UNIVERSITY OF CALIFORNIA, SAN DIEGO

Equality Saturation: Using Equational Reasoning to Optimize Imperative Functions

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

in

Computer Science

by

Ross Tate

2012

EPIGRAPH

Don’t worry; be happy.

Bobby McFerrin

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

There are innumerable people who helped me reach this point in life. I owe so much to my family, friends, mentors, and colleagues, past and present. I can only strive to give back to them as much as they have given to me.

With respect to this thesis, in addition to my committee members who have each offered me their own forms of guidance since my proposal, there are two people I need to thank in particular. First, my coauthor many times over, Michael Stepp, brought this project down to earth, solving the tangle of problems with Java and LLVM that I only dealt with in the abstract. Second, my advisor, Sorin Lerner, sowed the seeds for this project, bravely transfering his years of experience to an ignorant and unorthodox barely graduate student. Over the years, he has aided me more than I realize, despite me being a terribly difficult pupil to put up with for so long. As you can see below, most of this thesis exists due to their hard work.

Chapters 1, 2, 3, 5, 6, 10, 11, 12, and 19 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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

Chapters 1, 13, 14, 15, 16, 17, 18, and 19 contain material taken from “Generating Compiler Optimization from Proofs”, by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

Chapters 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 19, and Appendix A 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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 12 contain 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*, 2011. The dissertation author was the secondary investigator and author of this paper.

VITA

2004	Internship CustomFlix San Luis Obispo, California
2005	Internship Treyarch Santa Monica, California
2006	Bachelors of Science Mathematics and Computer Science California Polyntechnic State University, San Luis Obispo
2007	Internship Treyarch Santa Monica, California
2008	Internship Microsoft Research Redmond, Washington
2009	Microsoft Research Fellow
2009	Master of Science Computer Science University of California, San Diego
2009	Internship Microsoft Research Redmond, Washington
2012	Doctor of Philosophy Computer Science University of California, San Diego

PUBLICATIONS

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. “Equality Saturation: a New Approach to Optimization”. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2009

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*, 2010

Ross Tate, Juan Chen, and Chris Hawblitzel. “Inferable Object-Oriented Typed Assembly Language”. In *Proceedings of the 2011 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2010

Ross Tate, Alan Leung, and Sorin Lerner. “Taming Wildcards in Java’s Type System”. In *Proceedings of the 2011 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2011.

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

Michael Stepp, Ross Tate, and Sorin Lerner. “Equality-Based Translation Validator for LLVM”. In *Proceedings of the 23rd international conference on Computer Aided Verification (CAV)*, 2011

ABSTRACT OF THE DISSERTATION

Equality Saturation: Using Equational Reasoning to Optimize Imperative Functions

by

Ross Tate

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Sorin Lerner, Chair

Traditional optimizers have viewed imperative functions as a sequence or graph of commands. Some of these commands interact with the heap, throw exceptions, or interact with the outside world. Other commands simply read and modify local variables. In this dissertation, I take a different perspective on imperative functions. I view them as mathematical expressions, albeit with some quirks. These mathematical expressions have no need for local variables, making the flow of computation direct and so easier to reason about and manipulate. I will show how this enables equational techniques for program optimization, translation validation, and optimizer extensibility.

Chapter 1

Introduction

Compilers are one of the core tools that developers rely upon, and as a result they are expected to be reliable and provide good performance. Developing good compilers however is difficult, and the optimization phase of the compiler is one of the trickiest to develop. Compiler writers must develop complex transformations that are correct, do not have unexpected interactions, and provide good performance, a task that is made all the more difficult given the number of possible transformations and their possible interactions. This task is error prone and tedious, often requiring multiple iterations to get the rules correct.

In this thesis I provide a variety of technologies for improving compilers:

- a technique for inferring optimizations from basic properties of the language
- a technique for verifying the correctness of a given run of the optimizer
- a technique for learning an optimization from a before-and-after program pair

These technologies are centered around an intermediate representation we developed for algebraically representing imperative functions, which we call Program Expression Graphs (PEGs). This algebraic representation enables additive equational reasoning, specifically a process we call equality saturation, at a higher level of control flow, which automatically extends the above techniques to optimize loops and branches as well as straight-line code.

1.1 Inferring Optimizations

In a traditional compilation system, optimizations are applied sequentially, with each optimization taking as input the program produced by the previous one. This traditional approach to compilation has several well known drawbacks. One of these drawbacks is that the order in which optimizations are run 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 impact of future transformations.

In this thesis I present an 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 impact 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 superoptimizers, as Chapter 19 on related work will point out, superoptimizers 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, tail-recursion elimination, 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, when the optimization 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 an 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 we call equality 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 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.

Optimization Order is Irrelevant Our approach 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.

Global Profitability Heuristics Our approach 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 impact of future optimizations.

Our approach, on the other hand, 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, 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 a pseudo-boolean solver (a form of integer-linear-program solver). In particular, after our IR has been saturated with equalities, we use a pseudo-boolean solver and a static cost model for every node to pick the lowest-cost program that computes the same result as the original program.

1.2 Translation Validation

While compiler optimizers may be difficult to make, they have long played a crucial role in the software ecosystem. They allow programmers to express their programs at higher levels of abstraction without paying the performance cost, they allow just-in-time compilers to provide good performance for languages like Java and Javascript, and they allow programmers to benefit from architecture-specific optimizations without knowing the details of the architecture. At the same time, however, programmers also expect compiler optimizations to be correct, meaning they preserve the behavior of the programs they transform. Unfortunately, this seemingly simple requirement is hard to ensure in practice. Each optimization contains complex rules as to when it should apply, and these rules often have many subtle corner cases that need to handled correctly. Furthermore, a typical compiler contains not one, but many optimizations, and these optimizations can interact in unexpected ways. This results in a blowup in the number of individual corner cases and in the interaction of corner cases. This blowup not only makes it hard for optimization writers to reason about all the possible cases, it also makes it hard to write comprehensive test suites with good coverage.

One approach to improving the reliability of compiler optimizations is a technique called *translation validation* [64]. After each run of the optimizer, a separate tool called a translation validator tries to show that the optimized program is equivalent to the corresponding original program. Therefore, a translation validator is just a tool that tries to show two programs are equivalent [69]. Interestingly, equality saturation can be easily adapted to translation validation. We simply provide two programs, saturate with equalities, and then see if the two programs are found to be equivalent. Because our intermediate representation can represent branches and loops, our translation validator can handle a wide variety of intraprocedural optimizations. Our translation validator can also output a proof that the two programs are equivalent, which turns out to be useful for learning optimizations from examples.

1.3 Learning Optimizations

Implementing optimizations often involve languages and interfaces that are not familiar to the typical programmer: either a language for rewrite rules that the programmer needs to become familiar with, or an interface in the compiler that the programmer needs to learn. These difficulties raise the barrier to entry for non-compiler-experts who wish to customize their compiler.

In this thesis I present a paradigm for expressing compiler optimizations that drastically reduces the burden on the programmer. To implement an optimization in our approach, all the programmer needs to do is provide a simple concrete example of what the optimization looks like. Such an *optimization instance* consists of some original program and the corresponding transformed program. From this concrete optimization instance, our system abstracts away inessential details and learns a general optimization rule that can be applied more broadly than on the given concrete examples and yet is still guaranteed to be correct. In other words, our system generalizes optimization *instances* into correct optimization *rules*.

Our approach reduces the burden on the programmer who wishes to implement optimizations because optimization instances are much easier to develop than optimization rules. There is no more need for the programmer to learn a new language or interface for expressing transformations. Instead, the programmer can simply write down examples of the optimizations that they want to see happen, and our system can generate optimization rules from these examples. The simplicity of this paradigm would even enable end-user programmers, who are not compiler experts, to extend the compiler using what is most familiar to them, namely the source language they program in. In particular, if an end-user programmer sees that a program is not compiled as they wish, they can simply write down the desired transformed program, and from this concrete instance our approach can learn a general optimization rule to incorporate into the compiler. Furthermore, optimization instances can also be found by simply running a set of existing benchmarks through some existing compiler, thus allowing a programmer to harvest optimization capabilities from several existing compilers.

The key technical challenge in generalizing an optimization instance into an optimization rule is that we need to determine which parts of the programs in the optimization instance mattered and how they mattered. Consider for example the very simple concrete optimization instance `x+(x-x)` $\Rrightarrow$ `x`, in which the variable `x` is used three times in the original program. This optimization does not depend on *all* three uses referring to the same variable `x`. All that is required is that the uses in `(x-x)` refer to the same variable, whereas the first use of `x` can refer to another variable or, more broadly, to an entire expression.

Our insight is that a proof of correctness for the optimization instance can tell us precisely what conditions are necessary for the optimization to apply correctly. This proof could either be generated by a compiler (if the optimization instance was generated from a proof-generating compiler), or more realistically, it can be generated by performing translation validation on the optimization instance. Since the proof of correctness of the optimization instance captures precisely what parts of the programs mattered for correctness, it can be used as a guide for generalizing the instance. In particular, while keeping the structure of the proof unchanged, we simultaneously generalize the concrete optimization instance and its proof of correctness to get a generalized transformation and a proof that the generalized transformation is correct. In the example above, the proof of correctness for `x+(x-x)` $\Rrightarrow$ `x` does not rely on the first use of `x` referring to the same variable as the other uses in `(x-x)`, and so the optimization rule we would generate from the proof would not require them to be the same. In this way we can generalize concrete instances into optimization rules that apply in similar, but not identical, situations while still being correct.

Before generalizing proofs, though, we need a good language for expressing proofs of optimization correctness. This brings us back to our algebraic intermediate representation and equational reasoning, which we describe as we introduce our technique for inferring optimizations.

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

This chapter contains material taken from “Generating Compiler Optimization from Proofs”, by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to

republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

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*, volume 7, issue 1, 2011. The dissertation author was the primary 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*, 2011. The dissertation author was the secondary investigator and author of this paper.

Chapter 2

Optimizing with Equalities

Our approach for inferring optimizations 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 distributivity, i.e. $a * (b + c) = a * b + a * c$, but with more complicated computations such as branches and loops. We have designed a representation for computations, which we call Program Expression Graphs (PEGs), that meets these requirements. Similar to the *gated SSA* representation [41, 85], 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.

```
i := 0;                     i := 0;
while (...) {               while (...) {
   use(i * 5);                 use(i);
   i := i + 1;                 i := i + 5;
   if (...) {                  if (...) {
      i := i + 3;                 i := i + 15;
   }                           }
}                           }

          (a)                         (b)
```

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

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 E-PEG, and it is this E-PEG representation that we use in our approach. Using equivalence classes allows E-PEGs 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, E-PEGs are unique in their ability to represent multiple optimized versions of the input program. A more detailed discussion of how PEGs and E-PEGs relate to previous IRs can be found in Chapter 19.

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`.

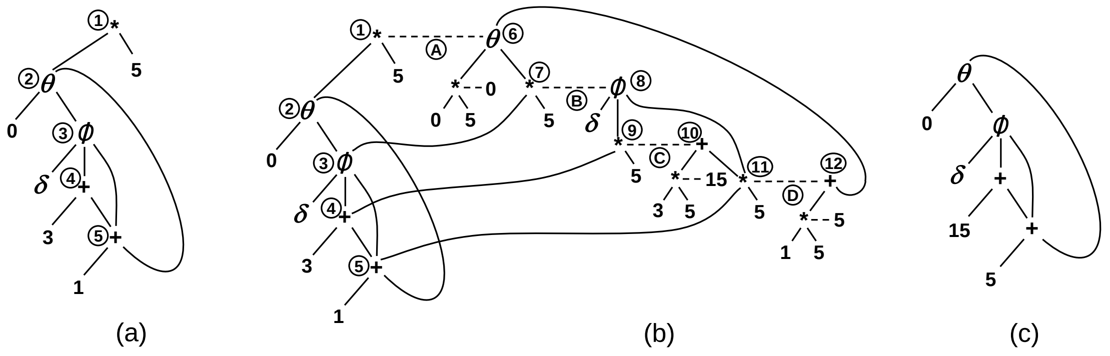


Figure 2.2. Loop-induction-variable Strength Reduction using PEGs: (a) shows the original PEG, (b) shows the E-PEG that our engine produces from the original PEG, and (c) shows the optimized PEG, which results by choosing nodes 6, 8, 10, and 12 from (b)

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

2.1 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 [21, 23, 60, 61] 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 [3, 20]. 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 [41, 85]: 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 does not use the branch condition in an interesting way, 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. We show how to express the value of variables after a loop in Chapter 3.

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 using 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 intraprocedural imperative code with branching 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 (or more generally we can use effect witnesses as described in Chapter 9). This functional representation of effectful programs allows our Peggy compiler to use PEGs to reason about Java and LLVM programs. The heap-summary node can also be used to encode method/function calls in an intraprocedural setting by simply threading the heap-summary node through special nodes representing method/function calls. Chapter 10 explains in more detail how we represent several features of Java programs in PEGs (including the heap and method calls).

2.2 Encoding Equalities using E-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 E-PEG is a graph that groups together PEG nodes that are equal into equivalence classes. As an example, Figure 2.2(b) shows the E-PEG 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 E-PEG is done through the application of optimizations, which in our approach take the form of equality analyses that add equality information to the E-PEG. 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 E-PEG. 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 E-PEG.

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 applications, 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 distributes through 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. As Chapter 11 will describe in more detail, our experiments show that 84% of methods can be completely saturated, without any bounds being imposed.

Figure 2.2(b) shows the saturated E-PEG 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 E-PEG. Furthermore, note that the instantiation of axiom (2.1) adds node 12 to the E-PEG, but it does not add the right child of node 12, namely $\theta(\ldots)*5$, because it is already represented in the E-PEG.

Once saturated with equalities, an E-PEG 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 E-PEG 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 E-PEG 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 E-PEG 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 value and nodes for all equivalence classes that the return value depends on. There are many plausible heuristics for choosing nodes in an E-PEG. In our Peggy implementation, we have chosen to select nodes using a pseudo-boolean solver, which is an integer-linear-program solver where variables are constrained to 0 or 1. In particular, we use a pseudo-boolean solver and a static cost model for every node to compute the lowest-cost program that is encoded in the E-PEG. In the example from Figure 2.2, the pseudo-boolean solver picks the nodes described above. Section 10.3 describes our technique for selecting nodes in more detail.

2.3 Benefits of our Approach

Optimization Order is Irrelevant 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 ubiquitous 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 [94] 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 optimization, 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 were 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 — Chapter 3 gives such an example. 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 or in what order relative to other optimizations.

Global Profitability Heuristics Profitability heuristics in traditional compilers tend to be local in nature, making it difficult to take into account the impact 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* [22], where the compiler simulates the inlining transformation, along with the impact 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 E-PEG stating that the call to a given function is equal to its body instantiated with the actual arguments. The resulting E-PEG 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 E-PEG 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.

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 3

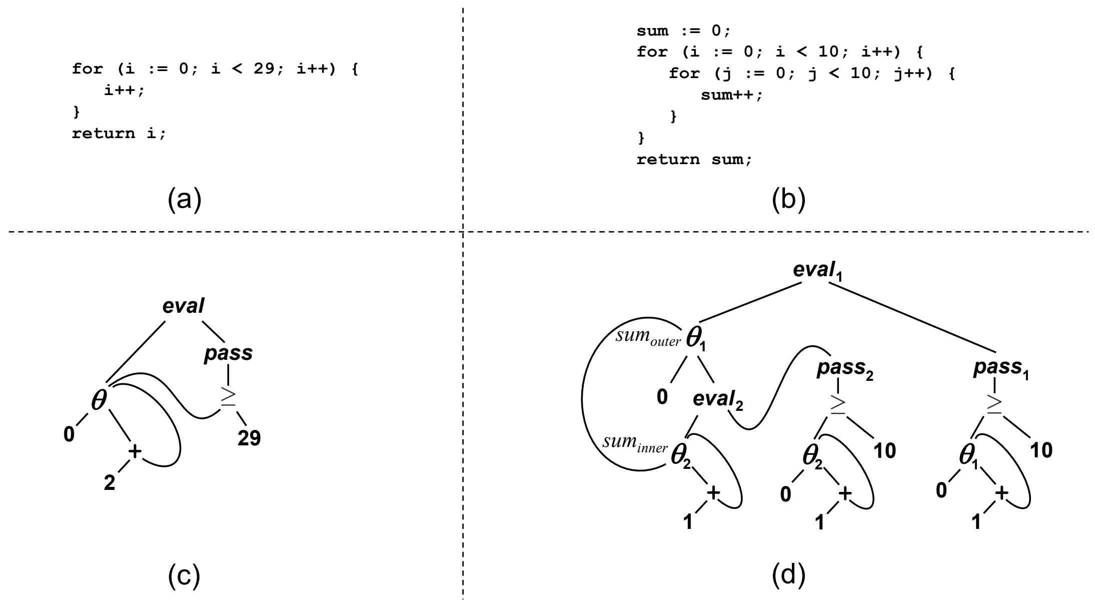
Reasoning about Loops

This chapter shows how our approach can be used to reason across nested loops. The example highlights the fact that a simple axiom set can produce unanticipated optimizations which traditional compilers would have to explicitly search for.

We start in Sections 3.1 and 3.2 by describing all PEG constructs used to represent loops. We then show in Section 3.3 how our approach can perform an inter-loop strength-reduction optimization.

3.1 Single Loop

Consider the simple loop from Figure 3.1(a). This loop iterates 15 times, incrementing the value of `i` each time by 2. The final value of `i` is then returned at the end of the function. The PEG for this code is shown in Figure 3.1(c). The value of `i` inside the loop is represented by a θ node. Intuitively, this θ node produces the sequence of values that `i` takes throughout the loop, in this case $[0,2,4,\ldots]$. The value of `i` after the loop is represented by the *eval* node at the top of the PEG. Given a sequence s and an index n, $eval(s,n)$ produces the n^{th} element of sequence s. To determine which element to select from a sequence, our PEG representation uses *pass* nodes. Given a sequence s of booleans, $pass(s)$ returns the index of the first element in the sequence that is true. In our example, the $\geq$ node uses the result of the θ node to produce the sequence of values



```
for (i := 0; i < 29; i++) {
   i++;
}
return i;
```

```
sum := 0;
for (i := 0; i < 10; i++) {
   for (j := 0; j < 10; j++) {
      sum++;
   }
}
return sum;
```

Figure 3.1. Two loops and their PEG representations

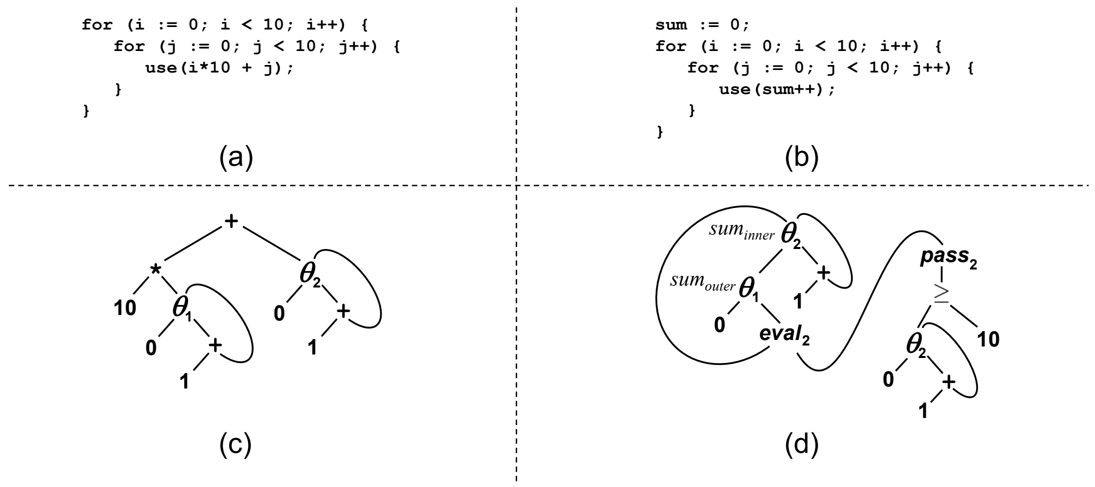
taken on by the boolean expression i $\geq$ 29 throughout the loop. This sequence is then sent to *pass*, which in this case produces the value 15, since the 15th value (counting from 0) of i in the loop (which is 30) is the first one to make i $\geq$ 29 true. The *eval* node then selects the 15th element of the sequence produced by the θ node, which is 30. In our previous example from Chapter 2, we omitted *eval*/*pass* from the PEG for clarity – because we were not interested in any of the values after the loop, the *eval*/*pass* nodes would not have been used in any reasoning.

Note that every loop-varying value will be represented by its own θ node, and so there will be one θ node in the PEG per live variable in the loop. Also, every variable that is live after the loop has its own *eval* node, which represents the value after the loop. However, there is only one *pass* node per loop, which represents the iteration at which the loop terminates. Thus, there can be many θ and *eval* nodes per loop, but only one *pass* node.

Since the *eval* and *pass* operators are often paired together, it is natural to consider merging them into a single operator. However, we have found that the separation is useful. For one, although there will be many *eval* nodes corresponding to a single loop, each loop has only one corresponding *pass* node. Having this single node to represent each loop is useful in many of the compilation stages for PEGs. Also, *pass* nodes are not the only nodes we will use as the second argument to an *eval* node. For example, to accomplish loop peeling (as shown in Section 4.3) we use ϕ nodes and other special-purpose nodes as the second argument. Furthermore, Section 10.4 will present a more detailed reflection on our design choice after we have shown how the *eval* and *pass* operators are used in our various compilation stages.

3.2 Nested Loops

We now illustrate, through an example, how nested loops can be encoded in our PEG representation. Consider the code snippet from Figure 3.1(b), which has two nested loops. The PEG for this code snippet is shown in Figure 3.1(d). Each θ, *eval*, and *pass* node is labeled with a subscript indicating what loop depth it operates on (we previously omitted these subscripts for clarity). The topmost $eval_1$ node represents the final value of `sum`. The node labeled sum_{inner} represents the value of `sum` at the beginning of the inner loop body. Similarly, sum_{outer} is the value of `sum` at the beginning of the outer loop body. Looking at sum_{inner}, we can see that: (1) on the first iteration (the left child of sum_{inner}), sum_{inner} gets the value of `sum` from the outer loop; (2) on other iterations, it gets one plus the value of `sum` from the previous iteration of the inner loop. Looking at sum_{outer}, we can see that: (1) on the first iteration, sum_{outer} gets 0; (2) on other iterations, it gets the value of `sum` right after the inner loop terminates. The value of `sum` after the inner loop terminates is computed using a similar *eval*/*pass* pattern as in Figure 3.1(c), as is the value of `sum` after the outer loop terminates.



```
for (i := 0; i < 10; i++) {
   for (j := 0; j < 10; j++) {
      use(i*10 + j);
   }
}
```

(a)

```
sum := 0;
for (i := 0; i < 10; i++) {
   for (j := 0; j < 10; j++) {
      use(sum++);
   }
}
```

(b)

Figure 3.2. Two equivalent loops and their PEG representations. The PEGs for the expressions inside the `use` statements in (a) and (b) are shown in (c) and (d), respectively.

3.3 Inter-Loop Strength Reduction

Our approach allows an optimizing compiler to perform intricate optimizations of looping structures. We present such an example here with a kind of inter-loop strength reduction. Consider the code snippets from Figure 3.2(a) and (b). The code in Figure 3.2(b) is equivalent to the code in Figure 3.2(a), but it is faster because `sum++` is cheaper than $\mathtt{i} * 10 + j$. We show how our approach can transform the code in Figure 3.2(a) to the code in Figure 3.2(b).

The PEGs for the code from parts (a) and (b) are shown in parts (c) and (d), respectively. We do not show the entire PEGs, but only the parts that are relevant to the optimization – namely the PEGs for the expressions inside the `use` statements. More specifically, Figure 3.2(c) shows the PEG for `i*10 + j`, which is the PEG that our optimization will apply to. The top-level + node occurs in some larger PEG context which includes *eval* and *pass* nodes, but we do not show the larger context (i.e. the parents of +) because they are not used in this example, except in one step that we will make explicit. The result of the optimization, in PEG form, is shown in Figure 3.2(d).

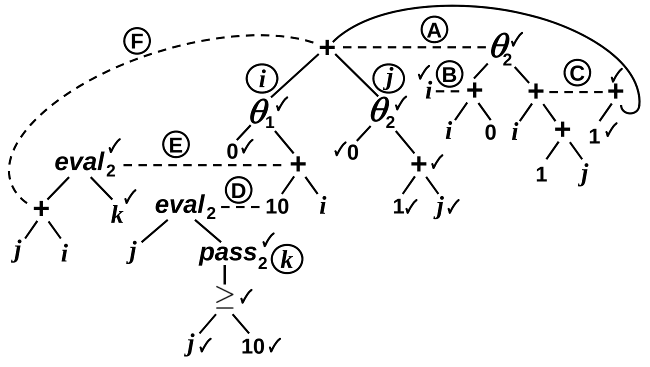


Figure 3.3. E-PEG that results from running the saturation engine on the PEG from Figure 3.2(c). By picking the nodes that are checkmarked, we get the PEG from Figure 3.2(d). To make the graph more readable, we sometimes label nodes and then connect an edge directly to a label name, rather than connecting it to the node with that label. For example, consider node j in the E-PEG, which reads as $\theta_2(0, 1 + j)$. Rather than explicitly drawing an edge from $+$ to j, we connect $+$ to a new copy of label j.

This is the PEG for the `sum++` expression from Figure 3.2(b). Note that the code snippet in Figure 3.2(b) is the same as Figure 3.1(b), and as a result Figure 3.2(d) is just the sum_{inner} node from Figure 3.1(d) along with its children. To summarize, in terms of PEGs, our optimization will replace the + node from Figure 3.2(c), which occurs in some larger PEG context, with the sum_{inner} node from Figure 3.2(d). The surrounding PEG context, which we do not show, remains unchanged.

Figure 3.3 shows the saturated E-PEG that results from running the saturation engine on the PEG from Figure 3.2(c). The checkmarks indicate which nodes will eventually be selected – they can be ignored for now. In drawing Figure 3.3, we have already performed loop-induction-variable strength reduction on the left child of the topmost $+$ from Figure 3.2(c). In particular, this left child has been replaced with a new node i, where $i = \theta_1(0, 10 + i)$. We skip the steps in doing this because they are similar to the ones described in Section 2.2.

Figure 3.3 shows the relevant equalities that our saturation engine would add. We describe each in turn.

- Edge A is added by distributing $+$ over θ_2:

$$i + \theta_2(0, 1 + j) = \theta_2(i + 0, i + (1 + j))$$

- Edge B is added because 0 is the identity of $+$:

$$i + 0 = i$$

- Edge C is added because addition is associative and commutative:

$$i + (1 + j) = 1 + (i + j)$$

- Edge D is added because 0, incremented n times, produces n:

$$eval_\ell(id_\ell, pass_\ell(id_\ell \geq n)) = n \text{ where } id_\ell = \theta_\ell(0, 1 + id_\ell)$$

  This is an example of a loop optimization expressible as a simple PEG axiom.

- Edge E is added by distributing $+$ over the first child of $eval_2$:

$$eval_2(j, k) + i = eval_2(j + i, k)$$

- Edge F is added because addition is commutative:

$$j + i = i + j$$

We use checkmarks in Figure 3.3 to highlight the nodes that Peggy would select using its pseudo-boolean profitability heuristic. These nodes constitute exactly the PEG from Figure 3.2(d), meaning that Peggy optimizes the code in Figure 3.2(a) to the one in Figure 3.2(b).

Summary This example illustrates several points. First, it shows how a transformation that locally seems undesirable, namely transforming the constant 10 into an expensive loop (edge D), in the end leads to much better code. Our global profitability heuristic is perfectly suited for taking advantage of these situations. Second, it shows an example of an *unanticipated optimization*, namely an optimization that we did not realize would fall out from the simple equality analyses we already had in place. In a traditional compilation system, a specialized analysis would be required to perform this optimization, whereas in our approach the optimization simply happens without any special casing. In this way, our approach essentially allows a few general equality analyses to do the work of many specialized transformations. Finally, it shows how our approach is able to reason about complex loop interactions, something that is beyond the reach of current superoptimizer-based techniques.

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

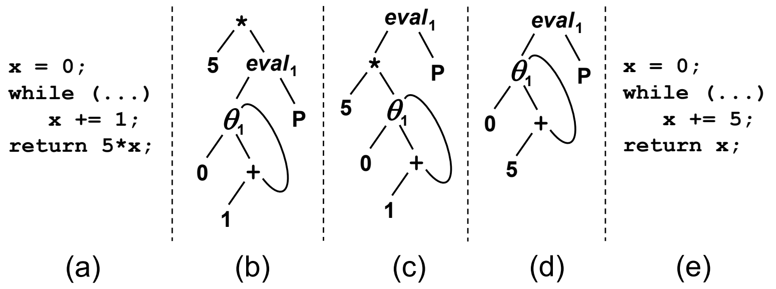
Chapter 4

Local Changes with Global Impacts

The axioms we apply during equality saturation tend to be simple and local in nature. It is therefore natural to ask how such axioms can perform anything more than peephole optimizations. The examples shown so far have already given a flavor of how local reasoning on a PEG can lead to complex optimizations. In this chapter, 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 chapter by describing some loop optimizations that we have not fully explored using PEGs and which could pose additional challenges.

4.1 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 4.1. 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.



```
x = 0;
while (...)
   x += 1;
return 5*x;
```

```
x = 0;
while (...)
   x += 5;
return x;
```

Figure 4.1. 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

Peggy can perform this optimization using local axiom applications without requiring any additional non-local reasoning. Figure 4.1(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 E-PEG 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 since they are similar to the earlier example from Chapter 2). 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.

