='V.getX() = V.mX'>
   @BS:rho_sigma(@GETX:invokevirtual(
           @SIGMA:*,
           @V:*,
           method("double CVector3D.getX()"),
           params()))
   @BV:rho_value(@GETX)
   ==>
   @BS = @SIGMA
   @BV = rho_value(getfield(@SIGMA,@V,field("double CVector3D.X")))
</simpleRule>
// Calling the getX method is the same as accessing the X field (similar axioms for Y, Z)

<simpleTransform name='cons(A,B,C).X = A'>
   rho_value(getfield(
      *,
      rho_value(invokestatic(
         *,
         method("CVector3D CVector3D.cons(double,double,double)"),
         params(@A:*,@B:*,@C:*))),
      field("double CVector3D.X")))
   =
   @A
</simpleTransform>
// The value of the X field of a newly-constructed vector is equal to the first input (similar
    axioms for Y, Z).

<simpleTransform name='cons(A,B,C).sub(cons(D,E,F)) = cons(A-D, B-E, C-F)'>
   invokevirtual(
      @SIGMA:*,
      rho_value(invokestatic(
         *,
         @CONS:method("CVector3D CVector3D.cons(double,double,double)"),
         params(@A:*,@B:*,@C:*))),
      method("CVector3D CVector3D.sub(CVector3D)"),
      params(
         rho_value(invokestatic(
            *,
            @CONS,
            params(@D:*,@E:*,@F:*)))))
   =
   invokestatic(
```

```
      @SIGMA,
      @CONS,
      params(sub(@A,@D), sub(@B,@E), sub(@C,@F)))
</simpleTransform>
// Subtracting two newly-constructed vectors is the same as constructing a new vector of the
    difference of the components (similar method for add).

<simpleTransform name='cons(A,B,C).scaled(D) = cons(A*D,B*D,C*D)'>
   invokevirtual(
      @SIGMA:*,
      rho_value(invokestatic(
         *,
         @CONS:method("CVector3D CVector3D.cons(double,double,double)"),
         params(@A:*,@B:*,@C:*))),
      method("CVector3D CVector3D.scaled(double)"),
      params(@D:*))
   =
   invokestatic(
      @SIGMA,
      @CONS,
      params(mul(@A,@D), mul(@B,@D), mul(@C,@D)))
</simpleTransform>
// Scaling a newly-constructed vector by a scalar is the same as making a new vector with scaled
    components.