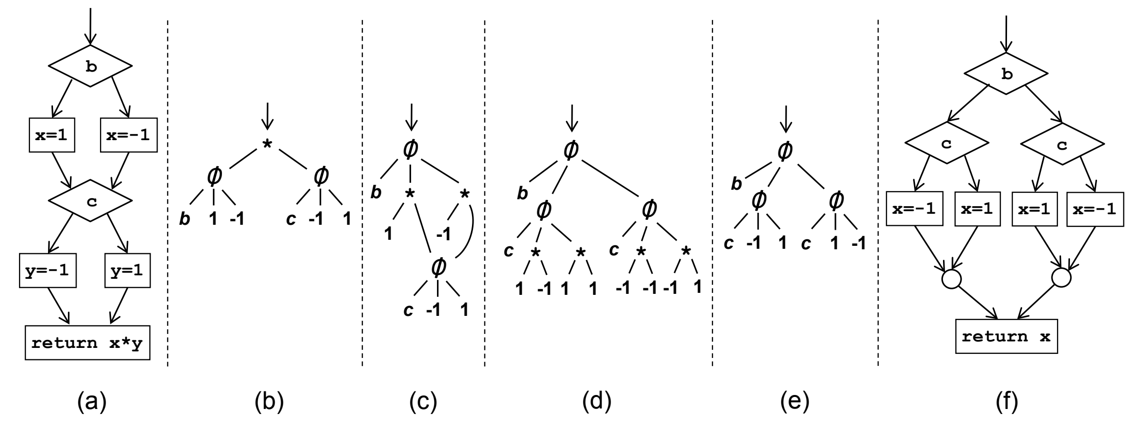
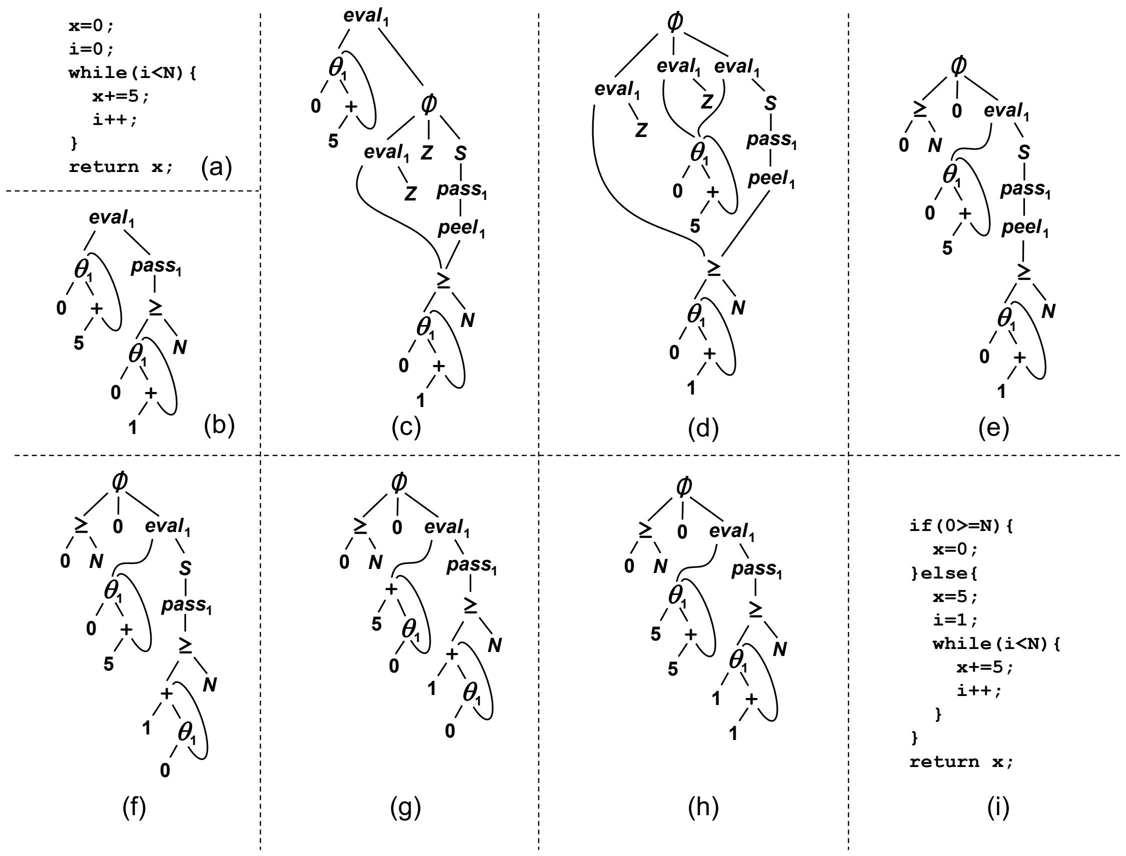


Figure 4.2. An example of how local changes in the PEG can cause large changes in the CFG: (a) the original CFG, (b) the original PEG, (c) the PEG after distributing $*$ through the left-hand ϕ, (d) the PEG after distributing $*$ through the bottom ϕ, (e) the PEG after constant folding, (f) the resulting CFG

4.2 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 4.2. 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 4.2(b) shows the PEG version of the original CFG, and parts (c)-(e) show the relevant portions of the E-PEG 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 CFG. This illustrates how small, local changes to the PEG representation can have large, far-reaching impacts on the program.



```
x=0;
i=0;
while(i<N){
  x+=5;
  i++;
}
return x;
```

```
if(0>=N){
  x=0;
}else{
  x=5;
  i=1;
  while(i<N){
    x+=5;
    i++;
  }
}
return x;
```

Figure 4.3. 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)

4.3 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 4.3(a). We want to perform loop peeling on this code, which will result in the code shown in Figure 4.3(i). This can be done through axiom application through the following steps, depicted in Figure 4.3 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 $pass_1(C) = \phi(eval_1(C,\mathrm{Z}),\mathrm{Z},\mathrm{S}(pass_1(peel_1(C))))$, 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 (i.e. $S = \lambda x.x+1$), and *peel* takes a sequence and strips off the first element (i.e. $peel(C)[i] = C[i+1]$). 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), propagates the two $eval_1(\cdot,\mathrm{Z})$ expressions downward, using the axiom $eval_1(op(a_1,\ldots,a_k),\mathrm{Z}) = op(eval_1(a_1,\mathrm{Z}),\ldots,eval_1(a_k,\mathrm{Z}))$ when *op* is a domain operator, such as $+, *$, or S. When the *eval* meets a θ, it simplifies using the following axiom: $eval_1(\theta_1(A,B),\mathrm{Z}) = A$. Furthermore, we also use the axiom $eval_1(C,\mathrm{Z}) = C$ for any constant or parameter C, which is why $eval_1(\mathtt{N},\mathrm{Z}) = \mathtt{N}$.

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

The fifth step, shown in part (g), involves removing the S node using the axiom

$eval_1(\theta_1(A,B),\mathrm{S}(C)) = eval_1(B,C)$.

The final step (which is not strictly necessary, as the peeling is complete at this point) involves distributing the two plus operators through their θs 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).

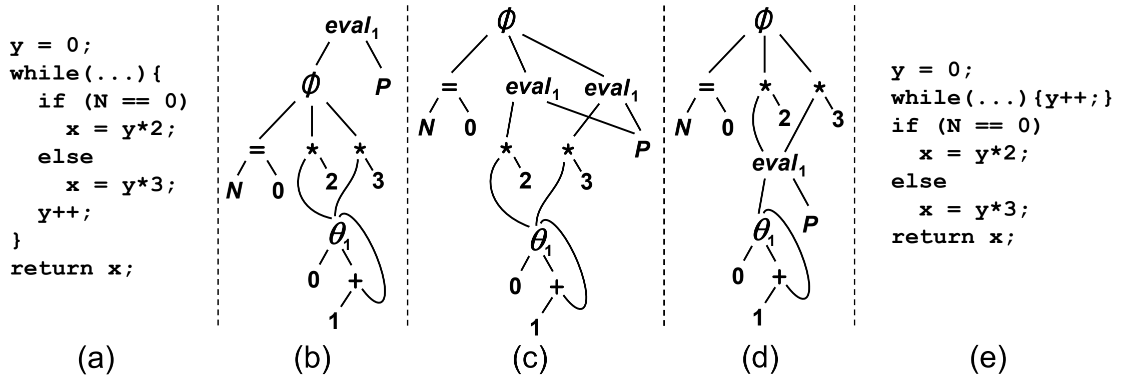
It is interesting to see that this version of loop peeling includes the conditional test before executing the peeled loop to make sure that the original loop would have iterated at least once. Another way to implement loop peeling is to exclude this test, opting only to peel when the analysis can determine statically that the loop will always have at least one iteration. This limits peeling to loops 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 4.3(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.

4.4 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 is 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 4.4(a). We assume that `N` is a parameter or a variable



```
y = 0;
while(...){
  if (N == 0)
    x = y*2;
  else
    x = y*3;
  y++;
}
return x;
```

```
y = 0;
while(...){y++;}
if (N == 0)
  x = y*2;
else
  x = y*3;
return x;
```

Figure 4.4. 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)

initialized elsewhere, and N is clearly not altered inside the loop. Hence the condition for 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) distributes 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* as $eval_1$. Part (d) comes from distributing the two *eval* nodes through the multiplication operator, using the axiom $eval_1(op(A,B),P) = op(eval_1(A,P), eval_1(B,P))$ 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.

4.5 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. For example, in the inter-loop strength reduction example from Chapter 3, the ideal output would be a single loop that increments the `sum` variable. 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 require adding new operators to the PEG, which would require carefully formalizing their semantics and the 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.

Acknowledgements

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Formalization of Equality Saturation

Having given an intuition for how our approach works through examples, we now move to a formal description. Figure 5.1 shows the Optimize function that embodies our approach. Optimize takes four steps: first, it converts the input CFG into an internal representation of the program; second, it saturates this internal representation with equalities; third, it uses a global profitability heuristic to select the best program from the saturated representation; finally, it converts the selected program back to a CFG.

An implementation of our approach therefore consists of three components: (1) an IR where equality reasoning is effective, along with the translation functions ConvertToIR and ConvertToCFG, (2) a saturation engine Saturate, and (3) a global profitability heuristic SelectBest. Future chapters will show how we implement these three components in our Peggy compiler.

```
1: function Optimize($cfg : CFG$) : $CFG$
2: let $ir$ = ConvertToIR($cfg$)
3: let $saturated\_ir$ = Saturate($ir, A$)
4: let $best$ = SelectBest($saturated\_ir$)
5: return ConvertToCFG($best$)
```

Figure 5.1. Optimization phase in our approach. We assume a global set A of equality analyses to be run.

Saturation Engine The saturation engine Saturate infers equalities by repeatedly running a set A of equality analyses. Given an equality analysis $a \in A$, we define $ir_1 \xrightarrow{a} ir_2$ to mean that ir_1 produces ir_2 when the equality analysis a runs and adds some equalities to ir_1. If a chooses not to add any equalities, then ir_2 is simply the same as ir_1. Note that a is not required to be deterministic: given a single ir_1, there may be many ir_2 such that $ir_1 \xrightarrow{a} ir_2$. This non-determinism gives equality analyses that are applicable in multiple locations in the IR (e.g. an E-PEG) the choice of where to apply. For example, the distributivity of an operator could apply in many locations, and the non-determinism allows the distributivity analysis the flexibility of choosing which instances of distributivity to apply.

We assume a partial order $\sqsubseteq$ on IRs indicating amount of information (e.g. for E-PEGs $ir_1 \sqsubseteq ir_2$ holds iff the nodes in ir_1 are a subset of the nodes in ir_2, and the equalities in ir_1 are a subset of the equalities in ir_2). The addivite aspect of our approach is formalized by the following property indicating that equality analyses only add information:

$$(ir_1 \xrightarrow{a} ir_2) \Rightarrow ir_1 \sqsubseteq ir_2 \tag{5.1}$$

We define an equality analysis a to be monotonic iff:

$$(ir_1 \sqsubseteq ir_2) \wedge (ir_1 \xrightarrow{a} ir_1') \Rightarrow \exists ir_2'.[(ir_2 \xrightarrow{a} ir_2') \wedge (ir_1' \sqsubseteq ir_2')] \tag{5.2}$$

This basically states that if a is able to apply to ir_1 to produce ir_1' and $ir_1 \sqsubseteq ir_2$, then there is a way to apply a on ir_2 to get some ir_2' such that $ir_1' \sqsubseteq ir_2'$

If a is monotonic, properties (5.1) and (5.2) immediately imply the following property:

$$(ir_1 \xrightarrow{a} ir_1') \wedge (ir_1 \xrightarrow{b} ir_2) \Rightarrow \exists ir_2'.[(ir_2 \xrightarrow{a} ir_2') \wedge (ir_1' \sqsubseteq ir_2')] \tag{5.3}$$

Intuitively, this simply states that applying an equality analysis b before a cannot make a less effective.

We now define $ir_1 \to ir_2$ as:

$$ir_1 \to ir_2 \iff \exists a \in A \;.\; (ir_1 \xrightarrow{a} ir_2 \wedge ir_1 \neq ir_2)$$

The $\to$ relation formalizes one step taken by the saturation engine. We also define $\to^*$ to be the reflexive transitive closure of $\to$. The $\to^*$ relation formalizes an entire run of the saturation engine. We call a sequence $ir_1 \xrightarrow{a} ir_2 \xrightarrow{b} \ldots$ a trace through the saturation engine. We define ir_2 to be a normal form of ir_1 if $ir_1 \to^* ir_2$ and there is no ir_3 such that $ir_2 \to ir_3$. It is straightforward to show the following property:

Given a set A of monotonic equality analyses, if ir_2 is a normal formal of ir_1, then any other normal form of ir_1 is equal to ir_2. (5.4)

In essence, property (5.4) states that if one trace through the saturation engine leads to a normal form (and thus a saturated IR), then any other trace that also leads to a normal form results in the same saturated IR. In other words, if a given *ir* has a normal form, it is unique.

If the set A of analyses makes the saturation engine terminate on all inputs, then property (5.4) implies that the engine is convergent, meaning that every *ir* has a unique normal form. In general, however, equality saturation may not terminate. For a given *ir* there may not be a normal form, and even if there is a normal form, some traces may not lead to it because they run forever. Non-termination occurs when the saturation engine never runs out of equality analyses that can match in the IR and produce more information. For example, the axiom $A = (A+1) - 1$ used in the direction from left to right can be applied an unbounded number of times, producing successively larger and larger expressions (`x`, `(x+1)-1`, `(((x+1)-1)+1)-1`, and so on). An inlining axiom applied to a recursive function can also be applied an unbounded number of times.

Because unrestricted saturation may not terminate, we bound the number of times that individual analyses can run, thus ensuring that the Saturate function will always halt. In the case when the saturation engine is stopped early, we cannot provide the same convergence property, but property (5.3) still implies that no area of the search space can be made unreachable by applying an equality analysis (a property that traditional compilation systems lack).

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 6

PEGs and E-PEGs

The first step in implementing our approach from the Chapter 5 is to pick an appropriate IR. To this end, we have designed a new IR called the E-PEG which can simultaneously represent multiple optimized versions of the input program. We first give a formal description of our IR (Section 6.1), then we present its benefits (Section 6.4), and finally we give a detailed description of how to translate from CFGs to our IR and back (Chapters 7 and 8).

6.1 Formalization of PEGs

A PEG is a triple $\langle N, L, C \rangle$, where N is a set of nodes, $L : N \to F$ is a labeling that maps each node to a semantic function from a set of semantic functions F, and $C : N \to N^*$ is a function that maps each node to its children (i.e. arguments). For a given node n, if $L(n) = f$, we say that n is *labeled* with f. We say that a node n' is a *child* of node n if n' is an element of $C(n)$. Finally, we say that n_k is a *descendant* of n_0 if there is a sequence of nodes $n_0, n_1, \ldots, n_k$ such that n_{i+1} is a child of n_i for $0 \leq i < k$.

Types Before giving the definition of semantic functions, we first define the types of values that these functions operate over. Values that flow through a PEG are lifted in two ways. First, they are $\perp$-lifted, meaning that we add the special value $\perp$ to each type domain. The $\perp$ value indicates that the computation does not terminate. Formally, for each type τ, we define $\tau_\perp = \tau \cup \{\perp\}$.

Second, values are loop lifted, meaning that, instead of representing the value at a particular iteration, PEG nodes represent values for all iterations at the same time. Formally, we let $\mathscr{L}$ be a set of loop identifiers, with each $\ell \in \mathscr{L}$ representing a loop from the original code (in our previous examples we used integers). We assume a partial order $\leq$ that represents the loop nesting structure: $\ell < \ell'$ means that ℓ' is nested within ℓ. An iteration index $\mathbf{i}$ captures the iteration state of all loops in the PEG. In particular, $\mathbf{i}$ is a function that maps each loop identifier $\ell \in \mathscr{L}$ to the iteration that loop ℓ is currently on. Suppose for example that there are two nested loops in the program, identified as ℓ_1 and ℓ_2. Then the iteration index $\mathbf{i} = [\ell_1 \mapsto 5, \ell_2 \mapsto 3]$ represents the state where loop ℓ_1 is on the 5th iteration and loop ℓ_2 is on the 3rd iteration. We let $\mathbb{I} = \mathscr{L} \to \mathbb{N}$ be the set of all loop iteration indices (where $\mathbb{N}$ denotes the set of non-negative integers). For $\mathbf{i} \in \mathbb{I}$, we use the notation $\mathbf{i}[\ell \mapsto n]$ to denote a function that returns the same value as $\mathbf{i}$ on all inputs, except that it returns n on input ℓ. The output of a PEG node is a map from loop iteration indices in $\mathbb{I}$ to values. In particular, for each type τ, we define a loop-lifted version $\widetilde{\tau} = \mathbb{I} \to \tau_\perp$. PEG nodes operate on these loop-lifted types.

Semantic Functions The semantic functions in F actually implement the operations represented by the PEG nodes. Each function $f \in F$ has type $\widetilde{\tau_1} \times \ldots \times \widetilde{\tau_k} \to \widetilde{\tau}$ for some k. Such an f can be used as the label for a node that has k children. That is to say, if $L(n) = f$, where $f : \widetilde{\tau_1} \times \ldots \times \widetilde{\tau_k} \to \widetilde{\tau}$, then $C(n)$ must be a list of k nodes.

The set of semantic functions F is divided into two: $F = \mathit{Prims} \cup \mathit{Domain}$. *Prims* contains the *primitive functions* like ϕ and θ, which are built into the PEG representation, whereas *Domain* contains semantic functions for particular domains like arithmetic.

Figure 6.1 defines the primitive functions $\mathit{Prims} = \{\phi, \theta_\ell, \mathit{eval}_\ell, \mathit{pass}_\ell\}$. These

$$\boxed{\phi : \widetilde{\mathbb{B}} \times \widetilde{\tau} \times \widetilde{\tau} \to \widetilde{\tau}}$$

$$\phi(cond, t, f)(\mathbf{i}) = \begin{cases} \textbf{if } cond(\mathbf{i}) = \bot & \textbf{then } \bot \\ \textbf{if } cond(\mathbf{i}) = \textbf{true} & \textbf{then } t(\mathbf{i}) \\ \textbf{if } cond(\mathbf{i}) = \textbf{false} & \textbf{then } f(\mathbf{i}) \end{cases}$$

$$\boxed{\theta_\ell : \widetilde{\tau} \times \widetilde{\tau} \to \widetilde{\tau}}$$

$$\theta_\ell(base, loop)(\mathbf{i}) = \begin{cases} \textbf{if } \mathbf{i}(\ell) = 0 \ \textbf{then } base(\mathbf{i}) \\ \textbf{if } \mathbf{i}(\ell) > 0 \ \textbf{then } loop(\mathbf{i}[\ell \mapsto \mathbf{i}(\ell) - 1]) \end{cases}$$

$$\boxed{eval_\ell : \widetilde{\tau} \times \widetilde{\mathbb{N}} \to \widetilde{\tau}}$$

$$eval_\ell(loop, idx)(\mathbf{i}) = \begin{cases} \textbf{if } idx(\mathbf{i}) = \bot \ \textbf{then } \bot \\ \textbf{else } monotonize_\ell(loop)(\mathbf{i}[\ell \mapsto idx(\mathbf{i})]) \end{cases}$$

$$\boxed{pass_\ell : \widetilde{\mathbb{B}} \to \widetilde{\mathbb{N}}}$$

$$pass_\ell(cond)(\mathbf{i}) = \begin{cases} \textbf{if } \mathscr{I} = \emptyset & \textbf{then } \bot \\ \textbf{if } \mathscr{I} \neq \emptyset & \textbf{then } \min \mathscr{I} \end{cases}$$

where $\mathscr{I} = \{i \in \mathbb{N} \mid monotonize_\ell(cond)(\mathbf{i}[\ell \mapsto i]) = \textbf{true}\}$

where $monotonize_\ell : \widetilde{\tau} \to \widetilde{\tau}$ is defined as:

$$monotonize_\ell(value)(\mathbf{i}) = \begin{cases} \textbf{if } \exists\, 0 \leq i < \mathbf{i}(\ell).\ value(\mathbf{i}[\ell \mapsto i]) = \bot \ \textbf{then } \bot \\ \textbf{if } \forall\, 0 \leq i < \mathbf{i}(\ell).\ value(\mathbf{i}[\ell \mapsto i]) \neq \bot \ \textbf{then } value(\mathbf{i}) \end{cases}$$

Figure 6.1. Definition of primitive PEG functions. The important notation: $\mathscr{L}$ is the set of loop identifiers, $\mathbb{N}$ is the set of non-negative integers, $\mathbb{B}$ is the set of booleans, $\mathbb{I} = \mathscr{L} \to \mathbb{N}$, $\tau_\bot = \tau \cup \{\bot\}$, and $\widetilde{\tau} = \mathbb{I} \to \tau_\bot$.

functions are polymorphic in τ, in that they can be instantiated for various τs, ranging from basic types like integers and strings to complicated types like the heap summary nodes that Peggy uses to represent Java's heap. The definitions of $eval_\ell$ and $pass_\ell$ make use of the function $monotonize_\ell$, whose definition is given in Figure 6.1. The $monotonize_\ell$ function transforms a sequence so that, once an indexed value is undefined, all following indexed values are undefined. The $monotonize_\ell$ function formalizes the fact that once a value is undefined at a given loop iteration, the value remains undefined at subsequent iterations.

The domain semantic functions are defined as $\mathit{Domain} = \{\widetilde{op} \mid op \in \mathit{DomainOp}\}$, where *DomainOp* is a set of domain operators (like $+$, $*$ and $-$ in the case of arithmetic), and $\widetilde{op}$ is a $\bot$-lifted and then loop-lifted version of op. Intuitively, the $\bot$-lifted version of an operator works like the original operator except that it returns $\bot$ if any of its inputs are $\bot$, and the loop-lifted version of an operator applies the original operator for each loop index.

As an example, the semantic function of $+$ in a PEG is $\widetilde{+}$, and the semantic function of 1 is $\widetilde{1}$ (since constants like 1 are simply nullary operators). However, to make the notation less crowded, we omit the tildes on all domain operators.

Node Semantics For a PEG node $n \in N$, we denote its semantic value by $[\![n]\!]$. We assume that $[\![\cdot]\!]$ is lifted to sequences N^* in the standard way. The semantic value of n is defined as:

$$[\![n]\!] = L(n)([\![C(n)]\!]) \tag{6.1}$$

Equation 6.1 is essentially the evaluation semantics for expressions. The only complication here is that our expression graphs are recursive. In this setting, one can think of Equation 6.1 as a set of recursive equations to be solved. To guarantee that a unique solution exists, we impose some well-formedness constraints on PEGs.

Definition 6.1 (PEG Well-formedness). *A PEG is well formed iff:*

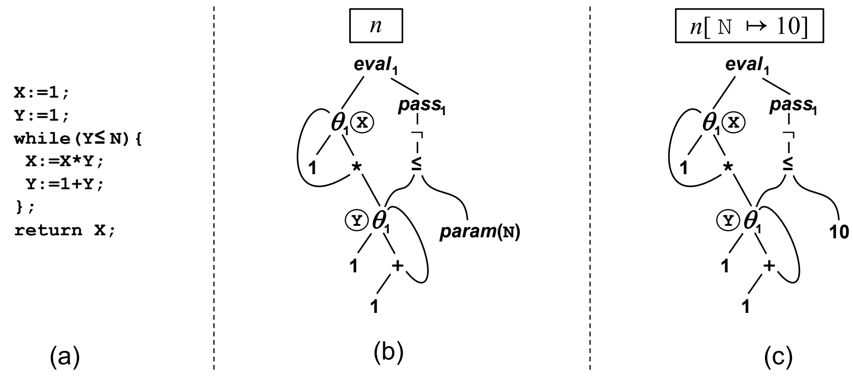
1. *All cycles pass through the second child edge of a θ*
2. *A path from a θ_ℓ, $eval_\ell$, or $pass_\ell$ to a $\theta_{\ell'}$ implies $\ell' \leq \ell$ or the path passes through the first child edge of an $eval_{\ell'}$ or $pass_{\ell'}$*
3. *All cycles containing $eval_\ell$ or $pass_\ell$ contain some $\theta_{\ell'}$ with $\ell' < \ell$*

Condition 1 states that all cyclic paths in the PEG are due to looping constructs. Condition 2 states that a computation in an outer loop cannot reference a value from inside an inner loop. Condition 3 states that the final value produced by an inner loop cannot be expressed in terms of itself, except if it is referencing the value of the inner loop from a *previous* outer loop iteration. From this point on, all of our discussion of PEGs will assume they are well formed.

Theorem 6.1. *If a PEG is well formed, then for each node n in the PEG there is a unique semantic value* $[\![n]\!]$ *satisfying Equation 6.1.*

The proof is by induction over the strongly-connected-component DAG of the PEG and the loop-nesting structure $\leq$.

Evaluation Semantics The semantic function $[\![\cdot]\!]$ can be evaluated on demand, which provides an executable semantics for PEGs. For example, suppose we want to know the result of $eval_\ell(x, pass_\ell(y))$ at some iteration state $\mathbf{i}$. To determine which case of $eval_\ell$'s definition we are in, we must evaluate $pass_\ell(y)$ on $\mathbf{i}$. From the definition of $pass_\ell$, we must compute the minimum i that makes y true. To do this, we iterate through values of i until we find an appropriate one. The value of i we have found is the number of times the loop iterates, and we can use this i back in the $eval_\ell$ function to extract the appropriate value out of x. This example shows how an on-demand evaluation of an *eval*/*pass* sequence essentially leads to an operational semantics for loops. Though it may seem that this semantics requires each loop to be evaluated twice (once to determine the *pass* value and once to determine the *eval* result), a practical implementation of PEGs (such as our PEG-to-imperative-code conversion algorithm in Chapter 8) can use a single loop to compute both the *pass* result and the *eval* result.



```
X:=1;
Y:=1;
while(Y≤N){
 X:=X*Y;
 Y:=1+Y;
};
return X;
```

Figure 6.2. Example showing a PEG with parameter nodes: (a) shows code for computing the factorial of N, where N is a parameter; (b) shows the PEG with n being the value returned; and (c) shows $n[\texttt{N} \mapsto 10]$, which is now a PEG whose semantics is well defined in our formalism

Parameter Nodes Our PEG definition can easily be extended to have *parameter nodes*, which are useful for encoding the input parameters of a function or method. In particular, we allow a PEG $\langle N, L, C\rangle$ to have a (possibly empty) set $N_p \subseteq N$ of parameter nodes. A parameter node n does not have any children, and its label is of the form $param(x)$ where x is the variable name of the parameter. To accommodate for this in the formalism, we extend the type of our labeling function L to $L : N \to F \cup P$, where $P = \{param(x) \mid x \text{ is a variable name}\}$. There are several ways to give semantics to PEGs with parameter nodes. One way is to update the semantic functions in Figure 6.1 to pass around a value context mapping variables to values. Another way, which we use here, is to first apply a substitution to the PEG that replaces all parameter nodes with constants, and then use the node semantics $[\![\cdot]\!]$ defined earlier. The node semantics $[\![\cdot]\!]$ is well defined on a PEG where all parameters have been replaced, since $L(n)$ in this case would always return a semantic function from F, never a parameter label $param(x)$ from P.

We use the following notation for substitution: given a PEG node n, a variable name x, and a constant c (which is just a nullary domain operator $op \in Domain$), we use

$n[x \mapsto c]$ to denote n with every descendant of n that is labeled with $param(x)$ replaced with a node labeled with $\widetilde{c}$. Figure 6.2 shows an example with parameter nodes and an example of the substitution notation.

6.2 Formalization of E-PEGs

An E-PEG is a PEG with a set of equalities E between nodes. Thus, formally, an E-PEG is a quadruple $\langle N, L, C, E\rangle$, where $\langle N, L, C\rangle$ is a PEG and $E \subseteq N \times N$ is a set of pairs of nodes representing equalities. An equality between n and n' denotes value equality: $[\![n]\!] = [\![n']\!]$. The set E forms an equivalence relation $\sim$ (that is, $\sim$ is the reflexive transitive symmetric closure of E), which in turn partitions the PEG nodes into equivalence classes. We denote by $[n]$ the equivalence class that n belongs, so that $[n] = \{n' \in N \mid n' \sim n\}$. We denote by N/E the set of all equivalence classes. For $n \in N$, we denote by $params(n)$ the list of equivalence classes that are parameters to n. In particular, if $C(n) = (n_1, \ldots, n_k)$ then $params(n) = ([n_1], \ldots, [n_k])$. As mentioned in more detail in Chapter 10, our implementation strategy keeps track of these equivalence classes, rather than the set E.

6.3 Built-in Axioms

We have developed a set of PEG built-in axioms that state properties of the primitive semantic functions. These axioms are used in our approach as a set of equality analyses that enable reasoning about primitive PEG operators. Some important built-in axioms are given below, where $\bullet$ denotes "does not matter":

$$\begin{aligned}
\theta_\ell(A,B) &= \theta_\ell(eval_\ell(A,0),B)\\
eval_\ell(\theta_\ell(A,\bullet),0) &= eval_\ell(A,0)\\
eval_\ell(eval_\ell(A,B),C) &= eval_\ell(A,eval_\ell(B,C))\\
pass_\ell(\mathbf{true}) &= 0\\
pass_\ell(\theta_\ell(\mathbf{true},\bullet)) &= 0\\
pass_\ell(\theta_\ell(\mathbf{false},A)) &= pass_\ell(A)+1
\end{aligned}$$

Furthermore, some axioms make use of an invariance predicate: $invariant_\ell(n)$ is true if the value of n does not vary on loop ℓ. Although we define $invariant_\ell$ here first, $invariant_\ell$ will be used more broadly than for defining axioms. It will also be used in Section 8.9 to optimize the PEG-to-imperative-code translation, and in Section 10.3 to help us define the cost model for PEGs. Invariance can be computed using several syntactic rules, as shown in the following definition, although there are PEG nodes which are semantically invariant but do not satisfy the following syntactic predicate. Note that we sometimes use "n is $invariant_\ell$" instead of $invariant_\ell(n)$.

Definition 6.2 (Invariance Predicate)**.** *The $invariant_\ell(n)$ predicate is the largest predicate that satisfies the following three rules:*

1. *if $L(n)$ is θ_ℓ, then n is not $invariant_\ell$*
2. *if $L(n)$ is $eval_\ell$, then if the second child of n is not $invariant_\ell$ then n is also not $invariant_\ell$*
3. *otherwise if $L(n)$ is not $pass_\ell$, then if any child of n is not $invariant_\ell$ then n is also not $invariant_\ell$*

Note that, due to the last rule, nodes without any children, such as constants and parameter nodes, will always be $\mathit{invariant}_\ell$ for all ℓ. Also, since no rule restricts pass_ℓ nodes, such nodes will always be $\mathit{invariant}_\ell$. This syntactic definition of $\mathit{invariant}_\ell$ is best computed using an optimistic dataflow analysis.

Having defined $\mathit{invariant}_\ell$, the following built-in axioms hold if $\mathit{invariant}_\ell(A)$ holds:

$$\begin{aligned} \mathit{eval}_\ell(A, \bullet) &= A \\ x &= A \quad \text{where } x = \theta_\ell(A, x) \\ \mathit{peel}_\ell(A) &= A \end{aligned}$$

One of the benefits of having a well defined semantics for primitive PEG functions is that we can reason formally about these functions. To demonstrate that this is feasible, we used our semantics to prove a handful of axioms, in particular, the above axioms, and all the axioms required to perform the optimizations presented in Chapters 2 through 4. Appendix A contains the much longer list of all axioms that we have used in Peggy.

6.4 How PEGs Enable our Approach

The key feature of PEGs that makes our equality-saturation approach effective is that they are referentially transparent, which intuitively means that the value of an expression depends only on the values of its constituent expressions [49, 67, 76]. In our PEG representation, referential transparency can be formalized as follows:

$$\forall (n, n') \in N^2.\ L(n) = L(n') \wedge [\![C(n)]\!] = [\![C(n')]\!] \implies [\![n]\!] = [\![n']\!]$$

This property follows from the definition in Equation (6.1), and the fact that for any n, $L(n)$ is a pure mathematical function.

Referential transparency makes equality reasoning effective because it allows us to show that two expressions are equal by only considering their constituent expressions, without having to worry about side effects. Furthermore, referential transparency has the benefit that a single node in the PEG entirely captures the value of a complex program fragment (including loops) enabling us to record equivalences between program fragments by using equivalence classes of nodes. Contrast this to CFGs, where to record equality between complex program fragments one would have to record subgraph equality.

Finally, PEGs allow us to record equalities at the granularity of individual values, for example the iteration count in a loop, rather than at the level of the entire program state. Again, contrast this to CFGs, where the simplest form of equality between program fragments would record program-state equality.

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Converting Imperative Code to PEGs

In this chapter we describe how programs written in an imperative style can be transformed to work within our PEG-based optimization system. We first define a minimal imperative language (Section 7.1) and then present ML-style pseudocode for converting any program written in this language into a PEG (Section 7.2). Next, we present a formal account of the conversion process using type-directed translation rules in the style of [51] (Section 7.3). Finally, we present an algorithm for converting CFGs to PEGs (Section 7.5).

7.1 The SIMPLE Programming Language

We present our algorithm for converting to the PEG representation using a simplified source language. In particular, we use the SIMPLE programming language, the grammar of which is shown in Figure 7.1. A SIMPLE program contains a single function `main`, which declares parameters with their respective types, a body which uses these variables, and a return type. There is a special variable `retvar`, the value of which is returned by `main` at the end of execution. SIMPLE programs may have an arbitrary set of primitive operations on an arbitrary set of types; we only require that there is a boolean type for conditionals (which makes the translation simpler). Statements in SIMPLE programs have four forms: statement sequencing (using semicolon), variable assignment

$$\begin{aligned} p &::= \texttt{main}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ \{s\} \\ s &::= s_1; s_2 \mid x := e \mid \textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\} \mid \textbf{while}\ (e)\ \{s\} \\ e &::= x \mid op(e_1, \ldots, e_n) \end{aligned}$$

Figure 7.1. Grammar for SIMPLE programs

$\boxed{\vdash p \quad \text{(programs)}}$

$$\text{Type-Prog}\ \frac{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash s : \Gamma \qquad \Gamma(\texttt{retvar}) = \tau}{\vdash \texttt{main}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ \{s\}}$$

$\boxed{\Gamma \vdash s : \Gamma' \quad \text{(statements)}}$

$$\text{Type-Seq}\ \frac{\Gamma \vdash s_1 : \Gamma' \qquad \Gamma' \vdash s_2 : \Gamma''}{\Gamma \vdash s_1; s_2 : \Gamma''} \qquad \text{Type-Asgn}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash x := e : (\Gamma, x : \tau)}$$

$$\text{Type-If}\ \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s_1 : \Gamma' \qquad \Gamma \vdash s_2 : \Gamma'}{\Gamma \vdash \textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\} : \Gamma'} \qquad \text{Type-While}\ \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s : \Gamma}{\Gamma \vdash \textbf{while}\ (e)\ \{s\} : \Gamma}$$

$$\text{Type-Sub}\ \frac{\Gamma \vdash s : \Gamma' \qquad \Gamma'' \subseteq \Gamma'}{\Gamma \vdash s : \Gamma''}$$

$\boxed{\Gamma \vdash e : \tau \quad \text{(expressions)}}$

$$\text{Type-Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{Type-Op}\ \frac{op : (\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma \vdash e_1 : \tau_1\ \ldots\ \Gamma \vdash e_n : \tau_n}{\Gamma \vdash op(e_1, \ldots, e_n) : \tau}$$

Figure 7.2. Typing rules for SIMPLE programs

(the variable implicitly inherits the type of the expression), if-then-else branches, and while loops. Expressions in SIMPLE programs are either variables or primitive operations (such as addition). Constants in SIMPLE are nullary primitive operations.

The typing rules for SIMPLE programs are shown in Figure 7.2. There are three kinds of judgements: (1) judgement $\vdash p$ (where p is a program) states that p is well typed; (2) judgement $\Gamma \vdash s : \Gamma'$ (where s is a statement) states that starting with context Γ, after s the context will be Γ'; and (3) judgement $\Gamma \vdash e : \tau$ (where e is an expression) states that in type context Γ, expression e has type τ.

For program judgements, there is only one rule, Type-Prog, which ensures that the statement inside of `main` is well typed; $\Gamma(\texttt{retvar}) = \tau$ ensures that the return value of `main` has type τ. For statement judgements, Type-Seq simply sequences the typing context through two statements. Type-Asgn replaces the binding for x with the type of the expression assigned into x (if Γ contains a binding for x, then $(\Gamma, x : \tau)$ is Γ with the binding for x replaced with τ; if Γ does not contain a binding for x, then $(\Gamma, x : \tau)$ is Γ extended with a binding for x). The rule Type-If requires the context after each branch to be the same. The rule Type-While requires the context before and after the body to be the same, specifying the loop-induction variables. The rule Type-Sub allows the definition of variables to be "forgotten", enabling the use of temporary variables in branches and loops.

7.2 Translating SIMPLE Programs to PEGs

Here we use ML-style pseudocode to describe the translation from SIMPLE programs to PEGs. Figure 7.3 shows the entirety of the algorithm, which uses a variety of simple types and data structures, which we explain first. Note that if we use SIMPLE programs to instantiate our equality saturation approach described in Figure 5.1, then the ConvertToIR function from Figure 5.1 is implemented using a call to TranslateProg.

In the pseudocode (and in Chapters 7 and 8), we use the notation $\overline{a}(n_1, \ldots, n_k)$ to represent a PEG node with label a and children n_1 through n_k. Whereas previously the distinction between creating PEG nodes and applying functions was clear from context, in a computational setting like pseudocode we want to avoid confusion between, for example, applying negation in the pseudocode $\neg(\ldots)$ or creating a PEG node labeled with negation $\overline{\neg}(\ldots)$. For parameter nodes, there cannot be any confusion because when we write $\mathit{param}(x)$, param is actually not a function – instead $\mathit{param}(x)$ as a whole is a label. Still, to be consistent in the notation, we use $\overline{\mathit{param}}(x)$ for constructing parameter

```
function TranslateProg$(p : Prog) : N =$
  let $m =$ InitMap$(p.params, \lambda x.\,\overline{\text{param}}(x))$
  in TS$(p.body, m, 0)(\texttt{retvar})$
```

```
function TS$(s : Stmt, \Psi : map[V,N], \ell : \mathbb{N}) : map[V,N] =$
  match $s$ with
    “$s_1; s_2$” $\Rightarrow$ TS$(s_2, \text{TS}(s_1, \Psi, \ell), \ell)$
    “$x := e$” $\Rightarrow \Psi[x \mapsto \text{TE}(e, \Psi)]$
    “if $(e)$ $\{s_1\}$ else $\{s_2\}$” $\Rightarrow$ PHI$(\text{TE}(e,\Psi), \text{TS}(s_1, \Psi, \ell), \text{TS}(s_2, \Psi, \ell))$
    “while $(e)$ $\{s\}$” $\Rightarrow$
      let $vars =$ Keys$(\Psi)$
      let $\Psi_t =$ InitMap$(vars, \lambda v.\,\text{TemporaryNode}(v))$
      let $\Psi' =$ TS$(s, \Psi_t, \ell + 1)$
      let $\Psi_\theta =$ THETA$_{\ell+1}(\Psi, \Psi')$
      let $\Psi'_\theta =$ InitMap$(vars, \lambda v.\,\text{FixpointTemps}(\Psi_\theta, \Psi_\theta(v)))$
      in EVAL$_{\ell+1}(\Psi'_\theta, \overline{pass_{\ell+1}}(\overline{\neg}(\text{TE}(e, \Psi'_\theta))))$
```

```
function TE$(e : Expr, \Psi : map[V,N]) : N =$
  match $e$ with
    “$x$” $\Rightarrow \Psi(x)$
    “$op(e_1, \ldots, e_k)$” $\Rightarrow \overline{op}(\text{TE}(e_1, \Psi), \ldots, \text{TE}(e_k, \Psi))$
```

```
function PHI$(n : N, \Psi_1 : map[V,N], \Psi_2 : map[V,N]) : map[V,N] =$
  Combine$(\Psi_1, \Psi_2, \lambda\ t\ f\ .\ \overline{\phi}(n, t, f))$

function THETA$_{\ell:\mathbb{N}}(\Psi_1 : map[V,N], \Psi_2 : map[V,N]) : map[V,N] =$
  Combine$(\Psi_1, \Psi_2, \lambda\ b\ n\ .\ \overline{\theta_\ell}(b, n))$

function EVAL$_{\ell:\mathbb{N}}(\Psi : map[V,N], n : N) : map[V,N] =$
  InitMap(Keys$(\Psi), \lambda v\ .\ \overline{eval_\ell}(\Psi(v), n)$)

function Combine$(m_1 : map[a,b], m_2 : map[a,c], f : b * c \to d) : map[a,d] =$
  InitMap(Keys$(m_1) \cap$ Keys$(m_2), \lambda k.f(m_1[k], m_2[k])$))
```

Figure 7.3. ML-style pseudocode for converting SIMPLE programs to PEGs

nodes.

We introduce the concept of a *node context* Ψ, which is a set of bindings of the form $x : n$, where x is a SIMPLE variable, and n is a PEG node. A node context states, for each variable x, the PEG node n that represents the current value of x. We use $\Psi(x) = n$ as shorthand for $(x : n) \in \Psi$. Aside from using node contexts here, we will also use them later in our type-directed translation rules (Section 7.3). For our pseudocode, we implement node contexts as an immutable map data structure that has the following operations defined on it

Map initialization The InitMap function is used to create maps. Given a set K of keys and a function f from K to D, InitMap creates a map of type $map[K,D]$ containing, for every element $k \in K$, an entry mapping k to $f(k)$.

Map keys Given a map m, $\text{Keys}(m)$ returns the set of keys of m.

Map read Given a map m and a key $k \in \text{Keys}(m)$, $m(k)$ returns the value associated with key k in m.

Map update Given a map m, $m[k \mapsto d]$ returns a new map in which key k has been updated to map to d.

The pseudocode also uses the types *Prog*, *Stmt*, *Expr*, and *V* to represent SIMPLE programs, statements, expressions, and variables. Given a program p, $p.params$ is a list of its parameter variables, and $p.body$ is its body statement. We use syntax-based pattern matching to extract information from *Stmt* and *Expr* types (as shown on lines 6 and 7 for statements, and lines 18 and 19 for expressions).

Expressions We explain the pieces of this algorithm one by one, starting with the TE function on line 16. This function takes a SIMPLE expression e and a node context Ψ

and returns the PEG node corresponding to e. There are two cases, based on what type of expression e is. Line 18 states that if e is a reference to variable x, then we simply ask Ψ for its current binding for x. Line 19 states that if e is the evaluation of operator op on arguments $e_1, \dots, e_k$, then we recursively call TE on each e_i to get n_i, and then create a new PEG node labeled op that has child nodes $n_1, \dots, n_k$.

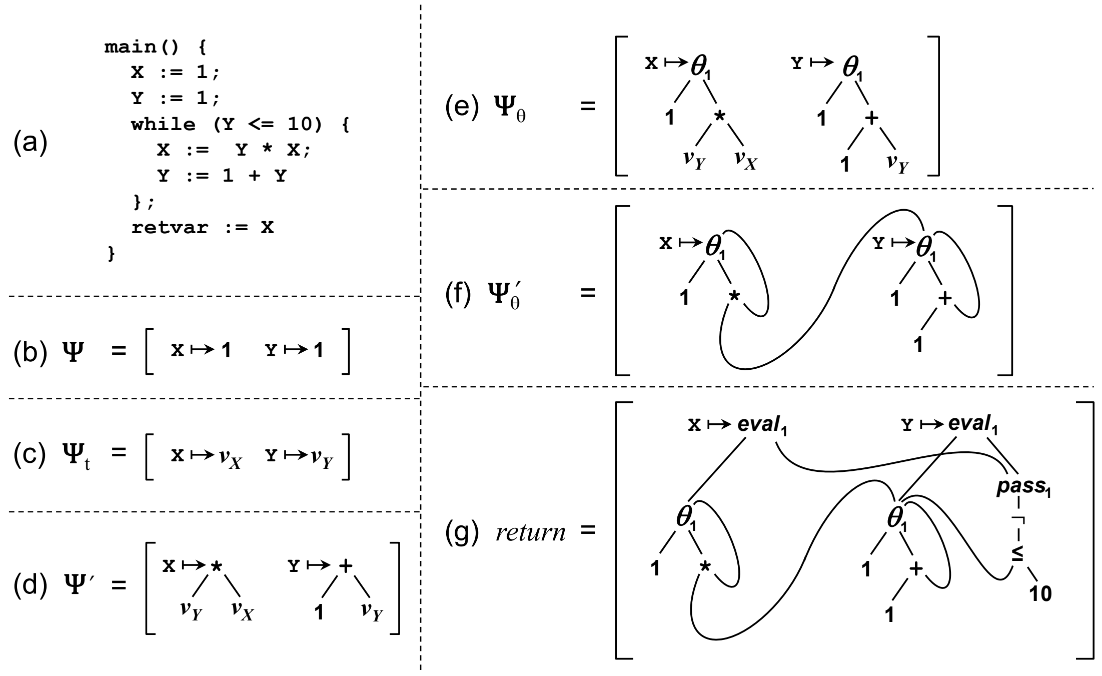
Statements Next we explore the TS function on line 4, which takes a SIMPLE statement s, a node context Ψ, and a loop depth ℓ and returns a new node context that represents the changes s made to Ψ. There are four cases, based on the four statement types.

Line 6 states that a sequence $s_1; s_2$ is simply the result of translating s_2 using the node context that results from translating s_1. Line 7 states that for an assignment $x := e$, we simply update the current binding for x with the PEG node that corresponds to e, which is computed with a call to TE.

Line 8 handles if-then-else statements by introducing ϕ nodes. We recursively produce updated node contexts Ψ_1 and Ψ_2 for statements s_1 and s_2 respectively, and compute the PEG node that represents the guard condition, call it n_c. We then create PEG ϕ nodes by calling the PHI function defined on line 20. This function takes the guard node n_c and the two node contexts Ψ_1 and Ψ_2 and creates a new ϕ node in the PEG for each variable that is defined in both node contexts. The true child for each ϕ node is taken from Ψ_1, and the false child is taken from Ψ_2, while all of them share the same guard node n_c. Note that this is slightly inefficient in that it will create ϕ nodes for all variables defined before the if-then-else statement, whether they are modified by it or not. These can be easily removed, however, by applying the rewrite $\phi(C,A,A) = A$.

Finally we come to the most complicated case on line 9, which handles while loops. In line 10 we extract the set of all variables defined up to this point, in the set *vars*.

We allocate a temporary PEG node for each item in *vars* on line 11, and bind them together in the node context Ψ_t. We use TemporaryNode(v) to refer to a temporary PEG node named v, which is a new kind of node that we use only for the conversion process. We then recursively translate the body of the while loop using the context full of temporary nodes on line 12. In the resulting context Ψ', the temporary nodes act as placeholders for loop-varying values. Note that here is the first real use of the loop-depth parameter ℓ, which is incremented by 1 since the body of this loop will be at a higher loop depth than the code before the loop. For every variable in *vars*, we create $\theta_{\ell+1}$ nodes using the THETA function defined on line 22. This function takes node contexts Ψ and Ψ' which have bindings for the values of each variable before and during the loop respectively. The binding for each variable in Ψ becomes the first child of the θ (the base case), and the binding in Ψ' becomes the second child (the inductive case). Unfortunately, the θ expressions we just created are not yet accurate, because the second child of each θ node is defined in terms of temporary nodes. The correct expression should replace each temporary node with the new θ node that corresponds to that temporary node's variable to "close the loop" of each θ node. That is the purpose of the FixpointTemps function called on line 14. For each variable $v \in vars$, FixpointTemps will rewrite $\Psi_\theta(v)$ by replacing any edges to TemporaryNode(x) with edges to $\Psi_\theta(x)$, yielding a new node context Ψ'_θ. Now that we have created the correct θ nodes, we merely need to create the *eval* and *pass* nodes to go with them. Line 15 does this, first by creating the $pass_{\ell+1}$ node which takes the break condition expression as its child. The break condition is computed with a call to TE on e, using node context Ψ'_θ since it may reference some of the newly created θ nodes. The last step is to create *eval* nodes to represent the values of each variable after the loop has terminated. This is done by the EVAL function defined on line 24. This function takes the node context Ψ'_θ and the $pass_{\ell+1}$ node and creates a new $eval_{\ell+1}$ node for each variable in *vars*. This final



```
main() {
  X := 1;
  Y := 1;
  while (Y <= 10) {
    X :=  Y * X;
    Y := 1 + Y
  };
  retvar := X
}
```

Figure 7.4. Steps of the translation process: (a) shows a SIMPLE program computing the factorial of 10; (b) through (f) show the contents of the variables in TS when processing the `while` loop; and (g) shows the return value of TS when processing the `while` loop

node context mapping each variable to an *eval* is the return value of TS. Note that, as in the case of if-then-else statements, we introduce an inefficiency here by replacing all variables with *eval*s, not just the ones that are modified in the loop. For any variable v that was bound to node n in Ψ and not modified by the loop, its binding in the final node context would be $eval_{\ell+1}(T, pass_{\ell+1}(C))$, where C is the guard condition node and $T = \theta_{\ell+1}(n, T)$ (i.e. the θ node has a direct self loop). We can easily remove the spurious nodes by applying a rewrite to replace the *eval* node with n.

Programs The TranslateProg function on line 1 is the top-level call to convert an entire SIMPLE program to a PEG. It takes a SIMPLE program p and returns the root node of the translated PEG. It begins on line 2 by creating the initial node context which contains bindings for each parameter variable. The nodes that correspond to the parameters are opaque parameter nodes that simply name the parameter variable they represent. Using this node context, we translate the body of the program starting at loop depth 0 on line 3. This will yield a node context that has PEG expressions for the final values of all the variables in the program. Hence, the root of our translated PEG will be the node that is bound to the special return variable `retvar` in this final node context.

Example We illustrate how the translation process works on the SIMPLE program from Figure 7.4(a), which computes the factorial of 10. After processing the first two statements, both `X` and `Y` are bound to the PEG node 1. Then TS is called on the `while` loop, at which point Ψ maps both `X` and `Y` to 1. Figures 7.4(b) through 7.4(g) show the details of processing the `while` loop. In particular, (b) through (f) show the contents of the variables in TS, and (g) shows the return value of TS. Note that in (g) the node labeled *i* corresponds to the *pass* node created on line 15 in TS. After the loop is processed, the assignment to `retvar` simply binds `retvar` to whatever `X` is bound to in Figure 7.4(g).

$$\boxed{\vdash p \;\;\triangleright\;\; n \quad \text{(programs)}}$$

$$\text{Trans-Prog}\;\frac{\begin{array}{c}\{x_1:\tau_1,\dots,x_k:\tau_k\}\vdash s:\Gamma \;\;\triangleright\;\; \{x_1:\overline{param}(x_1),\dots,x_k:\overline{param}(x_k)\}\rightsquigarrow_0 \Psi\\ n=\Psi(\texttt{retvar}) \text{ (well defined because } \Gamma(\texttt{retvar})=\tau)\end{array}}{\vdash \texttt{main}(x_1:\tau_1,\dots,x_k:\tau_k):\tau\;\{s\} \;\;\triangleright\;\; n}$$

$$\boxed{\Gamma\vdash s:\Gamma' \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell \Psi' \quad \text{(statements)}}$$

$$\text{Trans-Seq}\;\frac{\Gamma\vdash s_1:\Gamma' \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell \Psi' \qquad \Gamma'\vdash s_2:\Gamma'' \;\;\triangleright\;\; \Psi'\rightsquigarrow_\ell \Psi''}{\Gamma\vdash s_1;s_2:\Gamma'' \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell \Psi''}$$

$$\text{Trans-Asgn}\;\frac{\Gamma\vdash e:\tau \;\;\triangleright\;\; \Psi\rightsquigarrow n}{\Gamma\vdash x:=e:(\Gamma,x:\tau) \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell (\Psi,x:n)}$$

$$\text{Trans-If}\;\frac{\begin{array}{c}\Gamma\vdash e:\texttt{bool} \;\;\triangleright\;\; \Psi\rightsquigarrow n\\ \Gamma\vdash s_1:\Gamma' \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell \Psi_1 \qquad \Gamma\vdash s_2:\Gamma' \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell \Psi_2\\ \{x:n_{(x,1)}\}_{x\in\Gamma'}=\Psi_1 \qquad\qquad \{x:n_{(x,2)}\}_{x\in\Gamma'}=\Psi_2\\ \Psi'=\{x:\overline{\phi}(n,n_{(x,1)},n_{(x,2)})\}_{x\in\Gamma'}\end{array}}{\Gamma\vdash \textbf{if }(e)\;\{s_1\}\textbf{ else }\{s_2\}:\Gamma' \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell \Psi'}$$

$$\text{Trans-While}\;\frac{\begin{array}{c}\Gamma\vdash e:\texttt{bool} \;\;\triangleright\;\; \Psi\rightsquigarrow n\\ \Psi=\{x:v_x\}_{x\in\Gamma}\text{ each } v_x \text{ fresh} \qquad \ell'=\ell+1 \qquad \Gamma\vdash s:\Gamma \;\;\triangleright\;\; \Psi\rightsquigarrow_{\ell'} \Psi'\\ \{x:n_x\}_{x\in\Gamma}=\Psi_0 \qquad\qquad\qquad \{x:n'_x\}_{x\in\Gamma}=\Psi'\\ \Psi_\infty=\{x:\overline{eval_{\ell'}}(v_x,\overline{pass_{\ell'}}(\overline{\neg}(n)))\}_{x\in\Gamma}\text{ with each } v_x \text{ unified with } \overline{\theta_{\ell'}}(n_x,n'_x)\end{array}}{\Gamma\vdash \textbf{while }(e)\;\{s\}:\Gamma \;\;\triangleright\;\; \Psi_0\rightsquigarrow_\ell \Psi_\infty}$$

$$\text{Trans-Sub}\;\frac{\begin{array}{c}\Gamma\vdash s:\Gamma' \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell \Psi'\\ \{x:n_x\}_{x\in\Gamma'}=\Psi' \qquad \Psi''=\{x:n_x\}_{x\in\Gamma''}\text{ (well defined because } \Gamma''\subseteq\Gamma')\end{array}}{\Gamma\vdash s:\Gamma'' \;\;\triangleright\;\; \Psi\rightsquigarrow_\ell \Psi''}$$

$$\boxed{\Gamma\vdash e:\tau \;\;\triangleright\;\; \Psi\rightsquigarrow n \quad \text{(expressions)}}$$

$$\text{Trans-Var}\;\frac{n=\Psi(x)\text{ (well defined because } \Gamma(x)=\tau)}{\Gamma\vdash x:\tau \;\;\triangleright\;\; \Psi\rightsquigarrow n}$$

$$\text{Trans-Op}\;\frac{\Gamma\vdash e_1:\tau_1 \;\;\triangleright\;\; \Psi\rightsquigarrow n_1 \qquad \dots \qquad \Gamma\vdash e_k:\tau_k \;\;\triangleright\;\; \Psi\rightsquigarrow n_k}{\Gamma\vdash op(e_1,\dots,e_k):\tau \;\;\triangleright\;\; \Psi\rightsquigarrow \overline{op}(n_1,\dots,n_k)}$$

Figure 7.5. Type-directed translation from SIMPLE programs to PEGs

7.3 Type-Directed Translation

In this section we formalize the translation process described by the pseudocode implementation with a type-directed translation from SIMPLE programs to PEGs, in the style of [51]. The type-directed translation in Figure 7.5 is more complicated than the implementation in Figure 7.3, but it makes it easier to prove the correctness of the translation. For example, the implementation uses maps from variables to PEG nodes, and at various points queries these maps (for example, line 18 in Figure 7.3 queries the Ψ map for variable x). The fact that these map operations never fail relies on implicit properties which are tedious to establish in Figure 7.3, as they rely on the fact that the program being translated is well typed. In the type-directed translation, these properties are almost immediate since the translation operates on an actual proof that the program is well typed.

In fact, the rules in Figure 7.5 are really representations of each case in a constructive total deterministic function defined inductively on the proof of well-typedness. Thus when we use the judgement $\Gamma \vdash e : \tau \;\; \triangleright \;\; \Psi \leadsto n$ as an assumption, we are simply binding n to the result of this constructive function applied to the proof of $\Gamma \vdash e : \tau$ and the PEG context Ψ. Likewise, we use the judgement $\Gamma \vdash s : \Gamma' \;\; \triangleright \;\; \Psi \leadsto_{\ell} \Psi'$ to bind Ψ' to the result of the constructive function applied to the proof of $\Gamma \vdash s : \Gamma'$ and the PEG context Ψ. Here we explain how this type-directed translation works.

Expressions The translation process for an expression e takes two inputs: (1) a derivation showing the type correctness of e, and (2) a node context Ψ. The translation process produces one output, which is the node n that e translates to. We formalize this with a judgement $\Gamma \vdash e : \tau \;\; \triangleright \;\; \Psi \leadsto n$, which states that from a derivation of $\Gamma \vdash e : \tau$, and a node context Ψ (stating what node to use for each variable), the translation produces node n for expression e. For example, consider the Trans-Op rule, which is used for a primitive operation expression. The output of the translation process is a new PEG node with label op, where the argument nodes $n_1 \dots n_k$ are determined by translating the argument expressions $e_1 \dots e_k$.

The Trans-Var rule returns the PEG node associated with the variable x in Ψ. The definition $n = \Psi(x)$ is well defined because we maintain the invariant that the judgement $\Gamma \vdash e : \tau \;\; \triangleright \;\; \Psi \rightsquigarrow n$ is only used with contexts Γ and Ψ that are defined on precisely the same set of variables. Thus, because the Type-Var rule requires $\Gamma(x) = \tau$, Γ must be defined on x and so we know Ψ is also defined on x.

Note that a concrete implementation of the translation, like the one in Figure 7.3, would explore a derivation of $\Gamma \vdash e : \tau \;\; \triangleright \;\; \Psi \rightsquigarrow n$ bottom-up: the translation starts at the bottom of the derivation tree and makes recursive calls to itself, each recursive call corresponding to a step up in the derivation tree. Also note that there is a close relation between the rules in Figure 7.5 and those in Figure 7.2. In particular, the formulas on the left of the $\triangleright$ correspond directly to the typing rules from Figure 7.2.

Statements The translation for a statement s takes as input a derivation of the type correctness of s and a node context capturing the translation that has been performed up to s, and returns the node context to be used in the translation of statements following s. We formalize this with a judgement $\Gamma \vdash s : \Gamma' \;\; \triangleright \;\; \Psi \rightsquigarrow_\ell \Psi'$, which states that from a derivation of $\Gamma \vdash s : \Gamma'$ and a node context Ψ (stating what node to use for each variable in s), the translation produces an updated node context Ψ' after statement s (ignore ℓ for now). For example, the rule Trans-Asgn updates the node context to map variable x to the node n resulting from translating e (which relies on the fact that e is well typed in type context Γ).

Again, we maintain the invariant that in all the derivations we explore, the judgement $\Gamma \vdash s : \Gamma' \;\; \triangleright \;\; \Psi \rightsquigarrow \Psi'$ is only used with contexts Γ and Ψ that are defined on precisely the same set of variables, and furthermore the resulting contexts Γ' and Ψ' will always be defined on the same set of variables (although potentially different from Γ and Ψ). It is fairly obvious that the rules preserve this invariant, although Trans-Sub relies on the fact that Γ'' must be a subcontext of Γ'. The Trans-Seq and Trans-Asgn rules are self explanatatory, so below we discuss the more complicated rules for control flow.

The rule Trans-If describes how to translate if-then-else statements in SIMPLE programs to ϕ nodes in PEGs. First, it translates the boolean guard expression e to a node n which will later be used as the condition argument for each ϕ node. Then it translates the statement s_1 for the "true" branch, producing a node context Ψ_1 assigning each live variable after s_1 to a PEG node representing its value after s_1. Similarly, it translates s_2 for the "false" branch, producing a node context Ψ_2 representing the "false" values of each variable. Due to the invariant we maintain, both Ψ_1 and Ψ_2 will be defined on the same set of variables as Γ'. For each x defined in Γ', we use the name $n_{(x,1)}$ to represent the "true" value of x after the branch (taken from Ψ_1) and $n_{(x,2)}$ to represent the "false" value (taken from Ψ_2). Finally, the rule constructs a node context Ψ' which assigns each variable x defined in Γ' to the node $\overline{\phi}(n, n_{(x,1)}, n_{(x,2)})$, indicating that after the if-then-else statement the variable x has "value" $n_{(x,1)}$ if n evaluates to true and $n_{(x,2)}$ otherwise. Furthermore, this process maintains the invariant that the type context Γ' and node context Ψ' are defined on exactly the same set of variables.