<simpleRule name='cons(A,B,C).dot(cons(D,E,F)) = A*D + B*E + C*F'>
   @V:rho_value(@I:invokevirtual(
      @SIGMA:*,
      rho_value(invokestatic(
         *,
         @CONS:method("CVector3D CVector3D.cons(double,double,double)"),
         params(@A:*,@B:*,@C:*))),
      method("double CVector3D.dot(CVector3D)"),
      params(
         rho_value(invokestatic(
            *,
            @CONS,
            params(@D:*,@E:*,@F:*))))))
   @S:rho_sigma(@I)
   ==>
   @V = add(add(mul(@A,@D),mul(@B,@E)),mul(@C,@F))
   @S = @SIGMA
</simpleRule>
// Computing the dot product of two newly-constructed vectors is equal to operating on the
    components.
```

Appendix D

Axioms used in Figure 9.1

D.1 Axioms

In this section we describe the axioms used to produce the optimizations listed in Figure 9.1. We organize the axioms into two categories: general-purpose and domain-specific. The general-purpose axioms are useful enough to apply to a wide range of programming domains, while the domain-specific axioms give useful information about a particular domain.

The axioms provided below are not a complete list of the ones generally included in our engine during saturation. Instead, we highlight only those that were necessary to perform the optimizations in Figure 9.1.

D.1.1 General-purpose Axioms

The axioms presented here are usable in a wide range of programs. Hence, these axioms are included in all runs of Peggy.

Built-in EPEG ops. This group of axioms relates to the special PEG operators θ, eval, and ϕ. Many of these axioms describe properties that hold for any operation OP.

- if $T = \theta_i(\mathbf{A}, T)$ exists, then $T = \mathbf{A}$

  [A loop-varying value that always takes its previous value equals its initial value]

- if $\mathbf{A}$ is invariant w.r.t. i, then $\mathsf{eval}_i(\mathbf{A}, \mathbf{P}) = \mathbf{A}$
  [Loop-invariant operations have the same value regardless of the loop iteration]

- $\textsc{op}(\mathbf{A_1}, \ldots, \theta_j(\mathbf{B_i}, \mathbf{C_i}), \ldots, \mathbf{A_k}) =$
  $\theta_j(\textsc{op}(\mathsf{eval}_j(\mathbf{A_1}, 0), \ldots, \mathbf{B_i}, \ldots, \mathsf{eval}_j(\mathbf{A_k}, 0)),$
  $\textsc{op}(\mathsf{peel}_j(\mathbf{A_1}), \ldots, \mathbf{C_i}, \ldots, \mathsf{peel}_j(\mathbf{A_k})))$
  [Any domain operator can distribute through θ_j]

- $\phi(\mathbf{C}, \mathbf{A}, \mathbf{A}) = \mathbf{A}$
  [If both branches of a ϕ node are equal, then it is equal to them]

- $\phi(\mathbf{C}, \phi(\mathbf{C}, \mathbf{T_2}, \mathbf{F_2}), \mathbf{F_1}) = \phi(\mathbf{C}, \mathbf{T_2}, \mathbf{F_1})$
  [A ϕ node in a context where its condition is true is equal to its true case]

- $\textsc{op}(\mathbf{A_1}, \ldots, \phi(\mathbf{B}, \mathbf{C}, \mathbf{D}), \ldots, \mathbf{A_k}) = \phi(\mathbf{B}, \textsc{op}(\mathbf{A_1}, \ldots, \mathbf{C}, \ldots, \mathbf{A_k}),$
  $\textsc{op}(\mathbf{A_1}, \ldots, \mathbf{D}, \ldots, \mathbf{A_k}))$
  [All operators distribute through ϕ nodes]

- $\textsc{op}(\mathbf{A_1}, \ldots, \mathsf{eval}_j(\mathbf{A_i}, \mathbf{P}), \ldots, \mathbf{A_k}) = \mathsf{eval}_j(\textsc{op}(\mathbf{A_1}, \ldots, \mathbf{A_i}, \ldots, \mathbf{A_k}), \mathbf{P})$,
  when $\mathbf{A_1}, \ldots, \mathbf{A_{i-1}}, \mathbf{A_{i+1}}, \ldots, \mathbf{A_k}$ are invariant w.r.t. j
  [Any operator can distribute through eval_j]

Code patterns. These axioms are more elaborate and describe some complicated (yet still non-domain-specific) code patterns. These axioms are awkward to depict using our expression notation, so instead we present them in terms of before-and-after source code snippets.

- Unroll loop entirely:

```
x = B;                  ==      x = B;
for (i=0;i<D;i++)               if (D>=0) x += C*D;
   x += C;
```

[Adding C to a variable D times is the same as adding C*D (assuming $D \geq 0$)]

- Loop peeling:

```
A;                      ==      if (N>0) {
for (i=0;i<N;i++)                  B[i -> 0];
   B;                              for (i=1;i<N;i++)
                                      B;
                                } else {
                                   A;
                                }
```

[This axiom describes one specific type of loop peeling, where $B[i \rightarrow 0]$ means copying the body of B and replacing all uses of i with 0]

- Replace loop with constant:

```
for (i=0;i<N;i++){}     ==      x = N;
x = i;
```

[Incrementing N times starting at 0 produces N]

Basic Arithmetic. This group of axioms encodes arithmetic properties including facts about addition, multiplication, and inequalities. Once again, this is not the complete list of arithmetic axioms used in Peggy, just those that were relevant to the optimizations mentioned in Figure 9.1.

- $(\mathbf{A} * \mathbf{B}) + (\mathbf{A} * \mathbf{C}) = \mathbf{A} * (\mathbf{B} + \mathbf{C})$
- if $\mathbf{C} \neq 0$, then $(\mathbf{A}/\mathbf{C}) * \mathbf{C} = \mathbf{A}$
- $\mathbf{A} * \mathbf{B} = \mathbf{B} * \mathbf{A}$
- $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$
- $\mathbf{A} * 1 = \mathbf{A}$

- $\mathbf{A} + 0 = \mathbf{A}$
- $\mathbf{A} * 0 = 0$
- $\mathbf{A} - \mathbf{A} = 0$
- $\mathbf{A} \mod 8 = \mathbf{A}\&7$
- $\mathbf{A} + (-\mathbf{B}) = \mathbf{A} - \mathbf{B}$
- $-(-\mathbf{A}) = \mathbf{A}$
- $\mathbf{A} * 2 = \mathbf{A} << 1$
- $(\mathbf{A} + \mathbf{B}) - \mathbf{C} = \mathbf{A} + (\mathbf{B} - \mathbf{C})$
- $(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C})$
- if $\mathbf{A} \geq \mathbf{B}$ then $(\mathbf{A} + 1) > \mathbf{B}$
- if $\mathbf{A} \leq \mathbf{B}$ then $(\mathbf{A} - 1) < \mathbf{B}$
- $(\mathbf{A} > \mathbf{A}) =$ **false**
- $(\mathbf{A} \geq \mathbf{A}) = \mathit{true}$
- $\neg(\mathbf{A} > \mathbf{B}) \ = \ (\mathbf{A} \leq \mathbf{B})$
- $\neg(\mathbf{A} \leq \mathbf{B}) \ = \ (\mathbf{A} > \mathbf{B})$
- $(\mathbf{A} < \mathbf{B}) \ = \ (\mathbf{B} > \mathbf{A})$
- $(\mathbf{A} \leq \mathbf{B}) \ = \ (\mathbf{B} \geq \mathbf{A})$
- if $\mathbf{A} \geq \mathbf{B}$ and $\mathbf{C} \geq 0$ then $(\mathbf{A} * \mathbf{C}) \geq (\mathbf{B} * \mathbf{C})$

Java-specific. This group of axioms describes facts about Java-specific operations like reading from an array or field. Though they refer to Java operators explicitly, these axioms are still general-purpose within the scope of the Java programming language.

- $\textsc{getarray}(\textsc{setarray}(\mathbf{S}, \mathbf{A}, \mathbf{I}, \mathbf{V}), \mathbf{A}, \mathbf{I}) = \mathbf{V}$
  [Reading $\mathbf{A}[\mathbf{I}]$ after writing $\mathbf{A}[\mathbf{I}] \leftarrow \mathbf{V}$ yields $\mathbf{V}$]