The last rule, Trans-While, describes how to translate while loops in SIMPLE programs to combinations of θ, *eval*, and *pass* nodes in PEGs. The rule starts by creating a node context Ψ which assigns to each variable x defined in Γ a fresh temporary variable node v_x. The clause $\ell' = \ell + 1$ is used to indicate that the body of the loop is being translated at a higher loop depth. In general, the ℓ subscript in the notation $\Psi \rightsquigarrow_\ell \Psi'$ indicates the loop depth of the translation. Thus, the judgement $\Gamma \vdash s : \Gamma' \;\; \triangleright \;\; \Psi \rightsquigarrow_{\ell'} \Psi'$

translates the body s of the loop at a higher loop depth to produce the node context Ψ'. The nodes in Ψ' are in terms of the temporary nodes v_x in Γ and essentially represent how variables change in each iteration of the loop. Each variable x defined in Γ has a corresponding node n_x in the node context Ψ_0 from before the loop, again due to the invariant we maintain that Γ and Ψ are always defined on the same set of variables. This invariant also guarantees that each variable x defined in Γ also has a corresponding node n'_x in the node context Ψ'. Thus, for each such variable, n_x provides the base value and n'_x provides the iterative value, which can now be combined using a θ node. To this end, we unify the temporary variable node v_x with the node $\overline{\theta_{\ell'}}(n_x, n'_x)$ to produce a recursive expression which represents the value of x at each iteration of the loop. Lastly, the rule constructs the final node context Ψ_∞ by assigning each variable x defined in Γ to the node $\overline{eval_{\ell'}}(v_x, \overline{pass_{\ell'}}(\overline{\neg}(n)))$ (where v_x has been unified to produce the recursive expression for x). The node $\overline{\neg}(n)$ represents the break condition of the loop; thus $\overline{pass_{\ell'}}(\overline{\neg}(n))$ represents the number of times the loop iterates. Note that the same *pass* node is used for each variable, whereas each variable gets its own θ node. In this manner, the rule Trans-While translates while loops to PEGs, and furthermore preserves the invariant that the type context Γ and node context Ψ_∞ are defined on exactly the same set of variables.

Programs Finally, the rule Trans-Prog shows how to use the above translation technique in order to translate an entire SIMPLE program. It creates a node context with a PEG parameter node for each parameter to `main`. It then translates the body of `main` at loop depth 0 to produce a node context Ψ. Since the return `retvar` is guaranteed to be in the final context Γ, the invariant that Γ and Ψ are always defined on the same variables ensures that there is a node n corresponding to `retvar` in the final node context Ψ. This PEG node n represents the entire SIMPLE program.

Translation vs. Pseudocode The pseudocode in Figure 7.3 follows the rules from Figure 7.5 very closely. Indeed, the code can be seen as using the rules from the type-directed translation to find a derivation of $\vdash p \ \triangleright\ n$. The translation starts at the end of the derivation tree and moves up from there. The entry point of the pseudocode is TranslateProg, which corresponds to rule Trans-Prog, the last rule to be used in a derivation of $\vdash p \ \triangleright\ n$. TranslateProg calls TS, which corresponds to finding a derivation for $\Gamma \vdash s : \Gamma' \ \triangleright\ \Psi \rightsquigarrow_{\ell} \Psi'$. Finally, TS calls TE, which corresponds to finding a derivation for $\Gamma \vdash e : \tau \ \triangleright\ \Psi \rightsquigarrow n$. Each pattern-matching case in the pseudocode corresponds to a rule from the type-directed translation.

The one difference between the pseudocode and the type-directed translation is that in the judgements of the type-directed translation, one of the inputs to the translation is a derivation of the type correctness of the expression/statement/program being translated, whereas the pseudocode does not manipulate any derivations. This can be explained by a simple erasure optimization in the pseudocode: because of the structure of the typing rules for SIMPLE (in particular there is only one rule per statement kind), the implementation does not need to inspect the entire derivation – it only needs to look at the final expression/statement/program in the type derivation (which is precisely the expression/statement/program being translated). It is still useful to have the derivation expressed formally in the type-directed translation, as it makes the proof of correctness more direct. Furthermore, there are small changes that can be made to the SIMPLE language that prevent the erasure optimization from being performed. For example, if we add subtyping and implicit coercions, and we want the PEG translation process to make coercions explicit, then the translation process would need to look at the type derivation to see where the subtyping rules are applied.

Because the type-directed translation in Figure 7.5 is essentially structural induction on the proof that the SIMPLE program is well typed, we can guarantee that its implementation in Figure 7.3 terminates. Additionally, because of the invariants we maintain in the type-directed translation, we can guarantee that the implementation always successfully produces a translation. We discuss the correctness guarantees provided by the translation below.

7.4 Preservation of Semantics

While SIMPLE is a standard representation of programs, PEGs are far from standard. Furthermore, the semantics of PEGs are even less so, especially since the node representing the returned value is the first to be "evaluated". Thus, it is natural to ask whether the translation above preserves the semantics of the specified programs. We begin by defining the semantics of SIMPLE programs, and go on to examine their relationship to the semantics of PEGs produced by our algorithm.

Here we define the evaluation functions $[\![\cdot]\!]$, which implement the operational semantics of SIMPLE programs. We do not give full definitions, since these are standard. These functions are defined in terms of an *evaluation context* Σ, which is a map assigning values v to program variables x.

Definition 7.1 (Semantics of expressions)**.** *For a SIMPLE expression e we define $[\![e]\!]$ to be a partial function from evaluation contexts to values. This represents the standard operational semantics for SIMPLE expressions. For a given Σ, $[\![e]\!](\Sigma)$ returns the result of evaluating e, using the values of variables given by Σ.*

Definition 7.2 (Semantics of statements)**.** *For a SIMPLE statement s we define $[\![s]\!]$ to be a partial function from evaluation contexts to evaluation contexts. This represents the standard operational semantics for SIMPLE statements. For a given Σ, $[\![s]\!](\Sigma)$ returns the evaluation context that results from executing s in context Σ. If s does not terminate when started in Σ, then $[\![s]\!](\Sigma)$ is not defined.*

Definition 7.3 (Semantics of programs)**.** *For a SIMPLE program p, guaranteed to be of the form* $\mathbf{main}(x_1 : \tau_1, \ldots, x_k : \tau_k)\{s\}$*, and an evaluation context* Σ *mapping each* x_i *to an appropriately typed value, we define* $[\![p]\!](\Sigma)$ *as* $[\![s]\!](\Sigma)(\texttt{retvar})$.

We will use these functions in our discussion below. For the translation defined in Section 7.3 we have proven the following theorem (recall the substitution notation $n[x \mapsto v]$ from Section 6.1).

Theorem 7.1. *If* (1) $\vdash \mathbf{main}(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau\ \{s\} \quad \triangleright \quad n$ *holds,* (2) Σ *maps each* x_i *to an appropriately typed value* v_i*, and* (3) $\hat{n}$ *equals* $n[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$*, then* $[\![\mathbf{main}]\!](\Sigma) = v \implies [\![\hat{n}]\!](\lambda \ell.0) = v$.

The above theorem only states that our conversion process is *nearly* semantics-preserving, since it does not perfectly preserve non-termination. In particular, our translation from SIMPLE to PEG discards any PEG nodes which are never used to calculate the return value. Thus, an infinite SIMPLE loop whose value is never used will be removed, changing the termination behavior of the program. In the broader setting beyond SIMPLE, the only case where we would change the termination behavior of the program is if there is an infinite loop has no side effects (aside from non-termination) and does not contribute to the return value of the function. It is important to keep in mind that, since these loops have no side effects (aside from non-termination), they cannot modify the heap or perform I/O. This basically means that these loops are equivalent to a `while(true) { }` loop. Other modern compilers perform similar transformations that remove such IO-less infinite loops that do not contribute to the result [68]. In fact, the newly planned C++ standard allows the implementation to remove such IO-less loops even if termination cannot be proven [10]. Nonetheless, at the end of this section, we give a brief overview of how to preserve non-termination in PEGs.

In this theorem, we use both the function $[\![\cdot]\!]$ defined above for SIMPLE programs and the function $[\![\cdot]\!]$ defined in Equation 6.1 for PEG nodes. Throughout the rest of this section we will mix our uses of the various $[\![\cdot]\!]$ functions, and the reader can disambiguate them based on context. The common intuition is that these functions all implement program semantics and so represent executing the program fragment they are called upon.

We proved the above theorem in full formal detail using the Coq interactive theorem prover [12]. To conduct the proof in Coq we only had to assume the standard axioms for extensional equality of functions and of coinductive types. The machine-checkable Coq proof is available at http://cseweb.ucsd.edu/groups/progsys/peg-coq-proof. Here we present only the major invariants used in this proof without showing the details of why these invariants are preserved.

Given a program `main` with parameters $x_1, \ldots, x_k$, suppose we are given a set of actual values $v_1, \ldots, v_k$ that correspond to the values passed in for those parameters. Then given the PEG G created during the translation of `main`, we can construct a new PEG $\hat{G}$ that is identical to G except that every parameter node for x_i is replaced with a constant node for v_i. Thus, for every node $n \in G$, there is a corresponding node $\hat{n} \in \hat{G}$. Furthermore, this correspondence is natural: $\overline{\theta}$ nodes correspond to $\overline{\theta}$ nodes and so on (except for the parameter nodes which have been explicitly replaced). Similarly, every PEG context Ψ for G has a similarly corresponding PEG context $\hat{\Psi}$ in terms of nodes in $\hat{G}$. We can now phrase our primary invariants in terms of this node correspondence.

In this proof we will rely on the concept of loop invariance of PEG nodes. Earlier in Section 6.1, we defined some simple rules for determining when a node n is invariant with respect to a given loop depth ℓ, which we denote as $invariant_\ell(n)$. These rules are based on the syntax of the PEG rather than the semantics, so we say that the rules detect *syntactic loop invariance*, rather than semantic loop-invariance. Syntactic loop invariance is a useful property since it implies semantic loop invariance, which is in general undecidable. We can generalize the notion of *invariant*$_\ell$ to a PEG context as follows.

Definition 7.4. *Given a PEG context* Ψ *and a loop depth* ℓ*, we say that* Ψ *is syntactically loop invariant with respect to* ℓ *if for each binding* $(x : n) \in \Psi$*,* n *is syntactically loop invariant with respect to* ℓ*. We denote this by* $invariant_\ell(\Psi)$*.*

With this definition in mind, we can express the first two lemmas that will help in our proof of semantics preservation.

Lemma 7.1. *If* $\Gamma \vdash e : \tau \ \ \triangleright \ \ \Psi \rightsquigarrow n$*, then* $\forall \ell.\ invariant_\ell(\hat{\Psi}) \implies invariant_\ell(\hat{n})$

Proof. Proved by induction on the proof of $\Gamma \vdash e : \tau$. □

Lemma 7.2. *For all loop depths* ℓ*, if* $\Gamma \vdash s : \Gamma' \ \ \triangleright \ \ \Psi \rightsquigarrow_\ell \Psi'$*, then*

$$\forall \ell' > \ell.\ invariant_{\ell'}(\hat{\Psi}) \implies \forall \ell' > \ell, invariant_{\ell'}(\hat{\Psi}')$$

Proof. Proved by induction on the proof of $\Gamma \vdash s : \Gamma'$ using Lemma 7.1. □

Using the above lemmas and the fact that *invariant*$(\cdot)$ implies semantic loop-invariance, we can proceed to the critical invariant. First we must introduce the notion of the semantics of a PEG context. Given a PEG context Ψ, there is a unique evaluation context that is induced by Ψ for a given loop vector $\mathbf{i}$. Namely, it is the evaluation context that maps every variable x to the value $[\![\Psi(x)]\!](\mathbf{i})$. This provides a useful relationship between the semantics of PEGs and the semantics of SIMPLE programs.

Definition 7.5. *Given PEG context* Ψ*, we define* $[\![\Psi]\!]$ *to be a partial function from loop vectors to evaluation contexts defined by*

$$\forall \mathbf{i}.\ [\![\Psi]\!](\mathbf{i}) = \{(x : v) \mid v = [\![\Psi(x)]\!]\}$$

Lemma 7.3. *If* $\Gamma \vdash e : \tau \;\; \rhd \;\; \Psi \leadsto n$ *and* $[\![\hat{\Psi}]\!](\mathbf{i}) = \Sigma$*, then*

$$\forall v.\ [\![e]\!](\Sigma) = v \implies \hat{n}(\mathbf{i}) = v$$

Proof. Proved by induction on the proof of $\Gamma \vdash e : \tau$ using Lemma 7.1. □

Lemma 7.4. *For any loop depth* ℓ*, if* (1) $\Gamma \vdash s : \Gamma' \;\; \rhd \;\; \Psi \leadsto_\ell \Psi'$, (2) *for each* $\ell' > \ell$*, invariant*$_{\ell'}(\hat{\Psi})$ *holds, and* (3) $[\![\hat{\Psi}]\!](\mathbf{i}) = \Sigma$ *holds, then*

$$\forall \Sigma'.\ [\![s]\!](\Sigma) = \Sigma' \implies [\![\hat{\Psi}']\!](\mathbf{i}) = \Sigma'$$

Proof. Proved by induction on the proof of $\Gamma \vdash s : \Gamma'$ using Lemmas 7.2 and 7.3. For the **while** case, the proof relies on the fact that syntactic loop invariance of $\hat{\Psi}$ implies semantic loop invariance of $\hat{\Psi}$, which implies that $\hat{\Psi}$ corresponds to Σ at all loop vectors $\mathbf{i}'$ that only differ from $\mathbf{i}$ at loop depth $\ell + 1$. □

Our semantics-preservation theorem is a direct corollary of the above lemma, and so we have shown that the evaluation of a PEG is equivalent to the evaluation of its corresponding SIMPLE program, modulo termination.

Preserving Non-Termination There is a simple change to PEGs that would allow them to preserve non-termination even for loops that do not contribute to the result. In particular, we can adapt the concept of an effect witness [82]. For our task of preserving non-termination, the effect witness will encode the non-termination effect, although one can use a similar strategy for other effects as we formalize in Chapter 9. Any effectful operation must take an effect witness. If the operation might also change the state of the effect, then it must produce an effect witness as output. In SIMPLE, the $\div$ operation could modify our non-termination effect witness, if we choose to encode division-by-zero

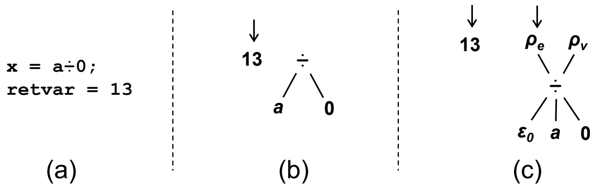


Figure 7.6. Representation of a division by zero: (a) the original source code, (b) the PEG produced without effect witness, and (c) the PEG produced with effect witnesses. The arrows indicate the return values.

as non-termination (since SIMPLE does not contain exceptions). If we added functions to SIMPLE, then function calls would also consume and produce a non-termination effect witness, since the function call could possibly not terminate.

For every loop, there is one *pass* node (although there may be many *eval* nodes), and evaluation of a *pass* node fails to terminate if the condition is never true. As a result, since a *pass* node may fail to terminate, it therefore must take and produce a non-termination effect witness. In particular, we would modify the *pass* node to take two inputs: a node representing the break condition of the loop and a node representing how the state of the effect changes inside the loop. The *pass* node would also be modified to have an additional output, which is the state of the effect at the point when the loop terminates. Then, the translation process is modified to thread this effect witness through all the effectful operations and *pass* nodes, and finally produce a node representing the effect of the entire function. This final effect node is added as an output of the function, along with the node representing the return value.

Whereas before an infinite loop would be elided if it does not contribute to the final return value of the program, now the *pass* node of the loop contributes to the result because the effect witness is threaded through it and returned. This causes the *pass* node to be evaluated, which causes the loop's break condition to be evaluated, which will lead to non-termination since the condition is never true.

We present an example in Figure 7.6 to demonstrate the use of effect witnesses. The SIMPLE code in part (a) shows an attempt to divide by zero, followed by a return value of 13. Let us define $\div$ to not terminate when dividing by 0. Part (b) shows the corresponding PEG without effect witnesses. The arrow indicates the return value, which is 13. Even though this PEG has the nodes for the division by zero, they are not reachable from the return value. Hence, the PEG would be optimized to 13, removing the divide by zero and so changing the termination behavior of the code.

Using a non-termination effect witness can fix this problem by producing the PEG in part (c). The division operator now returns a tuple of $(\mathit{effect}, \mathit{value})$ and the components are fetched using ρ_e and ρ_v respectively. As previously, we have 13 as a return value, but we now have an additional return value: the effect witness produced by the PEG. Since the division is now reachable from the return values, it is not removed anymore, even if the value of the division (the ρ_v node) is never used.

7.5 Converting a CFG to a PEG

We convert a CFG into a PEG in two steps. First, we convert the CFG into an Abstract PEG (A-PEG). Conceptually, an A-PEG is a PEG that operates over program stores rather than individual variables, with nodes representing the basic blocks of the CFG. Figure 7.7(b) shows a sample A-PEG, derived from the CFG in Figure 7.7(a). An A-PEG captures the structure of the original CFG using ϕ, θ, *pass*, and *eval* nodes, but does not capture the flow of individual variables, nor the details of how each basic block operates.

For each basic block n in the CFG, there is a node SE_n in the corresponding A-PEG that represents the execution of the basic block (*SE* stands for Symbolic Evaluator): given a store at the input of the basic block, SE_n returns the store at the output. For basic blocks that have multiple CFG successors, meaning that the last instruction in the

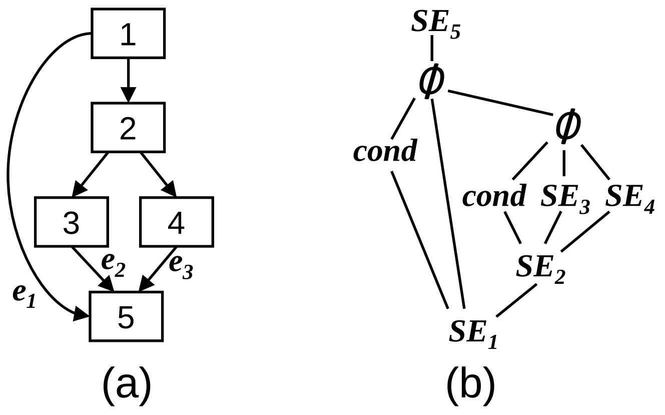


Figure 7.7. Sample CFG and corresponding A-PEG

block is a branch, we assume that the store returned by *SE* contains a specially named boolean variable whose value indicates which way the branch will go. The function *cond* takes a program store and selects this specially named variable from it. As a result, for a basic block *n* that ends in a branch, $cond(SE_n)$ is a boolean stating which way the branch should go.

Once we have an A-PEG, the translation from A-PEG to PEG is simple – all that is left to do is expand the A-PEG to the level of individual variables by replacing each SE_n node with a dataflow representation of the instructions in block *n*. For example, in Figure 7.7, if there were two variables being assigned in all the basic blocks, then the PEG would essentially contain two structural copies of the A-PEG, one copy for each variable.

Our algorithm for converting a CFG into an A-PEG starts with a reducible CFG (all CFGs produced from valid Java code are reducible, and furthermore, if a CFG is not reducible, it can be transformed to an equivalent reducible one at the cost of some node duplication [58]). Using standard techniques, we identify loops, and for each loop we identify (1) the loop-header node, which is the first node that executes when the loop begins (this node is guaranteed to be unique because the graph is reducible), (2) back edges, which are edges that connect a node inside the loop to the loop-header node, and (3) break edges, which are edges that connect a node inside the loop to a node that is not in the loop.

From the CFG we build what is called a *forward flow graph* (FFG), which is an acyclic version of the CFG. In particular, the FFG contains all the nodes from the CFG, plus a node n' for each loop-header node n; it also contains all the edges from the CFG, except that any back edge connected to a loop header n is instead connected to n'.

We use N to denote the set of nodes in the FFG, and E the set of edges, with $root\_edge$ the edge from which the CFG is initially entered. For any $n \in N$, $in(n)$ and $out(n)$ are the set of incoming and outgoing edges of n. If n is a basic block that ends in a branch statement, we use $out_{\mathbf{true}}(n)$ and $out_{\mathbf{false}}(n)$ for the true and false outgoing edge of n. We use $a \overset{*}{\to} b$ to represent that there is a path in the FFG from the node (or edge) a to the node (or edge) b. We identify a loop by its loop-header node ℓ, and for any $n \in N$, we use $loops(n) \subseteq N$ to denote the set of loops that n belongs to. Precisely, $\ell \in loops(n)$ holds whenever there is a directed cycle in the CFG containing both ℓ and n that does not pass through nodes that can reach ℓ in the FFG.

Our conversion algorithm from CFG to A-PEG is shown in Figure 7.8. We describe each function in turn.

ComputeAPEG Once the FFG is constructed, our conversion algorithm calls the ComputeAPEG function. Throughout the rest of the description we assume that the FFG and CFG are globally accessible. ComputeAPEG starts by creating, for each node n in the CFG (line 1), a globally accessible A-PEG node SE_n (line 2), and a globally accessible A-PEG node c_n (line 3). The conversion algorithm then sets the input of each SE_n node to the A-PEG expression computed by ComputeInputs(n) (lines 4 and 5).

```
1:  function ComputeAPEG()
2:  for each CFG node $n$ do
3:      let global $SE_n$ = create A-PEG node labeled “$SE_n$”
4:      let global $c_n = \overline{cond}(SE_n)$
5:  for each CFG node $n$ do
6:      set child of $SE_n$ to ComputeInputs($n$)
7:  return resulting A-PEG
```

```
8:  function ComputeInputs($n : N$)
9:  let $in\_edges = in(n)$
10: let $value\_fn = \lambda e : in\_edges\ .\ SE_{src(e)}$
11: let $result = \text{Decide}(root\_edge, in\_edges, value\_fn, loops(n))$
12: if $n$ is a loop-header node then
13:     let $i = |loops(n)|$
14:     let $result = \overline{\theta_i}(result, \text{ComputeInputs}(n'))$
15: return $result$
```

```
16: function Decide($source : E,\ \mathscr{E} : 2^E,\ value : \mathscr{E} \to N_{\text{A-PEG}},\ \mathscr{L} : 2^N$)
17: let $d = least\_dominator\_through(source, \mathscr{E})$
18: if $loops(d) \subseteq \mathscr{L}$ then
19:     if $\exists v.\ \forall e \in \mathscr{E}.\ value(e) = v$ then
20:         return $v$
21:     let $t = \text{Decide}(out_{\textbf{true}}(d), \{e \in \mathscr{E} \mid out_{\textbf{true}}(d) \overset{*}{\to} e\}, value, \mathscr{L})$
22:     let $f = \text{Decide}(out_{\textbf{false}}(d), \{e \in \mathscr{E} \mid out_{\textbf{false}}(d) \overset{*}{\to} e\}, value, \mathscr{L})$
23:     return $\overline{\phi}(c_d, t, f)$
24: else
25:     let $\ell$ be the outermost loop in $loops(d)$ that is not in $\mathscr{L}$
26:     let $i$ be the nesting depth of $\ell$
27:     let $break\_edges = \text{ComputeBreakEdges}(\ell)$
28:     let $break = \text{BreakCondition}(\ell, break\_edges, \mathscr{L} \cup \{\ell\})$
29:     let $val = \text{Decide}(source, \mathscr{E}, value, \mathscr{L} \cup \{\ell\})$
30:     return $\overline{eval_i}(val, \overline{pass_i}(break))$
```

```
31: function BreakCondition($\ell : N, break\_edges : 2^E, \mathscr{L} : 2^N$)
32: let $all\_edges = break\_edges \cup in(\ell')$
33: let $value\_fn = \lambda e : all\_edges\ .\ \big(\textbf{if } e \in break\_edges \textbf{ then } \overline{\textbf{true}} \textbf{ else } \overline{\textbf{false}}\big)$
34: return Simplify(Decide($root\_edge, all\_edges, value\_fn, \mathscr{L}$))
```

Figure 7.8. CFG to A-PEG conversion algorithm

ComputeInputs The ComputeInputs function starts out by calling Decide on the incoming edges of n (lines 7-9). Intuitively, Decide computes an A-PEG expression that, when evaluated, will decide between different edges (we describe Decide and its arguments in more detail shortly). After calling Decide, ComputeInputs checks if n is a loop-header node (line 10). If it is *not*, then one can simply return the *result* from Decide (line 13). On the other hand, if n is a loop-header node, then its FFG predecessors are nodes from outside the loop (since back edges originating from within the loop now go to n'). In this case, the *result* computed on line 9 only accounts for values coming from outside of the loop, and so we need lines 11 and 12 to adjust the result so that it also accounts for values coming from inside the loop. In particular, we use ComputeInputs(n') to compute the A-PEG expression for values coming from inside the loop, and then we create the θ_i expression that combines the two (with i being the loop-nesting depth of n).

Decide The Decide function is used to create an expression that decides between a set of edges. In particular, suppose we are given a set of FFG edges, and we already know that the program will definitely reach one of these edges starting from the *source* edge – then Decide returns an A-PEG expression that, when evaluated, computes which of these edges will be reached from the *source* edge. The first parameter to Decide is the edge assumed to already have been reached; the second parameter is the set of edges to decide between; the third parameter is a function mapping edges to A-PEG nodes – this function is used to created an A-PEG node from each edge; and the fourth parameter is a looping context, which is the set of loops that Decide is currently analyzing.

Decide starts by calling $least\_dominator\_through(source, \mathscr{E})$ to compute d, the least dominator (in the FFG reachable from *source*) of the given set of edges, where least means furthest away from *source* (line 14). If d is in the current looping context (line 15), then, after optimizing the case where *value* maps all edges to the same A-PEG node (lines 16 and 17), Decide calls itself to decide between the true and false cases (lines 18 and 19). Decide then creates the appropriate ϕ node, using the c_d node created in ComputeAPEG (line 20).

For example, suppose ComputeInputs is called on node 5 from Figure 7.7. Since there are no loops, ComputeInputs returns Decide$(\mathit{root_edge}, \{e_1, e_2, e_3\}, \mathit{value_fn}, \varnothing)$, and Decide only executes lines 14-20. As a result, this leads to the following steps (where we have omitted the first parameter and last two parameters to Decide because they are always the same):

$$\begin{aligned}
&\text{Decide}(\{e_1, e_2, e_3\})\\
&= \phi(c_1, \text{Decide}(\{e_1\}), \text{Decide}(\{e_2, e_3\}))\\
&= \phi(c_1, \text{Decide}(\{e_1\}), \phi(c_2, \text{Decide}(\{e_2\}), = \text{Decide}(\{e_3\})))\\
&= \phi(c_1, \mathit{value\_fn}(e_1), \phi(c_2, \mathit{value\_fn}(e_2), \mathit{value\_fn}(e_3)))\\
&= \phi(c_1, SE_1, \phi(c_2, SE_3, SE_4))
\end{aligned}$$

This is exactly the A-PEG expression used for the input to node 5 in Figure 7.7.

Going back to the code for Decide, if the dominator d is *not* in the same looping context, then the edges that we are deciding between originate from a more deeply nested loop. We therefore need to compute the appropriate break condition – the right combination of *eval*/*pass* that will correctly convert values from the more deeply nested loop into the current looping context. To do this, Decide picks the outermost loop ℓ of the more deeply nested loops that are not in the context (line 22); it computes the set of edges that break out of ℓ using ComputeBreakEdges, a straightforward function not shown here (line 24); it computes the break condition for ℓ using BreakCondition (line 25); it then computes an expression that decides between the edges $\mathscr{E}$, but this time adding ℓ to the loop context (line 26); and finally Decide puts it all together in an *eval*/*pass* expression (line 27).

BreakCondition The BreakCondition function creates a boolean A-PEG node that evaluates to true when the given loop ℓ breaks. Deciding whether or not a loop breaks amounts to deciding if the loop, when started at its header node, reaches the break edges ($break\_edges$) or the back edges ($in(\ell')$). We can reuse our Decide function described earlier for this purpose (lines 29 and 30). Finally, we use the Simplify function, not shown here, to perform basic boolean simplifications on the result of Decide (line 30).

Acknowledgements

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 8

Reverting PEGs to Imperative Code

In this chapter we present the complement to Chapter 7: a procedure for converting a PEG back to a SIMPLE program. Whereas the translation from SIMPLE programs to PEGs was fairly simple, the translation back (which we call reversion) is much more complicated. Since the order of SIMPLE execution is specified explicitly, SIMPLE programs have more structure than PEGs, and so it is not surprising that the translation from PEGs back to SIMPLE is more complex than the other direction. Because of the complexity involved in reversion, we start by presenting a simplified version of the process in Sections 8.1 through 8.5. This simple version is correct but produces SIMPLE programs that are inefficient because they contain a lot of duplicated code. We then present in Sections 8.6 through 8.9 several optimizations on top of the simple process to improve the quality of the generated code by removing the code duplication. These optimizations include branch fusion, loop fusion, hoisting code that is common to both sides of a branch, and hoisting code out of loops. In the setting of SIMPLE, these optimizations are optional – they improve the performance of the generated code but are not required for correctness. However, if we add side-effecting operations like heap reads/writes (as we do in Chapter 10 using the technique described in Chapter 9), these optimizations are not optional anymore: they are needed to make sure that we do not incorrectly duplicate side-effecting operations. Finally, we present more advanced techniques for reverting PEGs to CFGs rather than SIMPLE programs, taking advantage of the flexibility of CFGs in order to produce even more efficient translations of PEGs. This is particularly important when reverting PEGs translated from programs in a language with more advanced control flow such as `breaks` and `continues`.

8.1 CFG-Like PEGs

Before we can proceed, we need to define the precondition of our reversion process. In particular, our reversion process assumes that the PEGs we are processing are *CFG-like*, as formalized by the following definition.

Definition 8.1 (CFG-like PEG context)**.** *We say that a PEG context* Ψ *is CFG-like if* $\Gamma \vdash \Psi : \Gamma'$ *holds using the rules in Figure 8.1.*

The rules in Figure 8.1 impose various restrictions on the structure of the PEG which makes our reversion process simpler. For example, these rules guarantee that the second child of an *eval* node is a *pass* node, and that by removing the second outgoing edge of each θ node, the PEG becomes acyclic. If a PEG context is CFG-like, then it is well formed (Definition 6.1 from Section 6.1). Furthermore, all PEGs produced by our SIMPLE-to-PEG translation process from Chapter 7 or CFG-to-PEG conversion process from Section 7.5 are CFG-like. However, not all well formed PEGs are CFG-like, and in fact it is useful for equality saturation to consider PEGs that are not CFG-like as intermediate steps. To guarantee that the reversion process will work during optimization, the pseudo-boolean formulation described in Section 10.3 ensures that the PEG selected for reversion is CFG-like.

In Figure 8.1, Γ is a type context assigning parameter variables to types. ℓ is the largest loop depth with respect to which the PEG node n is allowed to be loop variant; initializing ℓ to 0 requires n to be loop invariant with respect to all loops. Θ is

$\boxed{\Gamma \vdash \Psi : \Gamma'}$

$$\text{Type-PEG-Context}\frac{\forall (x:\tau) \in \Gamma'.\ \Psi(x) = n \Rightarrow \Gamma \vdash n : \tau}{\Gamma \vdash \Psi : \Gamma'}$$

$\boxed{\Gamma \vdash n : \tau}$

$$\text{Type-PEG}\frac{\Gamma, 0, \varnothing \vdash n : \tau}{\Gamma \vdash n : \tau}$$

$\boxed{\Gamma, \ell, \Theta \vdash n : \tau}$

$$\text{Type-Param}\frac{\Gamma(x) = \tau}{\Gamma, \ell, \Theta \vdash \overline{param}(x) : \tau} \qquad \text{Type-Assume}\frac{(\ell \vdash n : \tau) \in \Theta}{\Gamma, \ell, \Theta \vdash n : \tau}$$

$$\text{Type-Op}\frac{op : (\tau_1, \ldots, \tau_n) \to \tau \quad \Gamma, \ell, \Theta \vdash n_1 : \tau_1 \ \ldots\ \Gamma, \ell, \Theta \vdash n_n : \tau_n}{\Gamma, \ell, \Theta \vdash \overline{op}(n_1, \ldots, n_n) : \tau}$$

$$\text{Type-Phi}\frac{\Gamma, \ell, \Theta \vdash c : \texttt{bool} \quad \Gamma, \ell, \Theta \vdash t : \tau \quad \Gamma, \ell, \Theta \vdash f : \tau}{\Gamma, \ell, \Theta \vdash \overline{\phi}(c, t, f) : \tau}$$

$$\text{Type-Theta}\frac{\begin{array}{c}\ell' = \ell - 1 \\ \Gamma, \ell', \Theta \vdash b : \tau \quad \Gamma, \ell, (\Theta, (\ell \vdash \overline{\theta_\ell}(b,n) : \tau)) \vdash n : \tau\end{array}}{\Gamma, \ell, \Theta \vdash \overline{\theta_\ell}(b, n) : \tau}$$

$$\text{Type-Eval-Pass}\frac{\begin{array}{c}\ell = \ell' + 1 \quad \forall \ell_1, n, \tau'.[(\ell_1 \vdash n : \tau') \in \Theta \Rightarrow \ell_1 < \ell'] \\ \Gamma, \ell', \Theta \vdash v : \tau \qquad \Gamma, \ell', \Theta \vdash c : \texttt{bool}\end{array}}{\Gamma, \ell, \Theta \vdash \overline{eval_{\ell'}}(v, \overline{pass_{\ell'}}(c)) : \tau}$$

$$\text{Type-Reduce}\frac{\ell \geq \ell' \quad \Theta \supseteq \Theta' \quad \Gamma, \ell', \Theta' \vdash n : \tau}{\Gamma, \ell, \Theta \vdash n : \tau}$$

Figure 8.1. Rules defining CFG-like PEGs

an assumption context used to type recursive expressions. Each assumption in Θ has the form $\ell \vdash n : \tau$, where n is a θ_ℓ node; $\ell \vdash n : \tau$ states that n has type τ at loop depth ℓ. Assumptions in Θ are introduced in the Type-Theta rule and used in the Type-Assume rule. The assumptions in Θ prevent “unsolvable” recursive PEGs such as $x = 1 + x$ or “ambiguous” recursive PEGs such as $x = 0 * x$. The requirement in Type-Eval-Pass regarding Θ prevents situations such as $x = \overline{\theta_2}(eval_1(eval_2(x, \ldots), \ldots), \ldots)$, in which essentially the initializer for the nested loop is the final result of the outer loop.

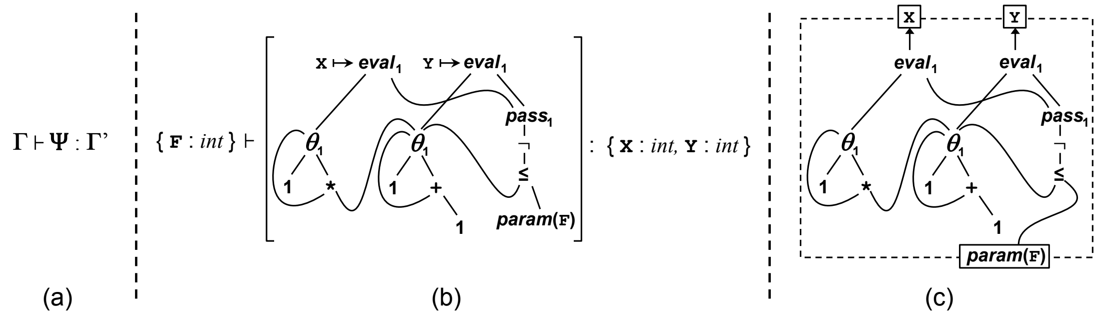


Figure 8.2. Visual representation of the judgement for CFG-like PEG contexts: (a) shows the syntactic form of the judgement; (b) shows an example of the syntactic form; and (c) shows the same example in visual form

Although $\Gamma \vdash \Psi : \Gamma'$ is the syntactic form which we will use in the body of the text, our diagrams will use the visual representation of $\Gamma \vdash \Psi : \Gamma'$ shown in Figure 8.2. Part (a) of the figure shows the syntactic form of the judgement (used in the text); (b) shows an example of the syntactic form; and finally (c) shows the same example in the visual form used in our diagrams with variable types implicit.

8.2 Overview

We can draw a parallel between CFG-like PEG contexts and well typed statements: both can be seen as taking inputs Γ and producing outputs Γ'. With this parallel in mind, our basic strategy for reverting PEGs to SIMPLE programs is to recursively translate CFG-like PEG contexts $\Gamma \vdash \Psi : \Gamma'$ to well typed SIMPLE statements $\Gamma \vdash s : \Gamma'$. Therefore, the precondition for our reversion algorithm is that the PEG context we revert must be CFG-like according to the rules in Figure 8.1.

For the reversion process, we make two small changes to the typing rules for SIMPLE programs. First, we want to allow the reversion process to introduce temporary variables without having to add them to Γ' (in $\Gamma \vdash s : \Gamma'$). To this end, we allow intermediate variables to be dropped from Γ', so that $\Gamma \vdash s : \Gamma'$ and $\Gamma'' \subseteq \Gamma'$ implies

$\Gamma \vdash s : \Gamma''$.

Second, to more clearly highlight which parts of the generated SIMPLE code modify which variables, we introduce a notion $\Gamma_0; \Gamma \vdash s : \Gamma'$ of SIMPLE statements s which use but do not modify variables in Γ_0 (where Γ and Γ_0 are disjoint). We call Γ_0 the *immutable context*. The rules in Figure 7.2 can be updated appropriately to disallow variables from Γ_0 to be modified. Similarly, we add Γ_0 to the notion of CFG-like PEG contexts: $\Gamma_0; \Gamma \vdash \Psi : \Gamma'$. Since PEG contexts cannot modify variables anyway, the semantics of bindings in Γ_0 is exactly the same as bindings in Γ (and so we do not need to update the rules from Figure 8.1). Still, we keep an immutable context Γ_0 around for PEG contexts because Γ_0 from the PEG context gets reflected into the generated SIMPLE statements where it has a meaning. Thus, our reversion algorithm will recursively translate CFG-like PEG contexts $\Gamma_0; \Gamma \vdash \Psi : \Gamma'$ to well typed SIMPLE statements $\Gamma_0; \Gamma \vdash s : \Gamma'$.

Throughout the rest of this chapter, we assume that there is a PEG context $\bar{\Gamma}_0; \bar{\Gamma} \vdash \bar{\Psi} : \bar{\Gamma}'$ that we are currently reverting. Furthermore, we define Γ_0 to be:

$$\Gamma_0 = \bar{\Gamma}_0 \cup \bar{\Gamma} \tag{8.1}$$

As will become clear in Section 8.3, the Γ_0 from Equation (8.1) will primarily be used as the immutable context in recursive calls to the reversion algorithm. The above definition of Γ_0 states that when making a recursive invocation to the reversion process, the immutable context for the recursive invocation is the entire context from the current invocation.

Statement Nodes The major challenge in reverting a CFG-like PEG context lies in handling the primitive operators for encoding control flow: ϕ for branches and *eval*, *pass*, and θ for loops. To handle such primitive nodes, our general approach is to repeatedly

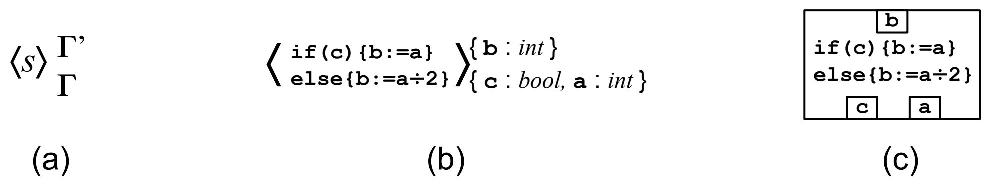


Figure 8.3. Diagrammatic representation of a statement node

replace a subset of the PEG nodes with a new kind of PEG node called a *statement node*. A statement node is a PEG node $\langle s\rangle_{\Gamma}^{\Gamma'}$ where s is a SIMPLE statement satisfying $\Gamma_0;\Gamma \vdash s : \Gamma'$ (recall that Γ_0 comes from Equation (8.1)). The node has many inputs, one for each variable in the domain Γ, and unlike any other node we have seen so far, it also has many outputs, one for each variable of Γ'. A statement node can be perceived as a primitive operator which, given an appropriately typed list of input values, executes the statement s with those values in order to produce an appropriately typed list of output values. Although $\langle s\rangle_{\Gamma}^{\Gamma'}$ is the syntactic form which we will use in the body of the text, our diagrams will use the visual representation of $\langle s\rangle_{\Gamma}^{\Gamma'}$ shown in Figure 8.3. Part (a) of the figure shows the syntactic form of a statement node (used in the text); (b) shows an example of the syntactic form; and finally (c) shows the same example in the visual form used in our diagrams. Note that the visual representation in part (c) uses the same visual convention that we have used throughout for all PEG nodes: the inputs flow into the bottom side of the node, and the outputs flow out from the top side of the node.

The general approach we take is that *eval* nodes will be replaced with while-loop statement nodes (which are statement nodes $\langle s\rangle_{\Gamma}^{\Gamma'}$ where s is a while statement) and ϕ nodes will be replaced with if-then-else statement nodes (which are statement nodes $\langle s\rangle_{\Gamma}^{\Gamma'}$ where s is an if-then-else statement). To this end, our most simplistic reversion algorithm, which we present first, converts PEG contexts to statements in three phases:

1. We replace all *eval*, *pass*, and θ nodes with while-loop statement nodes. This results in an acyclic PEG.

2. We replace all ϕ nodes with if-then-else statement nodes. This results in a PEG with only statement nodes and domain operators such as $+$ and $*$ (that is to say, there are no more *eval*, *pass*, θ, or ϕ nodes).

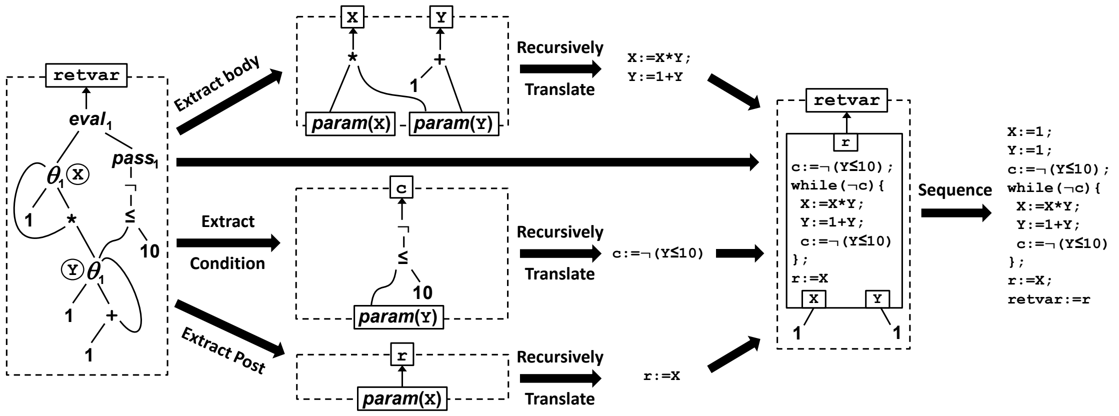
3. We sequence the statement nodes and domain operators into successive assignment statements. For a statement node $\langle s \rangle_{\Gamma}^{\Gamma'}$, we simply inline the statement s into the generated code.

We present the above three steps in Sections 8.3, 8.4, and 8.5, respectively. Finally, since the process described above is simplistic and results in large amounts of code duplication, we present in Sections 8.6 through 8.9 several optimizations that improve the quality of the generated SIMPLE code.

8.3 Translating Loops

In our first phase, we repeatedly convert each loop-invariant *eval* node, along with the appropriate *pass* and θ nodes, into a while-loop statement node. The nested loop-variant *eval* nodes will be taken care of when we recursively revert the "body" of the loop-invariant *eval* nodes to statements. For each loop-invariant $eval_\ell$ node, we apply the process described below.

First, we identify the set of θ_ℓ nodes reachable from the current $eval_\ell$ node or its $pass_\ell$ node without passing through other loop-invariant nodes (in particular, without passing through other $eval_\ell$ nodes). Let us call this set of nodes S. As an illustrative example, consider the left-most PEG context in Figure 8.4, which computes the factorial of 10. When processing the single $eval_1$ node in this PEG, the set S will contain both θ_1 nodes. The intuition is that each θ node in S will be a loop variable in the SIMPLE code we generate. Thus, our next step is to assign a fresh variable x for each θ_ℓ node in S; let b_x refer to the first child of the θ_ℓ node (i.e. the base case), and i_x refer to the second



```
c:=¬(Y≤10);
while(¬c){
  X:=X*Y;
  Y:=1+Y;
  c:=¬(Y≤10)
};
r:=X
```

```
X:=1;
Y:=1;
c:=¬(Y≤10);
while(¬c){
  X:=X*Y;
  Y:=1+Y;
  c:=¬(Y≤10)
};
r:=X;
retvar:=r
```

Figure 8.4. Example of converting *eval* nodes to while-loop statement nodes

child (i.e. the iterative case); also, given a node $n \in S$, we use $var(n)$ for the fresh variable we just created for n. In the example from Figure 8.4, we created two fresh variables x and y for the two θ_1 nodes in S. After assigning fresh variables to all nodes in S, we then create a type context Γ as follows: for each $n \in S$, Γ maps $var(n)$ to the type of node n in the PEG, as given by the typing rules in Figure 8.1. For example, in Figure 8.4 the resulting type context Γ is $\{\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int}\}$.

Second, we construct a new PEG context Ψ_i that represents the body of the loop. This PEG context will state in PEG terms how the loop variables are changed in one iteration of the loop. For each variable x in the domain of Γ, we add an entry to Ψ_i mapping x to a copy of i_x (recall that i_x is the second child of the θ node which was assigned variable x). The copy of i_x is a fully recursive copy, in which descendants have also been copied, but with one important modification: while performing the copy, when we reach a node $n \in S$, we do not copy n; instead we use a parameter node referring to $var(n)$. This has the effect of creating a copy of i_x with any occurrence of $n \in S$ replaced by a parameter node referring to $var(n)$, which in turn has the effect of expressing the next value of variable x in terms of the current values of all loop variables. From the way it is constructed, Ψ_i will satisfy $\Gamma_0; \Gamma \vdash \Psi_i : \Gamma$, essentially specifying, in terms of PEGs, how the loop variables in Γ are changed as the loop iterates (recall that Γ_0 comes from Equation (8.1)). Next, we recursively revert Ψ_i satisfying $\Gamma_0;\Gamma \vdash \Psi_i : \Gamma$ to a SIMPLE statement s_i satisfying $\Gamma_0;\Gamma \vdash s_i : \Gamma$. The top-center PEG context in Figure 8.4 shows Ψ_i for our running example. In this case Ψ_i states that the body of the loop modifies the loop variables as follows: the new value of `x` is `x*y`, and the new value of `y` is `1+y`. Figure 8.4 also shows the SIMPLE statement resulting from the recursive invocation of the reversion process.

Third, we take the second child of the *eval* node that we are processing. From the way the PEG type rules are set up in Figure 8.1, this second child must be the *pass* node of the *eval*. Next, we take the first child of this *pass* node and make a copy of this first child with any occurrence of $n \in S$ replaced by a parameter node referring to $var(n)$. Let c be the PEG node produced by this operation, and let Ψ_c be the singleton PEG context $\{x_c : c\}$, where x_c is fresh. Ψ_c represents the computation of the break condition of the loop in terms of the loop variables. From the way it is constructed, Ψ_c will satisfy $\Gamma_0;\Gamma \vdash \Psi_c : \{x_c : \texttt{bool}\}$. We then recursively revert Ψ_c satisfying $\Gamma_0;\Gamma \vdash \Psi_c : \{x_c : \texttt{bool}\}$ to a SIMPLE statement s_c satisfying $\Gamma_0;\Gamma \vdash s_c : \{x_c : \texttt{bool}\}$. s_c simply assigns the break condition of the loop to the variable x_c. The middle row of Figure 8.4 shows the PEG context for the break condition and the corresponding SIMPLE statement evaluating the break condition.

Fourth, we take the first child of the *eval* node and make a copy of this first child with any occurrence of $n \in S$ replaced with a parameter node referring to $var(n)$. Let r be the PEG node produced by this operation, and let Ψ_r be the singleton PEG context $\{x_r : r\}$, where x_r is fresh. Ψ_r represents the value desired after the loop in terms of the loop variables. From the way it is constructed, Ψ_r will satisfy $\Gamma_0;\Gamma \vdash \Psi_r : \{x_r : \tau\}$, where τ is the type of the first child of the *eval* node in the original PEG. We then recursively revert Ψ_r satisfying $\Gamma_0;\Gamma \vdash \Psi_r : \{x_r : \tau\}$ to a SIMPLE statement s_r satisfying

$\Gamma_0;\Gamma \vdash s_r : \{x_r : \tau\}$. s_r simply assigns the value desired after the loop into variable x_r. The bottom row of Figure 8.4 shows the PEG context for the value desired after the loop and the corresponding SIMPLE statement evaluating the desired value. Often, but not always, the first child of the $eval_\ell$ node will be a θ_ℓ node, in which case the statement will simply copy the variable as in this example.

Finally, we replace the $eval_\ell$ node being processed with the while-loop statement node $\langle s_c; \textbf{while}\ (\neg x_c)\ \{s_i; s_c\}; s_r\rangle_\Gamma^{\{x_r:\tau\}}$. Figure 8.4 shows the while-loop statement node resulting from translating the *eval* node in the original PEG context. Note that the while-loop statement node has one input for each loop variable, namely one input for each variable in the domain of Γ. For each such variable x, we connect b_x to the corresponding x input of the while-loop statement node (recall that b_x is the first child of the θ node which was assigned variable x). Figure 8.4 shows how in our running example, this amounts to connecting a 1 node to both inputs of the while-loop statement node. In general, our way of connecting the inputs of the while-loop statement node makes sure that each loop variable x is initialized with the its corresponding base value b_x. After this input initialization, s_c assigns the status of the break condition to x_c. While the break condition fails, the statement s_i updates the values of the loop variables, then s_c assigns the new status of the break condition to x_c. Once the break condition passes, s_r computes the desired value in terms of the final values of the loop variables and assigns it to x_r. Note that it would be more “faithful” to place s_r inside the loop, doing the final calculations in each iteration, but we place it after the loop as an optimization since s_r does not affect the loop variables. The step labeled “Sequence” in Figure 8.4 shows the result of sequencing the PEG context that contains the while-loop statement node. This sequencing process will be covered in detail in Section 8.5.

Note that in the computation of the break condition in Figure 8.4, there is a double negation, in that we have `c := ¬... ;` and `while(¬c)`. The more advanced reversion techniques described in Section 8.10 prevent such double negations.

8.4 Translating Branches

After our first phase, we have replaced *eval*, *pass*, and θ nodes with while-loop statement nodes. Thus, we are left with an acyclic PEG context that contains ϕ nodes, while-loop statement nodes, and domain operators like $+$ and $*$. In our second phase, we repeatedly translate each ϕ node into an if-then-else statement node. In order to convert a ϕ node to an if-then-else statement node, we must first determine the set of nodes which will always need to be evaluated regardless of whether the guard condition is true or false. This set can be hard to determine when there is another ϕ node nested inside the ϕ node. To see why this would be the case, consider the example in Figure 8.5, which we will use as a running example to demonstrate branch translation. Looking at part (a), one might think at first glance that the $\div$ node in the first diagram is always evaluated by the ϕ node labeled ① since it is used in the PEGs for both the second and third children. However, upon further examination one realizes that actually the $\div$ node is evaluated in the second child only when $\mathtt{x}{\neq}\mathtt{0}$ due to the ϕ node labeled ②. To avoid these complications, it is simplest if we first convert ϕ nodes that do not have any other ϕ nodes as descendants. After this replacement, there will be more ϕ nodes that do not have ϕ descendants, so we can repeat this until no ϕ nodes are remaining. In the example from Figure 8.5, we would convert node ② first, resulting in (b). After this conversion, node ① no longer has any ϕ descendants and so it can be converted next. Thus, we replace ϕ nodes in a bottom-up order. For each ϕ node, we use the following process.

First, we determine the set S of nodes that are descendants of both the second and third child of the current ϕ node (i.e. the true and false branches). These are the nodes that will get evaluated regardless of which way the ϕ goes. We assign a fresh variable to each node in this set, and as in the case of loops we use $var(n)$ to denote the variable we

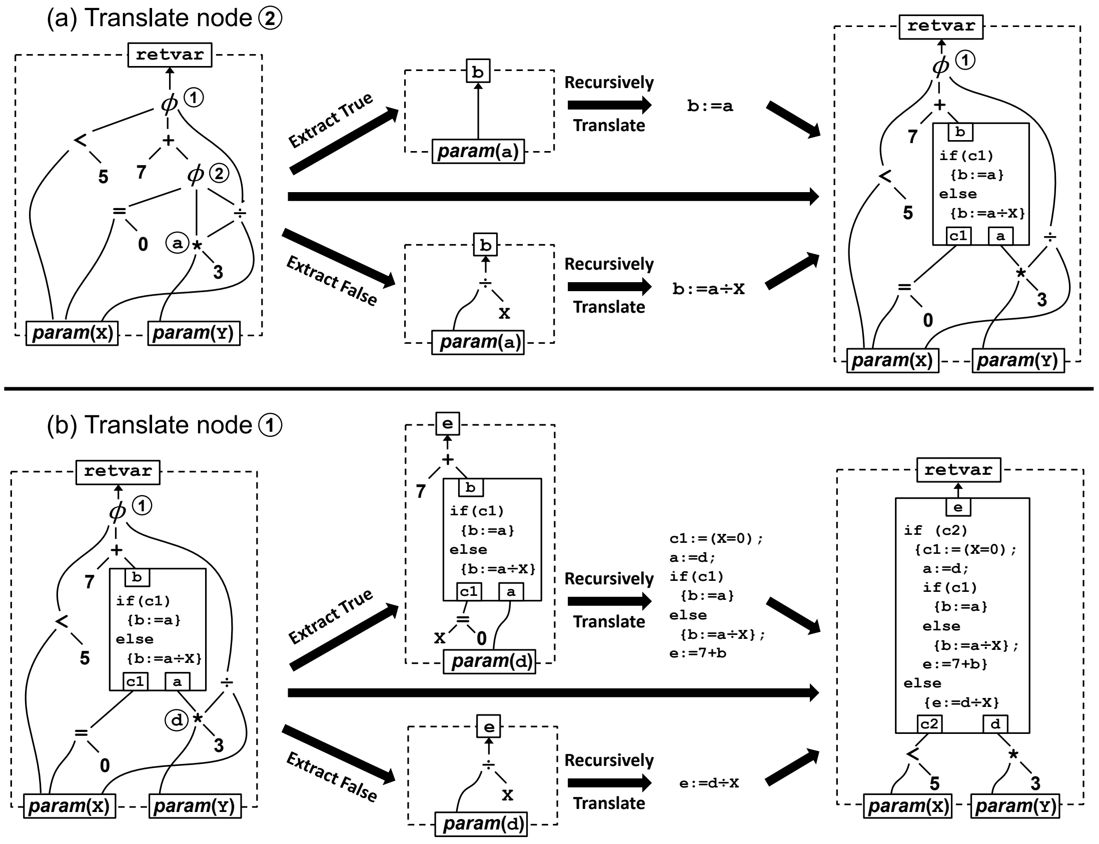

(a) Translate node ②
retvar
ϕ ①
Extract True
Recursively Translate
b:=a
Extract False
b:=a÷X
if(c1) {b:=a} else {b:=a÷X}
param(X)
param(Y)
param(a)
(b) Translate node ①
c1:=(X=0); a:=d; if(c1) {b:=a} else {b:=a÷X}; e:=7+b
e:=d÷X
if (c2) {c1:=(X=0); a:=d; if(c1) {b:=a} else {b:=a÷X}; e:=7+b} else {e:=d÷X}
param(d)

Figure 8.5. Example of converting ϕ nodes to if-then-else statement nodes

have assigned to n. In Figure 8.5(a), the $*$ node is a descendant of both the second and third child of node ②, so we assign it the fresh variable a. Note that the 3 node should also be assigned a variable, but we do not show this in the figure since the variable is never used. Next, we take the second child of the ϕ node and make a copy of this second child in which any occurrence of $n \in S$ has been replaced with a parameter node referring to $var(n)$. Let t be the PEG node produced by this operation. Then we do the same for the third child of the ϕ node to produce another PEG node f. The t and f nodes represent the true and false computations in terms of the PEG nodes that get evaluated regardless of the direction the ϕ goes. In the example from Figure 8.5(a), t is $\overline{param}(\mathtt{a})$, and f is $\div(\overline{param}(\mathtt{a}), \overline{param}(\mathtt{x}))$. Examining t and f, we produce a context Γ of the newly created fresh variables used by either t or f. In the example, Γ would be simply $\{\mathtt{a} : \mathtt{int}\}$. The domain of Γ does not contain `x` since `x` is not a new variable (i.e. `x` is in the domain of Γ_0, where Γ_0 comes from Equation (8.1)). Thus, t and f are PEGs representing the true and false cases in terms of variables Γ representing values that would be calculated regardless.

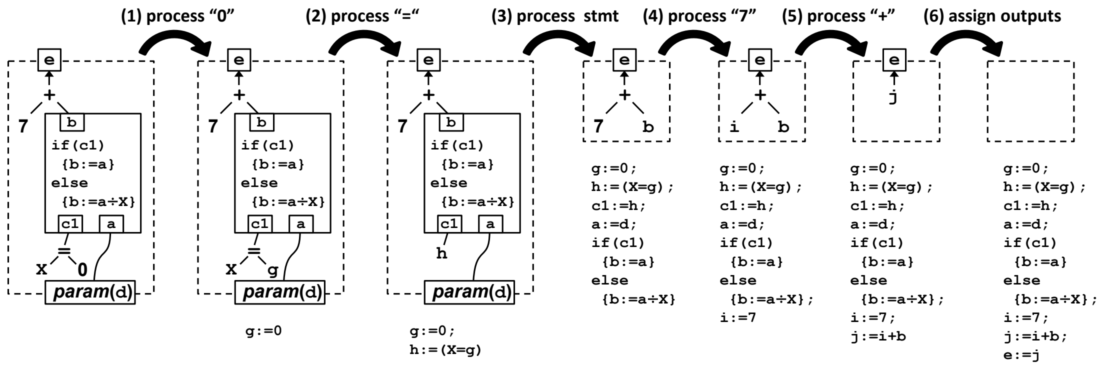
Second, we invoke the reversion process recursively to translate t and f to statements. In particular, we create two singleton contexts $\Psi_t = \{x_\phi : t\}$ and $\Psi_f = \{x_\phi : f\}$ where x_ϕ is a fresh variable, making sure to use the same fresh variable in the two contexts. From the way it is constructed, Ψ_t satisfies $\Gamma_0;\Gamma \vdash \Psi_t : \{x_\phi : \tau\}$, where τ is the type of the ϕ node. Thus, we recursively revert Ψ_t satisfying $\Gamma_0;\Gamma \vdash \Psi_t : \{x_\phi : \tau\}$ to a SIMPLE statement s_t satisfying $\Gamma_0;\Gamma \vdash s_t : \{x_\phi : \tau\}$. Similarly, we revert Ψ_f satisfying $\Gamma_0;\Gamma \vdash \Psi_f : \{x_\phi : \tau\}$ to a statement s_f satisfying $\Gamma_0;\Gamma \vdash s_f : \{x_\phi : \tau\}$. The steps labeled "Extract True" and "Extract False" in Figure 8.5 show the process of producing Ψ_t and Ψ_f in our running example, where the fresh variable x_ϕ is `b` in part (a) and `e` in part (b). Note that Ψ_t and Ψ_f may themselves contain statement nodes, as in Figure 8.5(b), but this poses no problems for our recursive algorithm. Finally, there is an important notational convention to note in Figure 8.5(a). Recall that Ψ_f satisfies $\Gamma_0;\Gamma \vdash \Psi_f : \{x_\phi : \tau\}$, and that in Figure 8.5(a) $\Gamma_0 = \{\mathtt{x} : \mathtt{int}\}$ and $\Gamma = \{\mathtt{a} : \mathtt{int}\}$. In the graphical representation of $\Gamma_0;\Gamma \vdash \Psi_f : \{x_\phi : \tau\}$ in Figure 8.5(a), we display variable `a` as a real boxed input (since it is part of Γ), whereas because `x` is in Γ_0, we display `x` without a box and using the shorthand of omitting the *param* (even though in reality it is there).

Finally, we replace the ϕ node we are processing with the if-then-else statement node $\langle \mathbf{if}\ (x_c)\ \{s_t\}\ \mathbf{else}\ \{s_f\}\rangle^{\{x_\phi:\tau\}}_{(\Gamma,x_c:\mathtt{bool})}$ (where x_c is fresh). This statement node has one input for each entry in Γ, and it has one additional input x_c for the guard value. We connect the x_c input to the first child of the ϕ node we are currently processing. For each variable x in the domain of Γ, we connect the x input of the statement node to the "always evaluated" node n for which $var(n) = x$. Figure 8.5 shows the newly created if-then-else statement nodes and how they are connected when processing ϕ nodes ① and ②. In general, our way of connecting the inputs of the if-then-else statement node makes sure that each always-evaluated node is assigned to the appropriate variable and the guard condition is assigned to x_c. After this initialization, the statement checks x_c, the guard condition, to determine whether to take the true or false branch. In either case, the chosen statement computes the desired value of the branch and assigns it to x_ϕ.

8.5 Sequencing Operations and Statements

When we reach our final phase, we have already eliminated all primitive operators ($eval$, $pass$, θ, and ϕ) and replaced them with statement nodes, resulting in an acyclic PEG context containing only statement nodes and domain operators like $+$ and $*$. At this point we need to sequence these statement nodes and operator nodes. We start off by initializing a statement variable S as the empty statement. We will process each PEG node one by one, postpending lines to S and then replacing the PEG node with a parameter node. It is simplest to process the PEG nodes from the bottom up, processing a node once all of its inputs are parameter nodes. Figure 8.6 shows every stage of converting a PEG context to a statement. At each step, the current statement S is shown below the PEG context.