- If $\mathbf{I} \neq \mathbf{J}$, $\textsc{getarray}(\textsc{setarray}(\mathbf{S}, \mathbf{A}, \mathbf{J}, \mathbf{V}), \mathbf{A}, \mathbf{I}) = \textsc{getarray}(\mathbf{S}, \mathbf{A}, \mathbf{I})$
  [Reading $\mathbf{A}[\mathbf{I}]$ after writing $\mathbf{A}[\mathbf{J}]$ (where $\mathbf{I} \neq \mathbf{J}$) is same as reading before writing]

- $\textsc{setarray}(\textsc{setarray}(\mathbf{S}, \mathbf{A}, \mathbf{I}, \mathbf{V_1}), \mathbf{A}, \mathbf{I}, \mathbf{V_2}) = \textsc{setarray}(\mathbf{S}, \mathbf{A}, \mathbf{I}, \mathbf{V_2})$
  [Writing $\mathbf{A}[\mathbf{I}] \leftarrow \mathbf{V_1}$ then $\mathbf{A}[\mathbf{I}] \leftarrow \mathbf{V_2}$ is the same as only writing $\mathbf{V_2}$]

- $\textsc{getfield}(\textsc{setfield}(\mathbf{S}, \mathbf{O}, \mathbf{F}, \mathbf{V}), \mathbf{O}, \mathbf{F}) = \mathbf{V}$
  [Reading $\mathbf{O}.\mathbf{F}$ after writing $\mathbf{O}.\mathbf{F} \leftarrow \mathbf{V}$ yields $\mathbf{V}$]

- If $\mathbf{F_1} \neq \mathbf{F_2}$,
  then $\textsc{getfield}(\textsc{setfield}(\mathbf{S}, \mathbf{O}, \mathbf{F_1}, \mathbf{V}), \mathbf{O}, \mathbf{F_2}) = \textsc{getfield}(\mathbf{S}, \mathbf{O}, \mathbf{F_2})$
  [Reading $\mathbf{A}[\mathbf{I}]$ after writing $\mathbf{A}[\mathbf{J}]$ (where $\mathbf{I} \neq \mathbf{J}$) is same as reading before writing]

- $\textsc{setfield}(\textsc{setfield}(\mathbf{S}, \mathbf{O}, \mathbf{F}, \mathbf{V_1}), \mathbf{O}, \mathbf{F}, \mathbf{V_2}) = \textsc{setfield}(\mathbf{S}, \mathbf{O}, \mathbf{F}, \mathbf{V_2})$
  [Writing $\mathbf{O}.\mathbf{F} \leftarrow \mathbf{V_1}$ then $\mathbf{O}.\mathbf{F} \leftarrow \mathbf{V_2}$ is the same as only writing $\mathbf{V_2}$]

D.1.2 Domain-specific

Each of these axioms provides useful information about a particular programming domain. These could be considered "application-specific" or "program-specific" axioms, and are only expected to apply to that particular application/program.

Inlining. Inlining in Peggy acts like one giant axiom application, equating the inputs of the inlined PEG with the actual parameters, and the outputs of the PEG with the outputs of the INVOKE operator.

- Inlining axiom:

```
x = pow(A,B);        ==      result = 1;
                             for (e = 0;e < B;e++)
                                result *= A;
                             x = result;
```

[A method call to pow is equal to its inlined body]

Sigma-invariance. It is very common for certain Java methods to have no effect on the heap. This fact is often useful, and can easily be encoded with axioms like the following.

- $\rho_\sigma(\text{INVOKE}(\mathbf{S}, \mathbf{L}, [\texttt{Object List.get()}], \mathbf{P})) = \mathbf{S}$
  [`List.get` is σ-invariant]

- $\rho_\sigma(\text{INVOKE}(\mathbf{S}, \mathbf{L}, [\texttt{int List.size()}], \mathbf{P})) = \mathbf{S}$
  [`List.size` is σ-invariant]

- $\rho_\sigma(\text{INVOKESTATIC}(\mathbf{S}, [\texttt{double Math.sqrt(double)}], \mathbf{P})) = \mathbf{S}$
  [`Math.sqrt` is σ-invariant]

Vector axioms. In our raytracer benchmark, there are many methods that deal with immutable 3D vectors. The following are some axioms that pertain to methods of the Vector class. These axioms when expressed in terms of PEG nodes are large and awkward, so we present them here in terms of before-and-after source code snippets.

- $\texttt{construct(}\mathbf{A}, \mathbf{B}, \mathbf{C}\texttt{).scaled(}\mathbf{D}\texttt{)} = \texttt{construct(}\mathbf{A} * \mathbf{D}, \mathbf{B} * \mathbf{D}, \mathbf{C} * \mathbf{D}\texttt{)}$
  [Vector (A, B, C) scaled by D equals vector $(A * D, B * D, C * D)$]

- $\mathbf{A}\texttt{.distance2(}\mathbf{B}\texttt{)} = \mathbf{A}\texttt{.difference(}\mathbf{B}\texttt{).length2()}$
  [The squared distance between A and B equals the squared length of vector $(A-B)$]

- $\mathbf{A}\texttt{.getX()} = \mathbf{A}\texttt{.mX}$
  $\mathbf{A}\texttt{.getY()} = \mathbf{A}\texttt{.mY}$
  $\mathbf{A}\texttt{.getZ()} = \mathbf{A}\texttt{.mZ}$
  [Calling the getter method is equal to accessing the field directly]

- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.mX} = \mathbf{A}$
  $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.mY} = \mathbf{B}$
  $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.mZ} = \mathbf{C}$
  [Reading field of vector (A, B, C) is equal to input]

- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.difference}(\texttt{construct}(\mathbf{D}, \mathbf{E}, \mathbf{F})) =$
  $\texttt{construct}(\mathbf{A} - \mathbf{D}, \mathbf{B} - \mathbf{E}, \mathbf{C} - \mathbf{F})$
  [The difference of vectors (A, B, C) and (D, E, F) equals $(A - D, B - E, C - F)$]

- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.dot}(\texttt{construct}(\mathbf{D}, \mathbf{E}, \mathbf{F})) = \mathbf{A} * \mathbf{D} + \mathbf{B} * \mathbf{E} + \mathbf{C} * \mathbf{F}$
  [The dot product of vectors (A, B, C) and (D, E, F) equals $A * D + B * E + C * F$]

- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.length2()} = \mathbf{A} * \mathbf{A} + \mathbf{B} * \mathbf{B} + \mathbf{C} * \mathbf{C}$
  [The squared length of vector (A, B, C) equals $A^2 + B^2 + C^2$]

- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.negative()} = \texttt{construct}(-\mathbf{A}, -\mathbf{B}, -\mathbf{C})$
  [The negation of vector (A, B, C) equals $(-A, -B, -C)$]

- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.scaled}(\mathbf{D}) = \texttt{construct}(\mathbf{A} * \mathbf{D}, \mathbf{B} * \mathbf{D}, \mathbf{C} * \mathbf{D})$
  [Scaling vector (A, B, C) by D equals $(A * D, B * D, C * D)$]

- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.sum}(\texttt{construct}(\mathbf{D}, \mathbf{E}, \mathbf{F})) =$
  $\texttt{construct}(\mathbf{A} + \mathbf{D}, \mathbf{B} + \mathbf{E}, \mathbf{C} + \mathbf{F})$
  [The sum of vectors (A, B, C) and (D, E, F) equals $(A + D, B + E, C + F)$]

- $\texttt{getZero()}\texttt{.mX} = \texttt{getZero()}\texttt{.mY} = \texttt{getZero()}\texttt{.mZ} = 0.0$
  [The components of the zero vector are 0]

Design patterns. These axioms encode scenarios that occur when programmers use particular coding styles that are common but inefficient.

- Axiom about integer wrapper object:

  $\mathbf{A}$`.plus(`$\mathbf{B}$`).getValue()` = $\mathbf{A}$`.getValue()` + $\mathbf{B}$`.getValue()`

  [Where `plus` adds two integer wrappers, and `getValue` gets wrapped value]

- Axiom about redundant method calls when using `java.util.List`:

```
Object o = ...                  ==   Object o = ...
List l = ...                         List l = ...
if (l.contains(o)) {                 int index = l.indexOf(o);
   int index = l.indexOf(o);         if (index >= 0) {
   ...                                  ...
}                                    }
```

  [Checking if a list contains an item then asking for its index is redundant]

Method Outlining. Method "outlining" is the opposite of method inlining; it is an attempt to replace a snippet of code with a procedure call that performs the same task. This type of optimization is useful when refactoring code to remove a common yet inefficient snippet of code, by replacing it with a more efficient library implementation.

- Body of selection sort replaced with `Arrays.sort(int[])`:

```
length = A.length;                  ==       Arrays.sort(A);
for (i=0;i<length;i++) {
   for (j=i+1;j<length;j++) {
      if (A[i] > A[j]) {
         temp = A[i];
         A[i] = A[j];
         A[j] = temp;
      }
   }
}
```

Specialized Redirect. This optimization is similar to Method Outlining, but instead of replacing a snippet of code with a procedure call, it replaces one procedure call with an equivalent yet more efficient one. This is usually in response to

some learned contextual information that allows the program to use a special-case implementation.

- if $I = \text{INVOKESTATIC}(\mathbf{S}, [\texttt{void sort(int[])}], \text{PARAMS}(\mathbf{A}))$ exists,
  then add equality $\mathit{isSorted}(\rho_\sigma(I), \mathbf{A}) = \mathit{true}$

  [If you call `sort` on an array A, then A is sorted in the subsequent heap]

- if $\mathit{isSorted}(\mathbf{S}, \mathbf{A}) = \mathit{true}$, then
  $\text{INVOKESTATIC}(\mathbf{S}, [\texttt{int linearSearch(int[],int)}], \text{PARAMS}(\mathbf{A}, \mathbf{B})) =$
  $\text{INVOKESTATIC}(\mathbf{S}, [\texttt{int binarySearch(int[],int)}], \text{PARAMS}(\mathbf{A}, \mathbf{B}))$
  [If array A is sorted, then a linear search equals a binary search]

Bibliography

[ACG+04] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *LCTES*, 2004.

[AJ97] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, 1997.

[App91] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1991.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[AWZ88] B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *POPL*, January 1988.

[BA06] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[BHW97] James M. Boyle, Terence J. Harmer, and Victor L. Winter. The TAMPR program transformation system: simplifying the development of numerical software. *Modern software tools for scientific computing*, pages 353–372, 1997.

[BKK94] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, August 1994.

[BKVV08] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[BS09] John C. Baez and Mike Stay. Physics, topology, logic and computation: A rosetta stone. `http://arxiv.org/abs/0903.0340`. Mar 2009.

[CC95] K. D. Cooper C. Click. Combining analyses, combining optimizations. *Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.

[CFR+89] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method for computing static single assignment form. In *POPL*, January 1989.

[Cli95] C. Click. Global code motion/global value numbering. In *PLDI*, June 1995.

[CSD99] K. D. Cooper, P. J. Schielke, and Subramanian D. Optimizing for reduced code space using genetic algorithms. In *LCTES*, 1999.

[Cur08] Pierre-Louis Curien. The joy of string diagrams. In *CSL '08: Proceedings of the 22nd international workshop on Computer Science Logic*, pages 15–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[DC94] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Conference on LISP and Functional Programming*, 1994.

[Dij68] E. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147 – 148, 1968.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG – fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[For82] Charles Forgy. Rete: A fast algorithm for the many pattern/-many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, New York, NY, USA, 1993. ACM.

[Gir98] Jean-Yves Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.

[GK92] Torbjorn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *PLDI*, 1992.

[GL05] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of IEEE*, 93(2), 2005.

[GW96] D. Goodwin and K. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Software Practice and Experience*, 26(8):929–965, August 1996.

[Hav93] P. Havlak. Construction of thinned gated single-assignment form. In *Workshop on Languages and Compilers for Parallel Computing*, 1993.

[HD94] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471, New York, NY, USA, 1994. ACM.

[JHM04] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[JL95] SL Peyton Jones and J Launchbury. State in haskell. In *Lisp and Symbolic Computation 8(4)*, 1995.

[JNR02] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *PLDI*, June 2002.

[KDC02] L. Torczon K. D. Cooper, D. Subramanian. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, pages 7–22, 2002.

[Ken07] Andrew Kennedy. Compiling with continuations, continued. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 177–190, New York, NY, USA, 2007. ACM.

[LGC02] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *POPL*, January 2002.

[llv] The LLVM compiler infrastructure. `http://llvm.org`.

[LMRC05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[Mas87] Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS*, 1987.

[Nec00] G. Necula. Translation validation for an optimizing compiler. In *PLDI*, June 2000.

[NO79] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[OBM90] K. Ottenstein, R. Ballance, and A. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, June 1990.

[PBJ91] K. Pengali, M. Beck, and R. Johson. Dependence flow graphs: an algebraic approach to program dependencies. In *POPL*, January 1991.

[PSS98a] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.

[PSS98b] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, 1998.

[SKR90] B. Steffen, J. Knoop, and O. Ruthing. The value flow graph: A program representation for optimal program transformations. In *European Symposium on Programming*, 1990.

[spe] The spec jvm98 benchmarks. `http://www.spec.org/jvm98/`.

[TGM11] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.

[TP95] P. Tu and D. Padua. Efficient building and placing of gating functions. In *PLDI*, June 1995.

[TSL10] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *POPL*, 2010.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL*, 2009.

[TSTL10] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Logical Methods in Computer Science*, December 2010.

[VBT98] E. Visser, Z. Benaissa, and A Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, 1998.

[vdBHKO02] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *Transactions on Programming Languages and Systems*, 24(4), 2002.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[Wad90a] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM.

[Wad90b] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[Wad98] Philip Wadler. The marriage of effects and monads. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 63–74, New York, NY, USA, 1998. ACM.

[WCES94] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *POPL*, 1994.

[WLH00] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *PLDI*, June 2000.

[WS90] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *PPOPP*, pages 137–146, March 1990.

[WS97a] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[WS97b] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[XSZ08] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI*, June 2008.

[ZPFG03] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.