If the node being processed is a domain operator, we do the following. Because we only process nodes where all of its inputs are parameter nodes, the domain operator node we are processing must be of the following form: $\overline{op}(\overline{param}(x_1), \ldots, \overline{param}(x_k))$. We first designate a fresh variable x. Then, we postpend the line $x := op(x_1, \ldots, x_k)$ to S. Finally, we replace the current node with the node $\overline{param}(x)$. This process is applied in the first, second, fourth, and fifth steps of Figure 8.6. Note that in the first and fourth steps, the constants 0 and 7 are a nullary domain operators.



```
g:=0
```

```
g:=0;
h:=(X=g)
```

```
g:=0;
h:=(X=g);
c1:=h;
a:=d;
if(c1)
 {b:=a}
else
 {b:=a÷X}
```

```
g:=0;
h:=(X=g);
c1:=h;
a:=d;
if(c1)
 {b:=a}
else
 {b:=a÷X};
i:=7
```

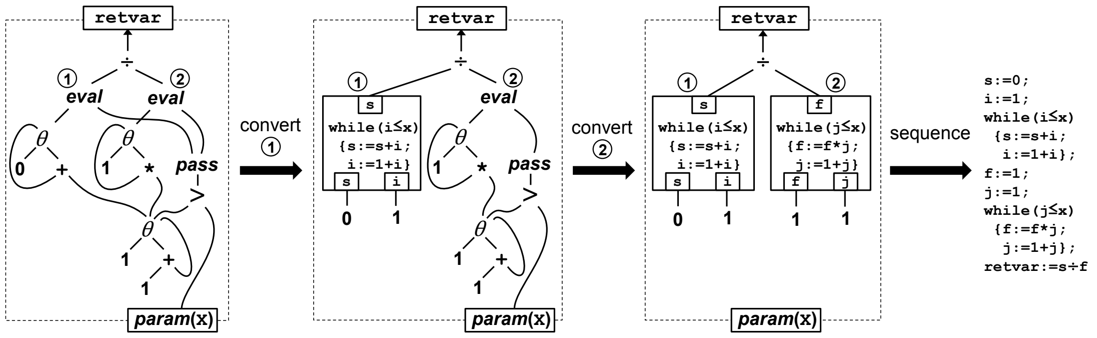
```
g:=0;
h:=(X=g);
c1:=h;
a:=d;
if(c1)
 {b:=a}
else
 {b:=a÷X};
i:=7;
j:=i+b
```

```
g:=0;
h:=(X=g);
c1:=h;
a:=d;
if(c1)
 {b:=a}
else
 {b:=a÷X};
i:=7;
j:=i+b;
e:=j
```

Figure 8.6. Example of sequencing a PEG context (with statement nodes) into a statement

If on the other hand the node being processed is a statement node, we do the following. This node must be of the following form: $\langle s\rangle_{\Gamma}^{\Gamma'}$. For each input variable x in the domain of Γ, we find the $\overline{param}(x_0)$ that is connected to input x, and postpend the line $x := x_0$ to S. In this way, we are initializing all the variables in the domain of Γ. Next, we postpend s to S. Finally, for each output variable x' in the domain of Γ', we replace any links in the PEG to the x' output of the statement node with a link to $\overline{param}(x')$. This process is applied in the third step of Figure 8.6.

Finally, after processing all domain operators and statement nodes, we will have each variable x in the domain of the PEG context being mapped to a parameter node $\overline{param}(x')$. So, for each such variable, we postpend the line $x' := x$ to S. All these assignments should intuitively run in parallel. This causes problems if there is a naming conflict, for example x gets y and y gets x. In such cases, we simply introduce intermediate fresh copies of all the variables being read, and then we perform all the assignments by reading from the fresh copies. In the case of x and y, we would create copies x' and y' of x and y, and then assign x' to y, and y' to x. This process is applied in the sixth and last step of Figure 8.6 (without any naming conflicts). The value of S is the final result of the reversion, although in practice we apply copy propagation to this statement since the sequencing process produces a lot of intermediate variables.



```
s:=0;
i:=1;
while(i≤x)
  {s:=s+i;
   i:=1+i};
f:=1;
j:=1;
while(j≤x)
  {f:=f*j;
   j:=1+j};
retvar:=s÷f
```

Figure 8.7. Reversion of a PEG without applying loop fusion

8.6 Loop Fusion

Although the process described above will successfully revert a PEG to a SIMPLE program, it does so by duplicating a lot of code. Consider the reversion process in Figure 8.7. The original SIMPLE program for this PEG was the following:

```
s:=0; f:=1; i:=1;
while(i<=x) { s:=s+i; f:=f*i; i:=1+i };
retvar:=s÷f
```

The conversion of the above code to PEG results in two *eval* nodes, one for each variable that is used after the loop. The reversion process described so far converts each *eval* node separately, resulting in two separate loops in the final SIMPLE program. Here we present a simple optimization that prevents this code duplication by fusing loops during reversion. In fact, this added loop-fusion phase can even fuse loops that were distinct in the original program. Thus, loop fusion can be performed simply by converting to a PEG and immediately reverting back to a SIMPLE program, without even having to do any intermediate transformations on the PEG.

We update our reversion process to perform loop fusion by making several changes. First, we modify the process for converting *eval* nodes to while-loop statement nodes in three ways. The revised conversion process is shown in Figure 8.8 using the

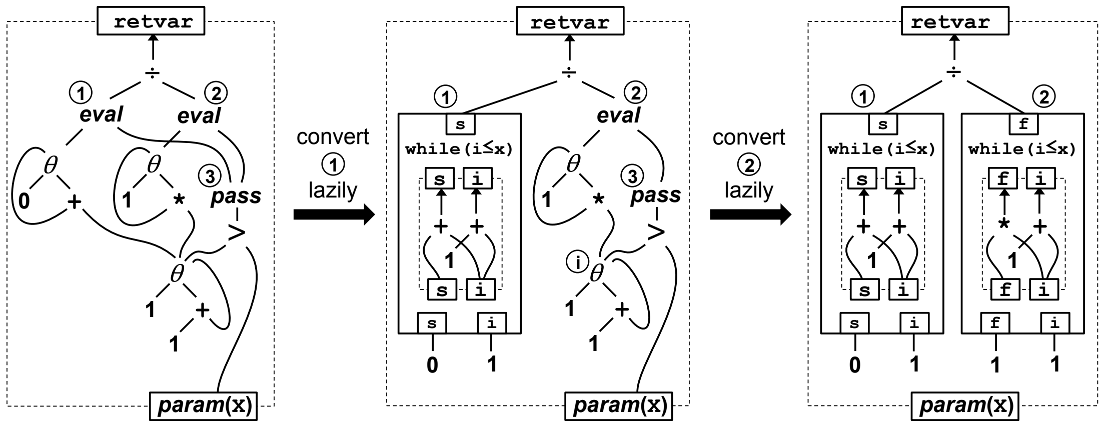


Figure 8.8. Conversion of *eval* nodes to revised loop nodes

same PEG as before. The first modification is that we tag each converted θ node with the fresh variable we designate for it. For example, the conversion process for the first *eval* node in Figure 8.8 generates a fresh variable `i` for one of the θ nodes, and so we tag this θ node with `i`. If we ever convert a θ node that has already been tagged from an earlier *eval* conversion, we reuse that variable. For example, when converting the second *eval* node in Figure 8.8, we reuse the variable `i` unlike in Figure 8.7 where we introduced a fresh variable `j`. This way all the *eval* nodes are using the same naming convention. The second modification is that when processing an *eval* node, we do not immediately revert the PEG context for the loop body into a statement, but rather we remember it for later. This is why the bodies of the while-loop statement nodes in Figure 8.8 are still PEG contexts rather than statements. Thus, we have to introduce a new kind of node, which we call a *loop node*, which is like a while-loop statement node, except that it stores the body (and only the body) of the loop as a PEG context rather than a statement – the remaining parts of the loop are still converted to statements (in particular, the condition and the post-loop computation are still converted to statements, as was previously shown in Figure 8.4). As an example, nodes ① and ② in the right-most part of Figure 8.8 are loop nodes. Furthermore, because we are leaving PEGs inside the loop nodes to be converted for later, we use the phrase “convert lazily” in Figure 8.8. The third modification is that the newly introduced loop nodes store an additional piece of information when compared to while-loop statement nodes. In particular, when we replace an *eval* node with a loop node, the new loop node will store a link back to the *pass* node of the *eval* node being replaced. We store these additional links so that we can later identify fusable loops: we will consider two loop nodes fusable only if they share the same *pass* node. We do not show these additional links explicitly in Figure 8.8, but all the loop nodes in that Figure implicitly store a link back to the same *pass* node, namely node ③.

Second, after converting the ϕ nodes but before sequencing, we search for loop nodes which can be fused. Two loop nodes can be fused if they share the same *pass* node and neither one is a descendant of the other. For example, the two loop nodes in Figure 8.8 can be fused. If one loop node is a descendant of the other, then the result of finishing the descendant loop is required as input to the other loop, so they cannot be executed simultaneously. To fuse the loops, we simply union their body PEG contexts, as well as their inputs and their outputs. The step labeled “fuse ① & ②” in Figure 8.9 demonstrates this process on the result of Figure 8.8. This technique produces correct results because we used the same naming convention across *eval* nodes and we used fresh variables for all θ nodes, so no two distinct θ nodes are assigned the same variable. We repeat this process until there are no fusable loop nodes.

Finally, the sequencing process is changed to first convert all loop nodes to while-loop statement nodes, which involves recursively translating the body PEG context inside the loop node to a statement. This additional step is labeled “convert loops to statements” in Figure 8.9. The final SIMPLE program has only one while loop which simultaneously calculates both of the desired results of the loop, as one would expect.

To summarize, the process described so far is to (1) translate *eval* nodes into loop nodes, (2) translate ϕ nodes into if-then-else statements, (3) perform fusion of loop nodes,

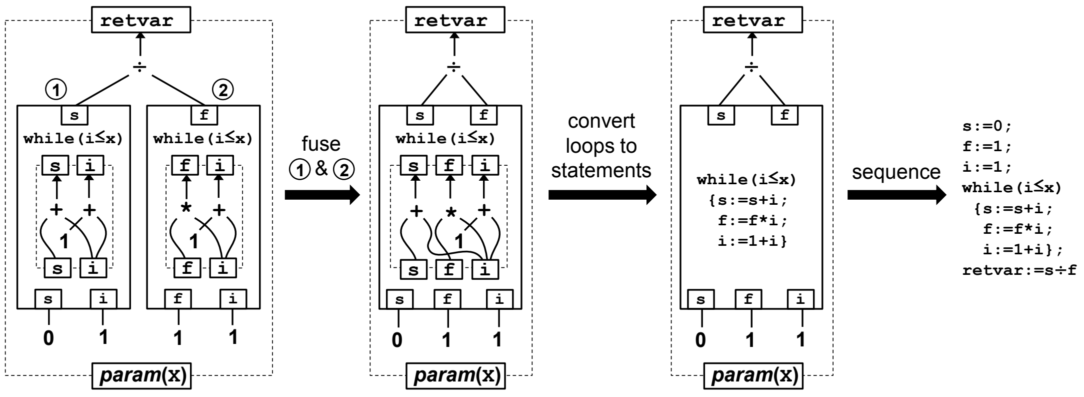
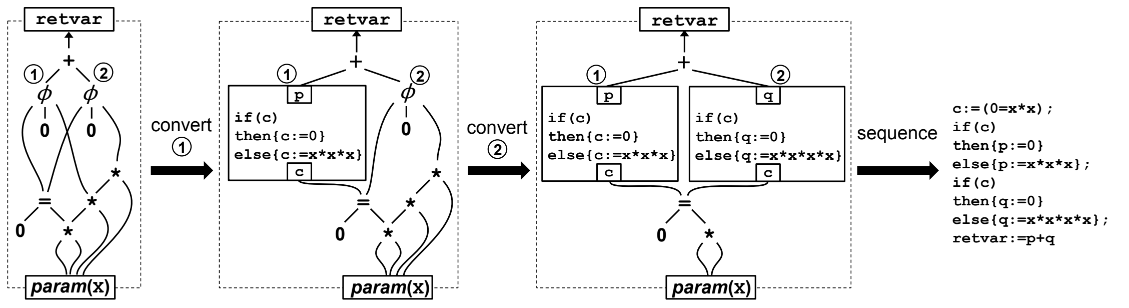


Figure 8.9. Fusion of loop nodes

and (4) sequence. It is important to perform the fusion of loop nodes *after* converting ϕ nodes, rather than right after converting *eval* nodes. Consider for example two loops with the same break condition, neither of which depend on the other, but where one is always executed and the other is only executed when some branch guard is true (that is to say, its result is used only on one side of a ϕ node). If we perform fusion of loop nodes before converting ϕ nodes, then these two loop nodes appear to be fusable, but fusing them would cause both loops to always be evaluated, which is not semantics preserving. We avoid this problem by processing all ϕ nodes first, after which point we know that all the remaining nodes in the PEG context must be executed (although some of these nodes may be branch nodes). In the example just mentioned with two loops, the loop which is under the guard (and thus under a ϕ node) will be extracted and recursively processed when the ϕ node is transformed into a branch node. In this recursive reversion, only one loop will be present, the one under the guard, and so no loop fusion is performed. After the ϕ node is processed, there will be only one remaining loop node, the one which is executed unconditionally. Again, since there is only one loop node, no loop fusion is performed, so the semantics is preserved.



```
c:=(0=x*x);
if(c)
then{p:=0}
else{p:=x*x*x};
if(c)
then{q:=0}
else{q:=x*x*x*x};
retvar:=p+q
```

Figure 8.10. Reversion of a PEG without applying branch fusion

8.7 Branch Fusion

In the same way that our previously described reversion process duplicated loops, so does it duplicate branches, as demonstrated in Figure 8.10. Similarly to loop fusion, our reversion process can be updated to perform branch fusion.

First, we modify the processing of ϕ nodes to make the reversion of recursive PEG contexts lazy: rather than immediately processing the extracted true and false PEG contexts, as was done in Figure 8.5, we instead create a new kind of node called a *branch node* and store the true and false PEG contexts in that node. A branch node is like an if-then-else statement node, except that instead of having SIMPLE code for the true and false sides of the statement, the branch node contains PEG contexts to be processed later. As with if-then-else statement nodes, a branch node has a guard input which is the first child of the ϕ node being replaced (that is to say, the value of the branch condition). For example, Figure 8.11 shows this lazy conversion of ϕ nodes on the same example as Figure 8.10. The nodes labeled ① and ② in the right-most part of Figure 8.11 are branch nodes.

Second, after all ϕ nodes have been converted to branch nodes, we search for branch nodes that can be fused. If two branch nodes share the same guard-condition input, and neither one is a descendant of the other, then they can be fused. Their true PEG

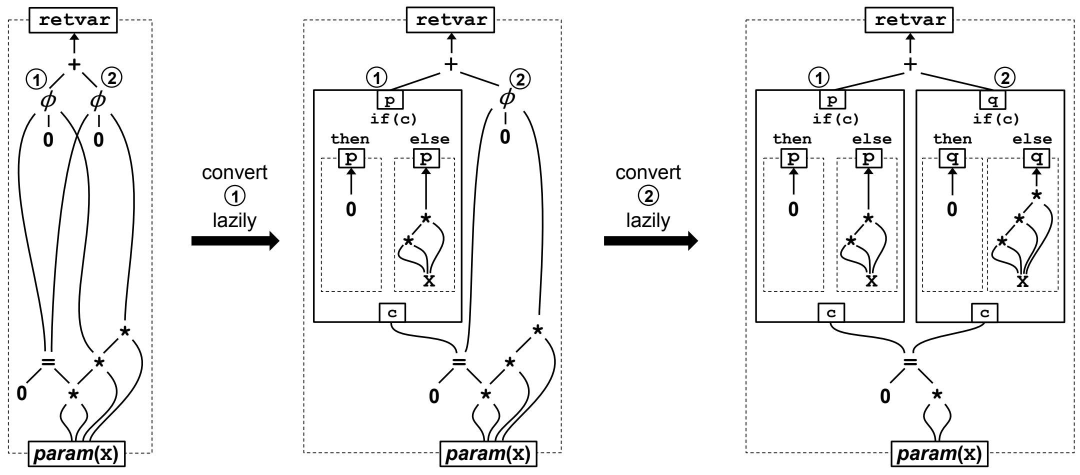
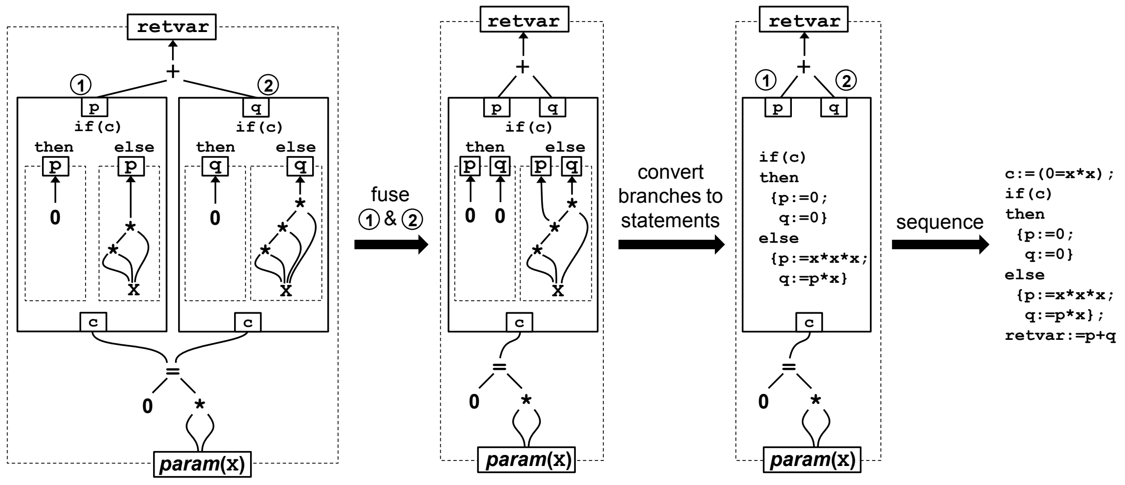


Figure 8.11. Conversion of ϕ nodes to revised branch nodes

contexts, false PEG contexts, inputs, and outputs are all unioned together respectively. This process is much like the one for loop fusion and is demonstrated in Figure 8.12 in the step labeled "fuse ① & ②". Notice that when we union two PEGs, if there is a node in each of the two PEGs representing the exact same expression, the resulting union will only contain one copy of this node. This leads to an occurrence of common sub-expression elimination in Figure 8.12: when the false and true PEGs are combined during fusion, the resulting PEG only has one `x*x*x`, which allows the computation for `q` in the final generated code to be `p*x`, rather than `x*x*x*x`.

Finally, the sequencing process is changed to first convert all branch nodes into statement nodes, which involves recursively translating the true and false PEG contexts inside the branch node to convert them to statements. This additional step is labeled "convert branches to statements" in Figure 8.12. The final SIMPLE program has only one if-then-else which simultaneously calculates both of the desired results of the branches, as one would expect.

To summarize, the process described so far is to (1) translate *eval* nodes into loop nodes, (2) translate ϕ nodes into branch nodes, (3) perform fusion of loop nodes,



```
if(c)
then
 {p:=0;
  q:=0}
else
 {p:=x*x*x;
  q:=p*x}
```

```
c:=(0=x*x);
if(c)
then
 {p:=0;
  q:=0}
else
 {p:=x*x*x;
  q:=p*x};
retvar:=p+q
```

Figure 8.12. Fusion of branch nodes

(4) perform fusion of branch nodes, and (5) sequence. As with loop fusion, it is important to perform fusion of branch nodes after each and every ϕ node has been converted to branch nodes. Otherwise one may end up fusing two branch nodes where one branch node is used under some ϕ and the other is used unconditionally.

If two branch nodes share the same guard-condition input but one is a descendant of the other, we can even elect to fuse them vertically, as shown in Figure 8.13. In particular, we sequence the true PEG context of one branch node with the true PEG context of the other, and do the same with the false PEG contexts. Note how, because the node labeled ⓘ is used elsewhere than just as an input to branch node ①, we added it as an output of the fused branch node.

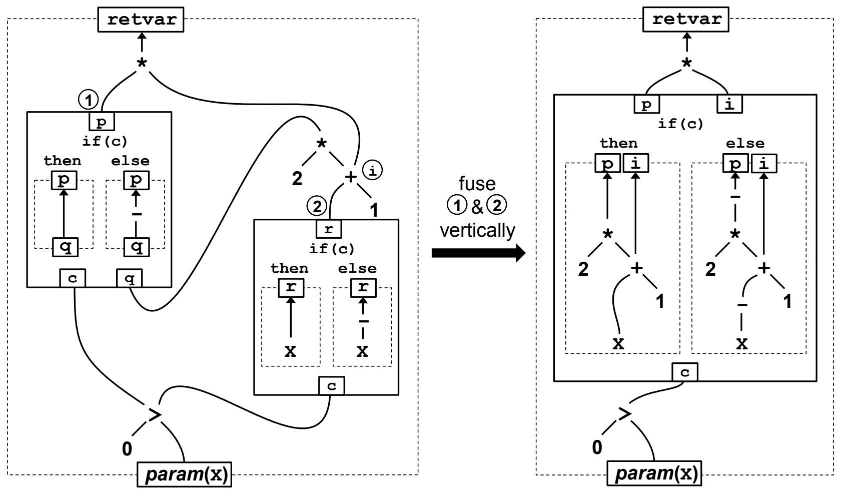


Figure 8.13. Vertical fusion of branch nodes

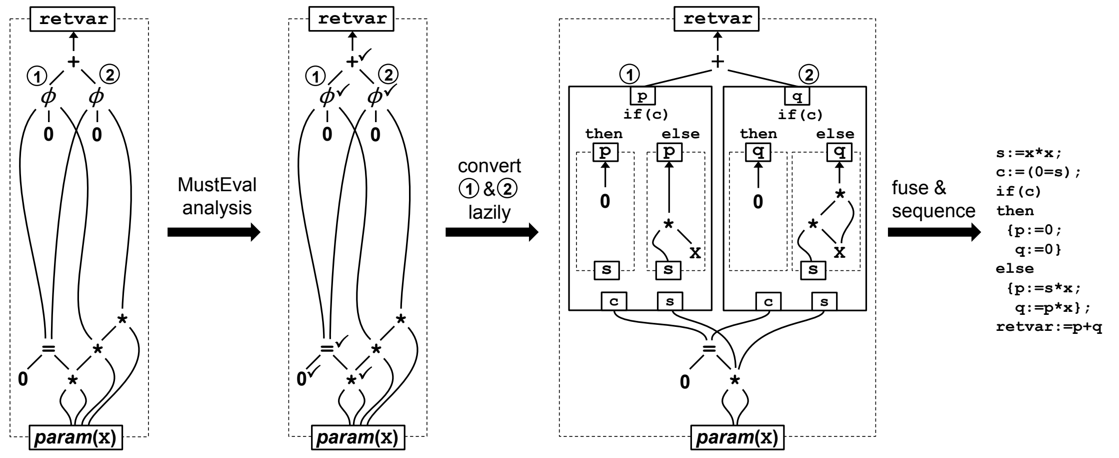
8.8 Hoisting Redundancies from Branches

Looking back at the branch-fusion example from Figures 8.11 and 8.12, there is still one inefficiency in the generated code. In particular, `x*x` is computed in the false side of the branch even though `x*x` has already been computed before the branch.

In our original description for converting ϕ nodes in Section 8.4, we tried to avoid this kind of redundant computation by looking at the set of nodes that are reachable from both the true and false children (second and third children) of a ϕ node. This set was meant to capture the nodes that, for a given ϕ, are need to be computed regardless of which side the ϕ node goes – we say that such nodes execute unconditionally with respect to the given ϕ node. These nodes were kept outside of the branch node (or the if-then-else statement node if using such nodes). As an example, the node labeled ⓐ in Figure 8.5 was identified as belonging to this set when translating ϕ node ②, and this is why the generated if-then-else statement node does not contain node ⓐ, instead taking it as an input (in addition to the `c1` input which is the branch condition).

It is important to determine as completely as possible the set of nodes that execute unconditionally with respect to a ϕ. Otherwise, code that intuitively one would think of executing unconditionally outside of the branch (either before the branch or after it) would get duplicated in one or both sides of the branch. This is precisely what happened in Figure 8.11: our approach of computing the nodes that execute unconditionally (by looking at nodes reachable from the true and false children) returned the empty set, even though `x*x` actually executes unconditionally. This is what lead to `x*x` being duplicated rather than being kept outside of the branch (in the way that ⓐ was in Figure 8.5). A more precise analysis would be to say that a node executes unconditionally with respect to a ϕ node if it is reachable from the true and false children of the ϕ (second and third children) *or* from the branch condition (first child). This would identify `x*x` as being executed unconditionally in Figure 8.5. However, even this more precise analysis has limitations. Suppose for example that some node is used only on the true side of the ϕ node, and never used by the condition, so that the more precise analysis would not identify this node as always executing. However, this node could be used unconditionally higher up in the PEG, or alternatively it could be the case that the condition of the ϕ node is actually equivalent to true. In fact, this last possibility points to the fact that computing exactly what nodes execute unconditionally with respect to a ϕ node is undecidable (since it reduces to deciding if a branch is taken in a Turing-complete computation). However, even though the problem is undecidable, more precision leads to less code duplication in branches.

MustEval Analysis To modularize the part of the system that deals with identifying nodes that must be evaluated unconditionally, we define a MustEval analysis. This analysis returns a set of nodes that are known to evaluate unconditionally in the current PEG context. An implementation has a lot of flexibility in how to define the MustEval analysis. More precision in this analysis leads to less code duplication in branches.



```
s:=x*x;
c:=(0=s);
if(c)
then
  {p:=0;
   q:=0}
else
  {p:=s*x;
   q:=p*x};
retvar:=p+q
```

Figure 8.14. Same example as in Figure 8.11, but this time hoisting redundancies from branches

Figure 8.14 shows the example from Figure 8.11 again, but this time using a refined process that uses the MustEval analysis. The nodes which our MustEval analysis identifies as always evaluated have been marked, including x*x. After running the MustEval analysis, we convert all ϕ nodes which are marked as always evaluated. All remaining ϕ nodes will be pulled into the resulting branch nodes and so handled by recursive calls. Note that this is a change from Section 8.4, where ϕ nodes were processed starting from the lower ones (such as node ② in Figure 8.5) to the higher ones (such as node ① in Figure 8.5). Our updated process, when running on the example from Figure 8.5, would process node ① first, and in doing so would place node ② in the true PEG context of a branch node. Thus, node ② would get processed later in a recursive reversion.

Going back to Figure 8.14, both ϕ nodes are marked as always evaluated, and so we process both of them. To have names for any values that are always computed, we assign a fresh variable to each node that is marked by the MustEval analysis, reusing the same fresh variables for each ϕ node we convert. For example, in Figure 8.14 we use the fresh variable s for the node x*x. We then produce a t and f node for each ϕ node as before, replacing nodes that will always be evaluated with parameter nodes of appropriate variables. Figure 8.11 shows that t for the first ϕ node is simply `0` whereas f is `s*x`, using `s` in place of `x*x`. After all the ϕ nodes have been converted, we perform branch fusion and sequencing as before. Figure 8.11 shows this updated process. The resulting code now performs the `x*x` computation before the branch.

One subtlety is that the MustEval analysis must satisfy some minimal precision requirements. To see why this is needed, recall that after the MustEval analysis is run, we now only process the ϕ nodes that are marked as always evaluated, leaving the remaining ϕ nodes to recursive invocations. Thus, if MustEval does not mark any nodes as being always evaluated, then we would not process any ϕ nodes, which is a problem since after the ϕ-processing phase we require there to be no more ϕ nodes in the PEG. As a result, we require the MustEval analysis to be *minimally precise*, as formalized in the following definition.

Definition 8.2. *We say that a MustEval analysis is minimally precise if for any PEG context Ψ, $S = MustEval(\Psi)$ implies the following properties:*

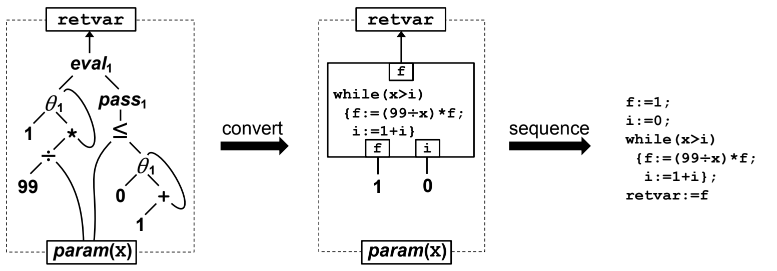
$$\begin{array}{rcl}
(x,n) \in \Psi & \Rightarrow & n \in S \\
\overline{op}(n_1 \ldots n_k) \in S & \Rightarrow & n_1 \in S \wedge \ldots \wedge n_k \in S \\
\overline{\phi}(c,a,b) \in S & \Rightarrow & c \in S \\
\overline{\langle s \rangle}(n_1 \ldots n_k) \in S & \Rightarrow & n_1 \in S \wedge \ldots \wedge n_k \in S
\end{array}$$

In essence the above simply states that MustEval must at least return those nodes which can trivially be identified as always evaluated. To see why this is sufficient to guarantee that we make progress on ϕ nodes, consider the worst case, which is when MustEval returns nothing more than the above trivially identified nodes. Suppose we have a ϕ node that is not identified as always evaluated. This node will be left to recursive invocations of reversion, and at some point in the recursive invocation chain, as we translate more and more ϕ nodes into branch nodes, our original ϕ node will become a top-level node that is always evaluated (in the PEG context being processed in the recursive invocation). At that point we will process it into a branch node.

8.9 Loop-Invariant Code Motion

The process described so far for converting *eval* nodes in Section 8.3 duplicates code unnecessarily. In particular, every node used by the loop body is copied into the loop body, including loop-invariant nodes. Figure 8.15 shows how placing the loop-invariant operation `99÷x` into the loop body results in the operation being evaluated in every iteration of the loop. In the same way that in Section 8.7 we updated our processing of ϕ nodes to hoist computations that are common to both the true and false sides, we can also update our processing of *eval* nodes to hoist computations that are common across all loop iterations, namely loop-invariant computations.

Recall that in Section 6.3 we defined a predicate $invariant_\ell(n)$ which is true if the value of n does not vary in loop ℓ, meaning that n is invariant with respect to loop ℓ. The general approach will therefore be as follows: when converting $eval_\ell$, we simply keep any node n that is invariant with respect to ℓ outside of the loop body. Unfortunately, there are some subtleties with making this approach work correctly. For example, consider the PEG from Figure 8.15. In this PEG the $\div$ node is invariant with respect to loop 1 (`99÷x` produces the same value no matter what iteration the execution is at). However, if we were to evaluate the loop-invariant operation `99÷x` before the loop, we would change the semantics of the program. In particular, if `x` were 0, the $\div$ operation would fail, whereas the original program would simply terminate and return 1 (because the original program only evaluates `99÷x` if `x` is strictly greater than 0). Thus, by pulling the loop-invariant operation out of the loop, we have changed the semantics of the program.



```
while(x>i)
 {f:=(99÷x)*f;
  i:=1+i}
```

```
f:=1;
i:=0;
while(x>i)
 {f:=(99÷x)*f;
  i:=1+i};
retvar:=f
```

Figure 8.15. Reversion of a PEG without applying loop-invariant code motion

Even traditional formulations of loop-invariant code motion must deal with this problem. The standard solution is to make sure that pulling loop-invariant code outside of a loop does not cause it to execute in cases where it would not have originally. In our PEG setting, there is a simple but very conservative way to guarantee this: when processing an $eval_\ell$ node, if we find a node n that is invariant with respect to ℓ, we pull n outside of the loop only if there are no θ or ϕ nodes between the $eval_\ell$ node and n. The intuition behind disallowing ϕ and θ is that both of these nodes can bypass evaluation of some of their children: the ϕ chooses between its second and third child, bypassing the other, and the θ node can bypass its second child if the loop performs no iterations. This requirement is more restrictive than it needs to be, since a ϕ node always evaluates its first child, and so we could even allow ϕ nodes, as long as the loop-invariant node was used in the first child, not the second or third. In general, it is possible to modularize the decision as to whether some code executes more frequently than another in an *evaluation-condition analysis*, or EvalCond for short. An EvalCond analysis would compute for every node in the PEG context an abstract evaluation condition capturing under which cases the PEG node is evaluated. EvalCond is a generalization of the MustEval analysis from Section 8.8, and, as with MustEval, an implementation has a lot of flexibility in defining EvalCond. In the more general setting of using an EvalCond analysis, we would only pull out a loop-invariant node if its evaluation condition is implied by the evaluation condition of the *eval* node being processed.

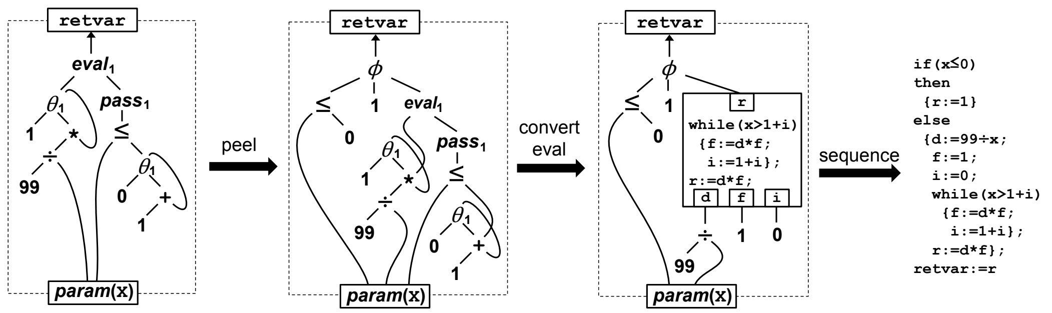
Rewrite $eval_\ell(a,pass_\ell(c))$ to $\phi(eval_\ell(c,\mathrm{Z}),eval_\ell(a,\mathrm{Z}),eval_\ell(peel_\ell(a),pass_\ell(peel_\ell(c))))$ to start loop peeling. Then apply the following rewrite rules until completion:

$$peel_\ell(n) \text{ rewrites to } \begin{cases} n = \theta_\ell(a,b) & b \\ invariant_\ell(n) & n \\ \textbf{otherwise } n = op(a_1,\ldots,a_k) & op(peel_\ell(a_1),\ldots,peel_\ell(a_k)) \end{cases}$$

$$eval_\ell(n,\mathrm{Z}) \text{ rewrites to } \begin{cases} n = \theta_\ell(a,b) & a \\ invariant_\ell(n) & n \\ \textbf{otherwise } n = op(a_1,\ldots,a_k) & op(eval_\ell(a_1,\mathrm{Z}),\ldots,eval_\ell(a_k,\mathrm{Z})) \end{cases}$$

Figure 8.16. Rewrite process for loop peeling

Since we now prevent loop-invariant code from being hoisted if it would execute more often after being hoisted, we correctly avoid pulling `99÷x` out of the loop. However, as is well known in the compiler literature [6], even in such cases it is still possible to pull the loop-invariant code out of the loop by performing loop peeling first. For this reason, we perform loop peeling in the reversion process in cases where we find a loop-invariant node that (1) cannot directly be pulled out because doing so would make the node evaluate more often after hosting and (2) is always evaluated provided the loop iterates a few times. Loop peeling in the reversion process works very similarly to loop peeling as performed in the engine (see Section 4.3) except that, instead of using equality analyses, it is performed destructively on the PEG representation. Figure 8.16 shows the rewrite process for loop peeling on a CFG-like PEG. Using the same starting example as before,



```
while(x>1+i)
 {f:=d*f;
  i:=1+i};
r:=d*f;
```

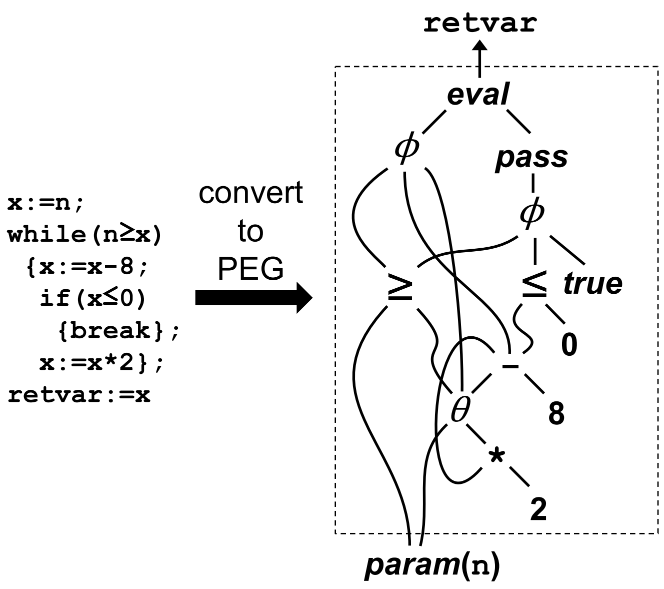
```
if(x≤0)
then
 {r:=1}
else
 {d:=99÷x;
  f:=1;
  i:=0;
  while(x>1+i)
   {f:=d*f;
    i:=1+i};
  r:=d*f};
retvar:=r
```

Figure 8.17. Reversion of a PEG after peeling the loop once

Figure 8.17 shows the result of this peeling process (step labeled “peel”). After peeling, the ϕ node checks the entry condition of the original loop and evaluates the peeled loop if this condition fails. Notice that the *eval* and $\leq$ nodes in the new PEG loop refer to the second child of the θ nodes rather than θ nodes themselves, effectively using the value of the loop variables after one iteration. An easy way to read such nodes is to simply follow the edges of the PEG; for example, the $+$ node can be read as “$1+\theta_1(\ldots)$”.

In general, we repeat the peeling process until the desired loop-invariant nodes used by the body of the loop are also used before the body of the loop. In our example from Figure 8.17, only one run of peeling is needed. Notice that, after peeling, the $\div$ node is still loop-invariant, but now there are no ϕ or θ nodes between the *eval* node and the $\div$ node. Thus, although `99÷x` is not always evaluated (such as when `x≤0` is true), it is always evaluated whenever the *eval* node is evaluated, so it is safe to keep it out of the loop body. As a result, when we convert the *eval* nodes to loop nodes, we no longer need to keep the $\div$ node in the body of the loop, as shown in Figure 8.17. Figure 8.17 also shows the final generated SIMPLE program for the peeled loop. Note that the final code still has some code duplication: `1+i` is evaluated multiple times in the same iteration, and `d*f` is evaluated both when the while-loop guard succeeds and when it fails. These redundancies are difficult to remove without using more advanced control structures that are not present in SIMPLE. Our implementation can take advantage of more advanced control structures to remove these remaining redundancies. We explain these techniques in Section 8.10.

We should also note that, since the EvalCond analysis can handle loop operators and subsumes the MustEval analysis, it is possible to convert ϕ nodes before converting *eval* nodes, although both still need to happen after the loop-peeling phase. This rearrangement enables more advanced redundancy elimination optimizations.



```
x:=n;
while(n≥x)
 {x:=x-8;
  if(x≤0)
   {break};
  x:=x*2};
retvar:=x
```

Figure 8.18. Conversion of a SIMPLE program with a `break` to a PEG

8.10 Advanced Control Flow

So far we have been reverting PEGs to SIMPLE programs. However, SIMPLE has simplistic control flow, whereas more realistic languages such as Java have more advanced and very useful control flow, specifically `break`s and `continue`s. Although programs using these constructs can be translated to use only basic branch and loop structures, this translation often results in less efficient and more complicated CFGs. In order to revert PEGs resulting from realistic languages to efficient CFGs, the reversion process needs to be able to handle advanced control structures. Here we explain how the reversion process can be revised to accomplish this.

Figure 8.18 shows how a SIMPLE program with a `break` is represented as a PEG. The PEG is significantly more complicated since `break` introduces a new exit point for the loop. In particular, the loop has two break conditions: the negation of the loop guard, and the guard of the `break` statement. The ϕ node used by the *pass* node indicates that if the loop guard fails then the loop ends, otherwise if the `break` guard passes then the loop ends. The ϕ node used by the *eval* node indicates that if the loop guard holds in the last iteration of the loop then the loop must have broken via the `break` statement so it must return the state of `x` at that point (which is the state of `x` at the beginning of the loop iteration minus 8), otherwise the loop must have ended due to the loop guard failing so it can simply return the state of `x` at the beginning of the loop iteration. In order to untangle this PEG and revert it to an efficient CFG, we must take advantage of the fact that the result of the loop and the break condition of the loop are both ϕ nodes with the same guard. This requires a more complex reversion process, but this process rewards us by producing more efficient CFGs even for simple loops. For example, this revised process would prevent the code duplication of `1+i` in the result in Figure 8.17. The revised process is accomplished by making two major changes: introducing *guarded* PEG contexts which are eventually converted to branching CFGs, and changing loop nodes to use these guarded PEG contexts.

Before when we replaced an *eval* node with a loop node, we would create three new PEG contexts: one for the condition, one for the body, and one for the final result. After loop fusion, we would convert each of these to statements and then use the results to construct a while-loop statement node. Now, instead of creating three PEG contexts, we will construct a single PEG context to describe the loop. This single PEG context updates the values of the loop variables if the break condition fails, and it updates the value of the result variable if the break condition passes. This PEG context also has a distinguished node labeled as the guard, which in this case is the break condition of the loop. We replace the *eval* node with a revised loop node which instead stores this guarded PEG context (although it still also remembers the associated pass node). The top-left diagram in Figure 8.19 is the PEG from Figure 8.18. Figure 8.19 shows how the loop is extracted into a guarded PEG context (below) and the *eval* node is replaced with

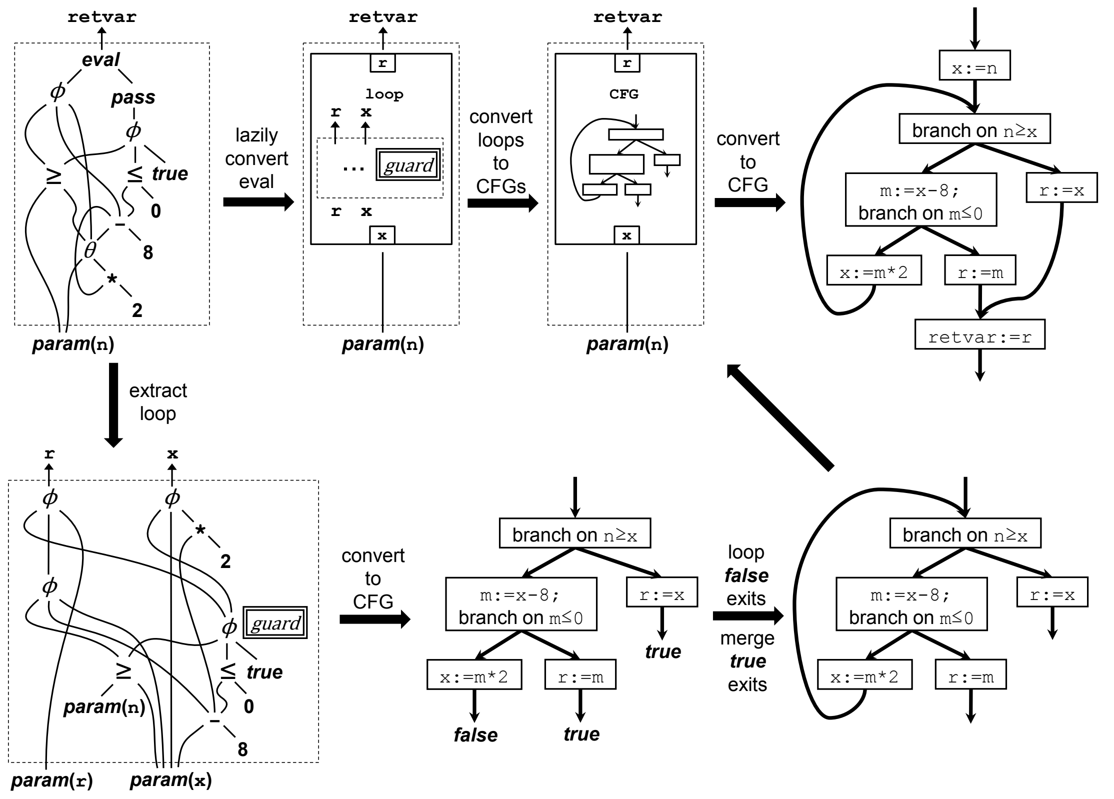


Figure 8.19. Reversion of a PEG to a CFG

the revised loop node containing that guarded PEG context (to the right). The process for constructing the guarded PEG context is described later.

We can still apply loop fusion to these revised loop nodes as we did before by taking the union of the guarded PEG contexts (the distinguished guard node will be the same in the PEG contexts of both loop nodes since they pertain to the same *pass* node). Afterwards, when it comes time to sequentialize the PEG, we need to first turn this loop node into a CFG node (instead of a statement node). A CFG node contains a CFG with a single entry and multiple exits. When sequencing the PEG, all of these exits are routed to whatever CFG block sequencing determines is next after the CFG node; thus a CFG node is simply a generalization of a statement node.

In order to convert a loop node to a CFG node, we first convert the contained guarded PEG context into a special single-entry CFG. This process is described later, but Figure 8.19 shows the CFG resulting from our example guarded PEG context. Each exit in this CFG is marked as either true or false, indicating whether that edge is taken if the distinguished guard node is true or false. Since the distinguished guard node of our guarded PEG context was the break condition of the loop, the exits marked as true are taken when the loop should end, and the exits marked as false are taken when the loop should continue. So, we can turn this CFG into the desired loop by routing all the exits marked as false back to the head of the loop, as exemplified in Figure 8.19. We then replace the loop node with the CFG node containing the resulting CFG, as shown in Figure 8.19.

In the steps described above we left two processes unexplained. The first such process is how to construct the guarded PEG context. We assign a fresh variable x to each relevant θ node as before. Also like before, we construct nodes c, n_x, and r representing the loop's break condition, loop-variable update, and desired result respectively in terms of these fresh variables. Then, we construct a PEG context mapping each θ variable x to $\overline{\phi}(c, \overline{param}(x), n_x)$ and mapping x_r (fresh) to $\overline{\phi}(c, r, \overline{param}(x_r))$. The assignment for the θ variable x means that, if the loop's break condition holds then the loop is ending immediately so there is no need to update the loop variable, otherwise the loop is iterating one more time so we should update the loop variable to its next value. The guarded PEG context in Figure 8.19 says that loop variable x should not change if the loop's break condition holds, and should be updated to $(\mathrm{x} - 8) * 2$ otherwise. The assignment for x_r means that, if the loop's break condition holds then the loop is ending so we should finally calculate the desired value, and otherwise we should do nothing. The use of $\overline{param}(x_r)$ is a slight trick, relying on the fact that no one actually uses this variable until after the loop terminates, by which point it would have been assigned the value of r. The guarded PEG context in Figure 8.19 says that the "result" variable r should not be changed if the loop's break condition is false, should be `x` $-$ 8 if otherwise the loop's guard holds (indicating that the loop exited through the `break` statement), and should be just `x` otherwise (because the loop exited from the beginning). c is the distinguished guard node so that edges marked true or false after translating this guarded PEG context to a CFG (described next) correspond to edges exiting the loop or continuing the loop respectively. Figure 8.19 shows the loop's break condition labeled as the guard.

Lastly, we describe how a guarded PEG context can be reverted to a single-entry CFG with multiple exits marked as either true or false. We step through the reversion of the guarded PEG context constructed in Figure 8.19. This reversion is shown in Figure 8.20. To make the explanation simpler, let us assume that during the entire reversion process the following simplifying replacements are applied automatically should the opportunity arise: a ϕ node whose first child is $\overline{true}$ is replaced with its second child; a ϕ node whose first child is $\overline{false}$ is replaced with its third child; and a node of the form $\overline{\phi}(\overline{\neg}(c),t,f)$ is replaced with $\overline{\phi}(c,f,t)$. How we revert a guarded PEG context depends on the distinguished guard node, so we describe the various cases in turn as we step through the reversion process.

If the distinguished guard node is neither true, false, nor a negation, then we must determine what condition we want to branch on first. This condition may be the distinguished guard node, but if the guard node is a ϕ then the guard node itself depends on another condition, so it would be more efficient to branch on that condition first. To determine the appropriate branch node C, first initialize C as the guard node. While the current value of C is itself a ϕ node, update the value of C to the first child of that ϕ node. Once this completes, C is the node for the first branch condition that must be evaluated. In the top diagram of Figure 8.20, C will be `n`$\geq$`x`. After determining the node C to branch on, we construct a guarded PEG context Ψ_t like the current one except with all uses of C replaced with $\overline{true}$ (and simplifying ϕs automatically). Similarly we construct Ψ_f by

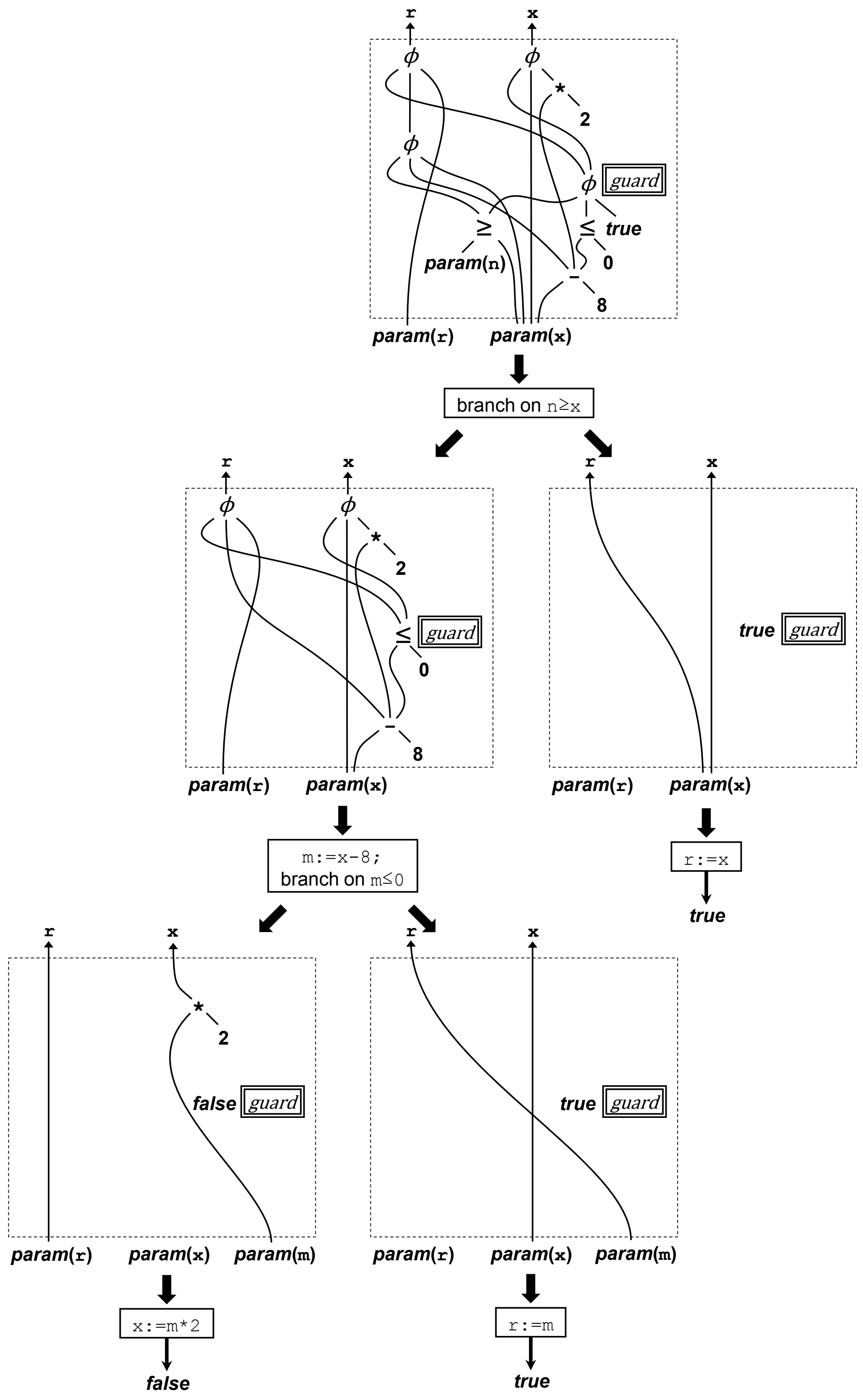


Figure 8.20. Reversion of a guarded PEG context to a CFG with exits marked true/false

replacing C with $\overline{false}$ (and simplifying). We recursively convert each of these guarded PEG contexts to CFGs, then connect them with a CFG block which branches to the first if C is true and to the second if C is false. In our example, the branch for the first stage is `n`$\geq$`x` and the resulting guarded PEG contexts Ψ_t and Ψ_f are shown underneath in Figure 8.20.

In the second stage (reading left to right) we can apply the same process as above, branching on `x-8`$\leq$`0` this time. However, we have an opportunity here to prevent some code duplication. In particular, the node `x-8` will be evaluated regardless of the branch condition, so we can improve by assigning its value to a variable before branching and then having each branch use that variable instead of the original node. Thus after determining the branch condition but before creating Ψ_t and Ψ_f, we determine the set of nodes which will always be evaluated regardless of the branch condition. Then, when creating Ψ_t and Ψ_f, each use of one of these nodes is replaced with a parameter node of a fresh variable, making sure to reuse the same variables across Ψ_t and Ψ_f. We construct a context Ψ which maps each of these fresh variables to its appropriate node and also calculates the branch condition, and then convert Ψ to a CFG. Finally, we route all the exits of this CFG to the branch block, which branches to the CFGs for the revised Ψ_t and Ψ_f as before. In the example in Figure 8.20, Ψ simply assigns `x-8` to the fresh variable `m` and computes the branch condition as `m`$\leq$`0`, so we combined the resulting simple CFG together with the branch block.

The remaining stages in the example all have either a true or false as their distinguished guard node. In these cases, we simply convert the PEG context (without the distinguished guard node) to a CFG. Then we label all the exits of this CFG with either true or false depending on the guard node, as shown in Figure 8.20. The final result of the reversion of this example is the bottom-center diagram in Figure 8.19.

There is one remaining case we need to cover not illustrated in the example: when the distinguished guard node is of the form $\neg(c)$. In this case, we make a recursive call with c as the distinguished guard node, and then simply swap the true and false markings on the exits in the resulting CFG.

We have shown how the improved reversion process can be applied to loops with advanced control structure. The process for converting ϕ nodes can be modified in a similar way. When the conditions of different ϕ nodes are related (say due to short-circuiting), this can produce more efficient CFGs with more advanced control flow. There are more optimizations besides the ones we have presented above, but we feel that these are the most critical to producing efficient CFGs and the most challenging.

Acknowledgements

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 9

Representing Effects

So far we have dealt with mostly pure programs. In Section 7.4 there were some complications due to non-termination, but we mostly bypassed those by allowing the conversion from imperative code to PEGs to improve the termination behavior of a program Yet, in order to handle realistic programs such as occur in Java or LLVM, we needed a way to handle effects. At first we wanted to handle the heap, which had a simple solution that we describe in Section 9.1. Eventually, though, we wanted to handle arbitrary effects such as exceptions and non-determinism. So we informally extended our technique for heap effects to arbitrary effects, as we explain in Section 9.2. Now we finally formalize the semantics of this technique using category theory in Section 9.3.

9.1 Representing the Heap

There are two fundamental operations for working with a heap: *load* and *store*. In imperative code, a *load* takes only one argument: the address in memory to load a value from. However, the value *load* returns also depends on the state of the heap at the time the operation executes. For imperative code, the syntax specifies the order in which operations occur, so the time at which *load* executes is unambiguous. However, PEGs have no order of evaluation, nor do we want them to because it would significantly complicate the equational reasoning we do with PEGs.

To address this problem, we decided to explicitly encode the state of the heap in PEGs. That is, *load* in PEGs takes *two* arguments: the address in memory to load a value from *and* the state of the heap to operate on. Similarly, *store* takes three arguments: the address in memory, the value to store, *and* the state of the heap to operate on. Furthemore, *store* outputs the state of the heap resulting from the operation. Then, if in the imperative code a *load* occurs immediately after a *store*, in the corresponding PEG the output of the translated *store* is routed to the input of the translated *load*, making the sequential order of these operations explicit in the PEG. This technique is similar to the effect witnesses used by Terauchi et al. to incorporate heap operations into pure functional programming [82]. However, our *load* operation does not output a state of the heap, which Terauchi et al. use primarily for execution purposes (and we run into a similar need when revertings effectful PEGs to CFGs as explained in Stepp's thesis [74]).

We can justify this technique semantically by having a heap-summary "value" mapping addresses to values. *load* and *store* can then be defined as standard mathematical functions operation on such a heap summary. So, although this heap summary is not explicit in the imperative code, by making it explicit in the PEG we can do the same equational reasoning we did with pure operations.

9.2 Extending to Arbitrary Effects

Eventually we decided to handle all effects that arise in Java and LLVM. The most common such effect, besides the heap which we had already addressed, was exceptions. For example, a division by zero in Java causes an `ArithmeticException` to be thrown. As such, in order to handle division we needed some way to represent exceptions.

At first, we represented exceptions as explicit control flow. Unfortunately, this became cumbersome both for equational reasoning and for reverting PEGs back to CFGs with exceptions. So, we decided to simply adapt the technique we had used for heap operations to exceptional operations. After all, that technique worked by making the sequencing of heap operations explicit in the PEG, so our intuition was that it should work for arbitrary effects. As such, we made division take *three* inputs, the two explicitly input integers and the effect witness, and produce *two* outputs, the explicitly output integer and the new effect witness. Then effect witnesses were passed between effectful operations to make their order of execution explicit in the PEG. We already briefly showed how to apply this technique to preserving non-termination in Section 7.4.

This subtle adjustment marks a rather significant change. In particular, we moved from the language of expressions, where nodes have multiple inputs and a single output, to a language akin to string diagrams [8, 19], where nodes have multiple inputs and multiple outputs. We dealt with this by having little projection nodes that would select the desired output, making the structure look like an expression, but this was more an implementation artifact rather than an essential technique.

Unfortunately, while effect witnesses make sense intuitively, the semantic justification we used when dealing with heaps does not translate to other effects like exceptions. In particular, division does not take an exception summary and integers and output a new exception summary and integer. If there is an exception, then there is no integer to output, so we cannot use the implicit state concept that worked for heaps. Fortunately, we can still formalize effect witnesses; we just need a more advanced semantic justification technique, namely category theory.

9.3 Categorical Semantics of Effect Witnesses

A category is a very natural way to formalize imperative programs. Morphisms and programs both have an input and an output. Morphisms and programs both have a way to be sequenced provided the inputs and outputs match appropriately. For any object or context, there is an identity morphism or empty program that essentially does nothing except propagate the input directly to the output.

Typical imperative programs have a little more structure though. For example, if we have a program p with input context Γ and output context Γ' and we have some fresh variable $\mathtt{x}$ with type τ, then there is a program with input $\Gamma, \mathtt{x} : \tau$ and output $\Gamma', \mathtt{x} : \tau$ that simply propagates the value of $\mathtt{x}$. We can formalize this categorically using a *premonoidal* category [65]. A premonoidal category has a binary function on objects $\otimes$ which corresponds to combining two contexts together. Also, for any object (i.e. context) $\bar{G}$, there is a functorial function on morphisms mapping $f : G \to G'$ to $f \otimes \bar{G} : G \otimes \bar{G} \to G' \otimes \bar{G}$, corresponding to extend f to propagate the unused context $\bar{G}$. Similarly, there is a functorial function $\bar{G} \otimes f$ propagating the unused context on the left side.

If the imperative language is pure, then there is yet more structure. Suppose we have programs p_1 and p_2 with inputs Γ_1 and Γ_2 and outputs Γ'_1 and Γ'_2 respectively. We can sequence p_1 and p_2 by propagating contexts, using $p_1 \otimes \Gamma_2$ followed by $\Gamma'_1 \otimes p_2$. We can also do the reverse order using $\Gamma_1 \otimes p_2$ followed by $p_1 \otimes \Gamma'_2$. However, since these two programs are pure, their order of execution does not matter, so these two programs will be equivalent semantically. As such, we can simultaneously represent both of them with $p_1 \otimes p_2$, essentially executing p_1 and p_2 side by side. We can formalize this categorically using a *monoidal* category [50]. This is essentially a premonoidal category where $\otimes$ actually forms a *bi*functor, meaning it can combine two morphisms together such that order of execution does not matter.

A typical imperative language, though, is a mix of pure and impure programs. Pure programs are those for which order of execution with any other program is irrelevant. Categorically, this is called the *center* of a premonoidal category: those morphisms $f : G \to G'$ such that for any $\bar{f} : \bar{G} \to \bar{G}'$ the composition of $f \otimes \bar{G}$ followed by $G' \otimes \bar{f}$ is equal to $G \otimes \bar{f}$ followed by $f \otimes \bar{G}'$ (and similarly swapping left and right sides). The center of a premonoidal category will always actually be a monoidal category. When $\otimes$ is actually the categorical product for the center, this is a slightly special case of Freyd categories [66], which are equivalent [42] to arrows [43], which are a generalization of strong monads on Cartesian categories [57], all of which have well known connections to effects. Note that $\otimes$ is well defined on pairs of morphisms when either the left or the right is in the center of the premonoidal category.

Diagrams and Order Irrelevance We have been emphasizing the role of order irrelevance. To understand why, let us explain how these concepts apply to effectful PEGs, or more generally to string diagrams. With both PEGs and string diagrams, nodes sort of float around. There is no clear ordering of the nodes except when the output of one node feeds into the input of another. There can be pairs of nodes where nothing flows, even transitively, from one to the other or vice versa, and so for these nodes the ordering is ambiguous. They are in a sense sitting side by side, and essentially correspond to using $\otimes$ to combine two diagrams by placing them side by side. So, for the semantics of PEGs or string diagrams to make sense, it is important that the order of execution is irrelevant for any nodes or diagrams with no direct flow of information between them.

Monoidal categories guarantee that the order of execution for any pair of disconnected morphisms. As such, string diagrams are actually an internal language for monoidal categories. This is why we had no problems using PEGs for pure computations. Unfortunately, premonoidal categories do not have such a guarantee. We can have impure operations op_1 and op_2, and it is not clear what the semantics for node op_1 sitting next to node op_2 should be since $op_1 \otimes op_2$ is undefined (and cannot be functorially defined since their order of execution affects the semantics). In fact, this tells us that there cannot be *any* monoidal category with morphisms op_1 and op_2.

Partially Monoidal Categories Fortunately, using the concept of effect witnesses, we can turn any premonoidal category into a *partially* monoidal category. By this term we mean a monoidal category except with $\otimes$ being a *partial* bifunctor such that, given $f_1 : G_1 \to G'_1$ and $f_2 : G_1 \to G'_2$, $f_1 \otimes f_2$ is defined whenever $G_1 \otimes G_2$ and $G'_1 \otimes G'_2$ are defined. This means that not all morphisms/diagrams/programs can be placed by side, but whenever it is possible for two such morphisms/diagrams/programs to be connected essentially side by side then their order of execution will be irrelevant. In other words, given a valid PEG or diagram, any rearrangement of that PEG or diagram will also be valid and have the same semantics.

The intuition is that there should be only one effect witness live at any point in time. So, if two diagrams both use effect witnesses then they cannot be placed side by side since that would require two effect witnesses to be live at the same point in time. In other words, if G and G' both contain an effect witness, then $G \otimes G'$ is undefined; otherwise, if at most one of G and G' contains an effect witness, then $G \otimes G'$ is defined.

With that intuition, given a premonoidal category $\mathbf{P}$, we define the following partially monoidal category $\mathbf{EW_P}$ whose string diagrams correspond to our effectful PEGs:

- For any object G of $\mathbf{P}$, we have two objects $\bar{G}$ and $\hat{G}$ corresponding to G without and with an effect witness respectively.
- For any two objects G and G' of $\mathbf{P}$, the morphisms from $\bar{G}$ to $\bar{G}'$ are the morphisms from G to G' in the center of $\mathbf{P}$, and the morphisms from $\hat{G}$ to $\hat{G}'$ are the morphisms from G to G' in $\mathbf{P}$ (and there are no morphisms between $\bar{G}$ and $\hat{G}'$).
- Identity and composition are inherited from $\mathbf{P}$.
- For any two objects G and G' of $\mathbf{P}$, $\bar{G} \otimes \bar{G}'$ is defined as $\overline{G \otimes G'}$, and $\bar{G} \otimes \hat{G}'$ and $\hat{G} \otimes \bar{G}'$ are defined as $\widehat{G \otimes G'}$ (and $\hat{G} \otimes \hat{G}'$ is undefined).

- Given a pair of morphisms $f_1 : G_1 \to G_1'$ and $f_2 : G_2 \to G_2'$ of $\mathbf{EW_P}$, our definition of $\otimes$ on objects guarantees that $f_1 \otimes f_2$ is well defined in $\mathbf{P}$ whenever $G_1 \otimes G_2$ and $G_1' \otimes G_2'$ are defined.

In practice, we actually use different kinds of effect witnesses and will even allow multiple effect witnesses of certain kinds to be live at the same point in time. This has to do with more specialized properties of the effect being witnessed, such as the fact that the state of the heap can be expressed as a heap-summary value. Nonetheless, we leave thorough investigation into such extensions of this technique to future work.

Chapter 10

The Peggy Implementation

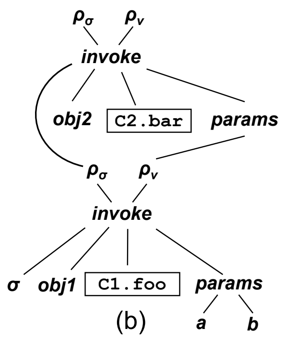
In this chapter we discuss details of our concrete implementation of equality saturation as the core of an optimizer for Java bytecode and LLVM bitcode programs. We call our system Peggy, named after our PEG intermediate representation. As opposed to the previous discussion of the SIMPLE language, Peggy can operate on the entire Java bytecode instruction set or LLVM bitcode instruction set, complete with side effects, method calls, heaps, and exceptions. Recall from Figure 5.1 that an implementation of our approach consists of three components: (1) an IR where equality reasoning is effective, along with the translation functions ConvertToIR and ConvertToCFG, (2) a saturation engine Saturate, and (3) a global profitability heuristic SelectBest. We now describe how each of these three components work in Peggy.

10.1 Intermediate Representation

Peggy uses the PEG and E-PEG representations which, as explained in Chapter 6, are well suited for our approach. Because Java and LLVM are effectful higher-order languages, Peggy has to handle effects and method/function calls. Effects are represented in PEGs usin the techniques described in Chapter 9, but reverting effectful PEGs to CFGs comes with challenges we did not address in Chapter 8. Method/function calls have a few subtleties of their own. Here we describe our solution to these in more explicit detail.

```
C1 obj1 = ...
C2 obj2 = ...
T x = obj1.foo(a,b);
T y = obj2.bar(x);
```

(a)



(b)

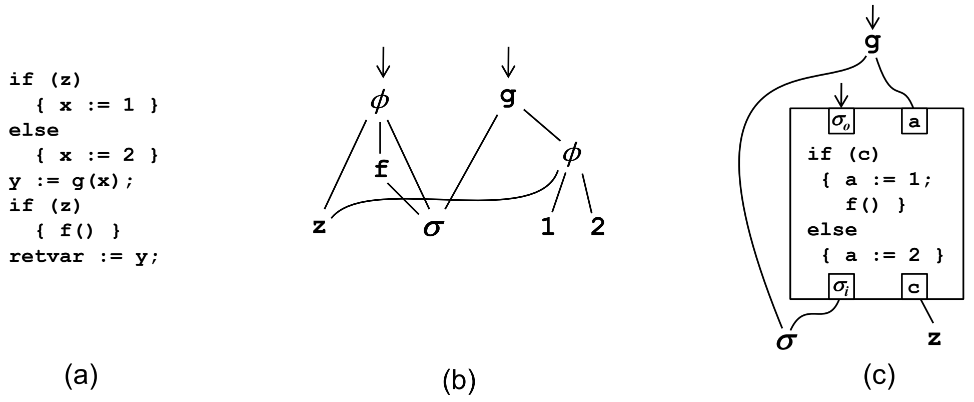
Figure 10.1. Representation of Java method calls in a PEG: (a) the original Java source code and (b) the corresponding PEG

Heap We model the heap using heap summaries which we call σ nodes. Any operation that can read and/or write some object state may have to take and sometimes return additional σ values. Because Java and LLVM stack and register variables cannot be modified except by direct assignments, operations on them are precise in our PEGs and do not involve σ nodes. None of these decisions of how to represent the heap are built into the PEG representation. As with any heap-summarization strategy, one can have different levels of abstraction, and we have simply chosen one where all objects are put into a single summarization object σ. Because we handle other effects as well, and these effects can have their own impact on the heap, we actually use effect witnesses as described in Chapter 9, though we still denote them with σ.

Method Calls Figure 10.1 shows an example of how we encode two sequential method calls in a PEG. Each non-static method call operator has four parameters: the input σ effect witness, the context object of the method call, a method identifier, and a list of actual parameters. A function call or static method call simply elides the context object. Logically, our `invoke` nodes return a tuple (σ, v), where σ is the resulting effect witness and v is the return value of the method. The operator ρ_σ is used to project out the effect witness from this tuple, and ρ_v is used to project out the return value. From this figure we can see that the call to `bar` uses the output effect witness from `foo` as its input effect witness. This is how we encode the control dependency between the two `invoke` operators. Many other Java and LLVM operators have side effects such as array accesses, field accesses, object creation, and synchronization. They similarly take n effect witness as input and sometimes return one as output, encoding the dependencies between the different effectful operations.

Reverting Uses of the Heap One issue that affects our Peggy implementation is that the effect witnesses cannot be duplicated or copied when running the program. More technically, effect witnesses must be used linearly in the CFG. Linear values complicate the reversion presented in Chapter 8, which assumes any value can be duplicated freely. In order to adapt the algorithm from Chapter 8 to handle linear values, one has to "linearize" effect witneses, which consists of finding a valid order for executing operations so that the effect witness does not need to be duplicated. Since PEGs come from actual programs which use the effect witnesses linearly, we decided to linearize the uses of the effect witnesses within each branch node and loop node generated in the reversion algorithm. This approach, unfortunately, is not complete. There are cases (which we can detect while running the conversion algorithm) where partitioning into branch nodes and loop nodes and *then* trying to solve linearity constraints, leads to unsolvable constraints, even though the constraints are solvable when taking a global view of the PEG. Experimentally, this incompleteness occurs in less than 3% of the Java methods we compiled (in which case we simply do not optimize the method). We briefly present the challenges behind solving this problem and some potential solutions.

Consider the SIMPLE style code shown in Figure 10.2(a), and its PEG representation in Figure 10.2(b). Suppose that g is a unary operation that reads the heap and returns a value but does not make any heap modifications. In the PEG representation,



```
if (z)
  { x := 1 }
else
  { x := 2 }
y := g(x);
if (z)
  { f() }
retvar := y;
```

```
if (c)
 { a := 1;
   f() }
else
 { a := 2 }
```

Figure 10.2. Example demonstrating challenges of linearizing heap values.

g takes two inputs, an effect witness plus its explicit input, and g also returns a single value, the computed return value of g, but g does *not* return a new effect witness since it does not modify the heap. We also assume that `f` is a nullary operation that reads and writes to the heap. For example, `f` could just increment a global on the heap. In the PEG representation, `f` takes an input effect witness, and produces an output effect witness. We see from the code in part (a) that its return value is produced by g. However, since `f` may have modified the heap, we must also encode the new effect witness as a return value. Thus, the code in part (a) returns two values: its regular return value, and a new effect witness, as shown with the two arrows in part (b).

Looking at the PEG in Figure 10.2(b), we already see something that may be cause for concern: the σ node, which represents a linear value in imperative code, is used in three different places. However, there is absolutely nothing wrong with doing this from the perspective of PEGs, since PEGs treat all values (including effect witnesses) functionally. The only problem is that, to convert this PEG back to imperative code, we must find a valid ordering of instructions so that we can run all the instructions in the PEG without having to duplicate the effect witness. The ordering in this case is obvious: since `f` modifies the heap, and g does not, run g first, followed by `f`.

The problem with our current approach is that our reversion algorithm first creates and fuses branch nodes, and then it tries to linearize effect witnesses in each branch node. For example, Figure 10.2(c) shows the PEG after creating a branch node for each of the two ϕ nodes in part (a) and fusing these two branch nodes together. Unfortunately, once we have decided to place `f` inside the branch node, the linearization constraints are not solvable anymore: `f` can no longer be executed after g since g relies on the result of the branch node that executes `f`. This example shows the source of incompleteness in our current reversion algorithm for linear values, and points to the fact that one needs to solve the linearization constraints while taking a global view of the PEG.

However, devising a complete linearization algorithm for effect witnesses that takes a global view of the PEG is non-trivial. Effect witnesses can be used by many functions, some of which may commute with others. Other functions may occur in different control paths, such as one on the true path of a branch and the other on the false path. In a PEG, identifying which situations these functions fall in is not a simple task, especially considering that PEGs can represent complex loops and branches. This gets more challenging when using the saturation engine. The saturation engine can determine that a write is accessing a different array index or field than a read, and therefore the write commutes with the read. This information is not available to the reversion algorithm, though, so it cannot determine that the write can be safely moved after the read.

Fortunately, in his thesis Stepp describes a number of techniques that address this issue [74]. We will present a high-level summary here. First, the problem in Figure 10.2 is caused by g not outputing an effect witness. So, Stepp has g output an effect witness so that it is subsequently used by `f`, making the ordering explicit in the PEG. This way any PEG translated from imperative code is guaranteed to be revertible. Then, during equality saturation he adds an equivalence indicating that the output effect witness of g is equivalent to the input effect witness of g, which enables equality saturation to optimize `f` as if its input were the input to `g` as in Figure 10.2. Finally, he modifies the global profitibality heuristic so that it only selects linearizable PEGs. The original PEG, now guaranteed to be linearizable, is still in the E-PEG, so at worst that option will always be available. Thanks to these improvements, while linearization constraints still inhibit optimization, we can at least reliably handle effectful programs.

Equivalences The E-PEG data structure contains a large number of PEGs and stores the equivalences between their nodes. The number of equivalences discovered during saturation can be exponential in the number of axioms applied. So far in this thesis, I have depicted equivalences as dotted lines between E-PEG nodes. In reality, storing a distinct object for each equivalence discovered would require a large amount of memory. Instead, we represent equivalences between nodes by partitioning the nodes into equivalence classes and simply storing the members of each class. Before saturation, every node is in its own equivalence class. Saturation proceeds by merging pairs of equivalence classes A and B together whenever an equality is discovered between a node in A and a node in B. The merging is performed using Tarjan's union-find algorithm [34]. This approach makes the memory overhead of an E-PEG proportional to the number of nodes it contains, requiring little additional memory to track the equivalences between nodes.

10.2 Saturation Engine

The saturation engine's purpose is to repeatedly dispatch equality analyses. In our implementation an equality analysis is a pair (p, f) where p is a trigger, which is an E-PEG pattern with free variables, and f is a callback function that should be run when the pattern p is found in the E-PEG. While running, the engine continuously monitors the E-PEG for the presence of the pattern p. When it is discovered, the engine constructs a *matching substitution*, which is a map from each node in the pattern to the corresponding

```
1: function Saturate($peg : PEG, A$ : set of analyses) : $EPEG$
2: let $epeg = \text{CreateInitialEPEG}(peg)$
3: while $\exists (p, f) \in A, subst \in S$ . $subst = \text{Match}(p, epeg)$ do
4:     $epeg := \text{AddNodesAndEqualities}(epeg, f(subst, epeg))$
5: return $epeg$
```

Figure 10.3. Peggy's Saturation Engine. We use S to denote the set of all substitutions from pattern nodes to E-PEG nodes.

E-PEG node. At this point, the engine invokes f with this matching substitution as a parameter, and f returns a set of equalities that the engine adds to the E-PEG. In this way, an equality analysis will be invoked only when events of interest to it are discovered. Furthermore, the analysis does not need to search the entire E-PEG because it is provided with the matching substitution.

Occasionally the callback f will want to add new does to the E-PEG because it has found a new way to represent values. To do so, f provides the engine with a specification of these new nodes. The engine then checks to see if nodes matching the specification, or congruently equivalent nodes, are already present in the E-PEG. If so, then it imposes the equivalences specified by f onto the existing nodes. If not, it creates new nodes with the appropriate equivalences. This technique significantly reduces our memory usage.

Figure 10.3 shows the pseudocode for Peggy's saturation engine. On line 2, the call to CreateInitialEPEG takes the input PEG and generates an E-PEG that initially contains no equality information. The Match function invoked in the loop condition performs pattern matching: if an analysis trigger occurs inside an E-PEG, then Match returns the matching substitution. Once a match occurs, the saturation engine uses AddNodesAndEqualities to add the nodes and equalities computed by the analysis to the E-PEG.

A remaining concern in Figure 10.3 is how to efficiently implement the existential check on line 3. The main challenge in applying axioms lies in the fact that one axiom application may trigger others. A naïve implementation would repeatedly check all axioms once an equality has been added, which leads to a lot of redundant work since many of the axioms will not be triggered by the new equality. Our original attempt at an implementation used this approach, and it was unusably slow. To make our engine efficient, we use well known techniques from the AI community. In particular, our problem of applying axioms is very similar to that of applying rules to infer facts in rule-based systems, expert systems, or planning systems. These systems make use of an efficient pattern-matching algorithm called the Rete algorithm [36]. Intuitively, the Rete algorithm stores the state of every partially completed match as a finite-state machine. When new information is added to the system, rather than reapplying every pattern to every object, it simply steps the state of the relevant machines. When a machine reaches its accept state, the corresponding pattern has made a complete match. We have adapted this pattern-matching algorithm to the E-PEG domain. The patterns of our Rete network are the preconditions of our axioms. These generally look for the existence of particular sub-PEGs within the E-PEG, but can also check for other properties such as loop invariance. When a pattern is complete it triggers the response part of the axiom, which can build new nodes and establish new equalities within the E-PEG. The creation of new nodes and equalities can cause other state machines to progress, and hence earlier axiom applications may enable later ones.

In general, equality saturation may not terminate. Termination is also a concern in traditional compilers where, for example, inlining recursive functions can lead to unbounded expansion. By using triggers to control when equality edges are added (a technique also used in automated theorem provers), we can often avoid infinite expansion. The trigger for an equality axiom typically looks for the left-hand side of the equality and then makes it equal to the right-hand side. On occasion, though, we use more restrictive triggers to avoid expansions that are likely to be useless. For example, the trigger for the axiom equating a constant with a loop expression used to add edge D in Figure 3.3 also checks that there is an appropriate *pass* expression. In this way, it does not add a loop to the E-PEG if there was not some kind of loop to begin with. Using our current axioms and triggers, our engine completely saturates 84% of the methods in our Java benchmarks.

In the remaining cases, we impose a limit on the number of expressions that the engine fully processes (where fully processing an expression includes adding all the equalities that the expression triggers). To prevent the search from running astray and exploring a single infinitely deep branch of the search space, we currently use a breadth-first order for processing new nodes in the E-PEG. This traversal strategy, however, can be customized in the implementation of the Rete algorithm to better control the searching strategy in those cases where an exhaustive search would not terminate.

10.3 Global Profitability Heuristic

Peggy's SelectBest function uses a pseudo-boolean solver called Pueblo [70] to select which nodes from an E-PEG to include in the optimized program. A pseudo-boolean problem is an integer-linear-programming (ILP) problem where all the variables have been restricted to 0 or 1. By using these 0-1 variables to represent whether or not nodes have been selected, we can encode the constraints that must hold for the selected nodes to be a CFG-like PEG. In particular, for each node or equivalence class x, we define a pseudo-boolean variable that takes on the value 1 (true) if we choose to evaluate x, and 0 (false) otherwise. The constraints then enforce that the resulting PEG is CFG-like. The nodes assigned 1 in the solution that Pueblo returns are selected to form the PEG that SelectBest returns.

Recall that an E-PEG is a quadruple $\langle N, L, C, E\rangle$, where $\langle N, L, C\rangle$ is a PEG and E is a set of equalities inducing a set of equivalence classes N/E. Also recall that for $n \in N$, $params(n)$ is the list of equivalence classes that are parameters to n. We use $q \in params(n)$ to denote that equivalence class q is in the list. For each node $n \in N$, we define a boolean variable B_n that takes on the value true if we choose to evaluate node n, and false otherwise. For equivalence class $q \in (N/E)$, we define a boolean variable B_q that takes on the value true if we choose to evaluate some node in the equivalence class, and false otherwise. We use r to denote the equivalence class of the return value.

Peggy generates the boolean constraints for a given E-PEG $\langle N, L, C, E\rangle$ using the following Constraints function (to simplify exposition, we describe the constraints here as boolean constraints, but these can easily be converted into the standard ILP constraints that Pueblo expects):

$$\begin{array}{lcl} \mathsf{Constraints}(\langle N,L,C,E\rangle) & \equiv & B_r \wedge \bigwedge_{n\in N} \mathsf{F}(n) \wedge \bigwedge_{q\in(N/E)} \mathsf{G}(q) \\ \mathsf{F}(n) & \equiv & B_n \Rightarrow \bigwedge_{q\in params(n)} B_q \\ \mathsf{G}(q) & \equiv & B_q \Rightarrow \bigvee_{n\in q} B_n \end{array}$$

Intuitively, these constraints state that (1) we must compute the return value of the function, (2) for each node that is selected, we must select all of its parameters, and (3) for each equivalence class that is selected, we must compute at least one of its nodes.

Once the constraints are computed, Peggy sends the following minimization problem to Pueblo:

$$\min_{\mathsf{Constraints}(\langle N,L,C,E\rangle)} \sum_{n\in N} B_n \cdot C_n$$

where C_n is the constant cost of evaluating n according to our cost model. The nodes which are set to true in the solution that Pueblo returns are selected to form a PEG.

The cost model that we use assigns a constant cost C_n to each node n. In particular, $C_n = basic\_cost(n) \cdot k^{depth(n)}$, where $basic\_cost(n)$ accounts for how expensive n is by itself, and $k^{depth(n)}$ accounts for how often n is executed. k is simply a constant, which we have chosen as 20. We use $depth(n)$ to denote the loop-nesting depth of n, computed as follows (recalling Definition 6.2 of $invariant_\ell$ from Section 6.3): $depth(n) = \max_\ell \neg invariant_\ell(n)$. Using this cost model, Peggy asks Pueblo to minimize the objective function subject to the constraints described above. Hence, the PEG that Pueblo returns has minimal cost according to our cost model.

The above cost model is very simple, taking into account only the cost of operators and how deeply nested they are in loops. Despite being crude, and despite the fact that PEGs pass through a reversion process that performs branch fusion, loop fusion, and loop-invariant code motion, our cost model is a good predictor of *relative* performance. A smaller cost usually means that, after reversion, the code will use cheaper operators or will have certain operators moved outside of loops, leading to more efficient code. One of the main contributors to the accuracy of our cost model is that $depth(n)$ is defined in terms of $invariant_\ell$, and $invariant_\ell$ is what the reversion process uses for pulling code outside of loops (see Section 8.9). As a result, the cost model can accurately predict at what loop depth the reversion algorithm will place a certain node, which makes the cost model relatively accurate even in the face of reversion.

There is an additional subtlety in the above encoding. Unless we are careful, the pseudo-boolean solver can return a PEG that contains cycles in which none of the nodes are θ nodes. Such PEGs are not CFG-like. For example, consider the expression $x+0$. After axiom application, this expression (namely, the $+$ node) will become equivalent to the x node. Since $+$ and x are in the same equivalence class, the above encoding allows the pseudo-boolean solver to select $+$ with $+$ and 0 as its arguments. To forbid such invalid PEGs, we explicitly encode that all cycles must have a θ node in them. In particular, for each pair of nodes i and j, we define a boolean variable $B_{i \leadsto j}$ that represents whether or not i reaches j without going through any θ nodes in the selected solution. We then state rules for how these variables are constrained. In particular, if a non-θ node i is selected (B_i) then i reaches its immediate children (for each child j of i, $B_{i \rightsquigarrow j}$). Also, if i reaches a non-θ node j in the current solution ($B_{i \rightsquigarrow j}$), and j is selected (B_j), then i reaches j's immediate children (for each child k of j, $B_{i \rightsquigarrow k}$). Finally, we add the constraint that for each non-θ node n, $B_{n \rightsquigarrow n}$ must be false. Note that, in the face of effects, there are additional challenges to maintaining the linearity of effect witnesses, as we explained in Section 10.1. In his thesis, Stepp describes the additional processes and pseudo-boolean constraints that address this additional challenge [74].

It is worth noting that the particular choice of pseudo-boolean solver is independent of the correctness of this encoding. We have chosen to use Pueblo because we have found that it runs efficiently on the types of problems that Peggy generates, but it is a completely pluggable component of the overall system. This modularity is beneficial because it makes it easy to take advantage of advances in the field of pseudo-boolean solvers. In fact, we have tested two other solvers within our framework: Minisat [28] and SAT4J. We have found that occasionally Minisat performs better than Pueblo and that SAT4J uniformly performs worse than the other two. These kinds of comparisons are very simple to do with our framework since we can easily swap one solver for another.

10.4 Eval and Pass

One might wonder why we have *eval* and *pass* as separate operators rather than combining them into a single operator, say μ. At this point we can reflect upon this design decision and argue why we maintain the separation. One simple reason why we maintain this separation is that there are useful operators other than *pass* that can act as the second child to an *eval*. The loop-peeling example from Section 4.3 gives three such examples, namely S, Z, and ϕ. It is also convenient to have each loop represented by a single node, namely the *pass* node. This does not happen when using μ nodes, since there would be many μ nodes for each loop. These μ nodes would all share the same break condition, but we illustrate below why that does not suffice.

Suppose that during equality saturation, some expensive analysis decides the engine should explore peeling a loop. Using *eval* and *pass*, this expensive analysis could initiate the peeling process by simply replacing the *pass* node of that loop with an appropriate ϕ node. Afterward, simple axioms would apply to each *eval* node independently in order to propagate the peeling process. Using μ nodes on the other hand, the expensive analysis would have to explicitly replace every μ node with its peeled version. Thus, using *eval* and *pass* allows the advanced analysis to initiate peeling only once, whereas using μ nodes requires the advanced analysis to process each μ node separately.

Next we consider our global profitability heuristic in this situation after loop peeling has been performed. Now for every *eval* or μ node there are two versions: the peeled version and the original version. Ideally we would select either only peeled versions or only original versions. If we mix them up, this forces us to have two different versions of the loop in the final result. In our pseudo-boolean heuristic with *eval* and *pass* nodes, we encourage the use of only one loop by making *pass* nodes expensive; thus the solver would favor PEGs with only one *pass* node (i.e. one loop) over two *pass* nodes. However, there is no way to encourage this behavior using μ nodes as there is no single node which represents a loop. The heuristic would select the peeled versions for those μ nodes where peeling was beneficial and the original versions for those μ nodes where peeling was detrimental, in fact encouraging an undesirable mix of peeled and original versions.

For similar reasons, the separation of *eval* and *pass* nodes is beneficial to the process of reverting PEGs to CFGs. A loop is peeled by rewriting the *pass* node, and then all *eval* nodes using that *pass* node are automatically peeled simultaneously. Thus, when an optimization such as loop-invariant code motion determines that a loop needs to be peeled, the optimization needs to make only one change and the automatic rewriting mechanisms will take care of the rest.

To summarize, the separation of *eval* and *pass* nodes makes it easy to ensure that any restructuring of a loop is applied consistently: the change is just made to the *pass* node and the rest follows suit. This allows restructuring analyses to apply once and be done with. The separation also enables us to encourage the global profitability heuristic to select PEGs with fewer loops.

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

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*, volume 7, issue 1, 2011. The

dissertation author was the primary investigator and author of this paper.

Chapter 11

Evaluation of 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 11.1), and it is effective at discovering both simple and intricate optimization opportunities (Section 11.2).

11.1 Time and Space Overhead

To evaluate the running time of the various Peggy components, we compiled SpecJVM, which comprises 2,461 methods. 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 four 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 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 200MB, 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 the engine ran until fully processing 500 expressions in the E-PEG 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 200MB of heap, our E-PEGs represented more than 2^{103} versions of the input program (using geometric average).

11.2 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 11.1(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 11.1(b) shows the optimizations that we observed fall out from our approach using equality analyses 1 through 6, and Figure 11.1(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

(a) EQ Analyses	**Description**
1. Built-in E-PEG 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. Tail-Rec Elim	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. Inter-Loop SR	Optimization described in Chapter 3
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 Distrib	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 Fact	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 CSE	Removes redundant computations based on domain axioms
24. Temp-Obj Rem	Remove temp objects made by calls to, e.g., matrix libraries
25. Math Lib Spec	Spec matrix algs based on, e.g., the size of the matrix
26. Design-Patt Opts	Remove overhead of common design patterns
27. Method Outlining	Replace code by method call performing same computation
28. Spec Redirect	Replace call with more efficient call based on calling context

Figure 11.1. Optimizations performed by Peggy. Throughout this table we use the following abbreviations: EQ means “equality”, DS means “domain-specific”, CSE means “common-subexpression elimination”, SR means “strength reduction”

code when isolated from the rest of the program.

Figure 11.2 shows our experimental results for the run times of the micro-benchmarks listed in Figure 11.1(b) and (c). The y-axis shows run time normalized to the run time of the unoptimized code. Each number along the x-axis is a micro-benchmark exemplifying the optimization from the corresponding row number in Figure 11.1. The "rt" and "sp" columns correspond to our larger ray-tracer 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 intraprocedural optimizations turned on, which include: common-subexpression 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 run-time 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 run-time improvements, and in some cases makes the program run slower. For the larger ray-tracer 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)

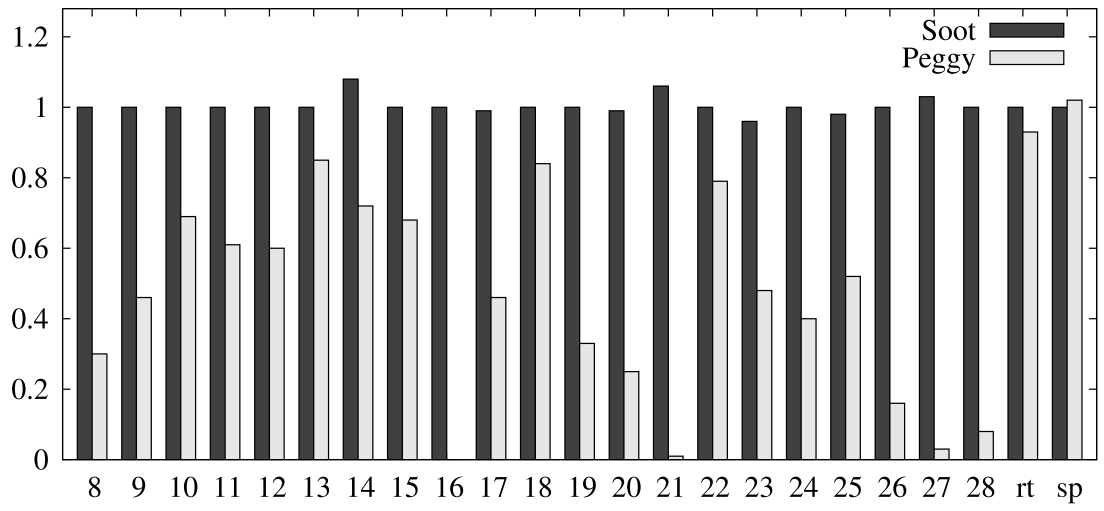


Figure 11.2. Run times of generated code from Soot and Peggy, normalized to the runtime of the unoptimized code. The x-axis denotes the optimization number from Figure 11.1, where “rt” is our raytracer benchmark and “sp” is the average over the SpecJVM benchmarks.

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 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 (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 pure-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 11.1).

Our second example targets a common programming idiom involving `Lists`, 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 $\texttt{contains}(e)$ and a call to $\texttt{indexOf}(e)$ 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 $l.\texttt{contains}(e) = (l.\texttt{indexOf}(e) \neq \texttt{-1})$, Peggy can convert the clean code into the hand-optimized code that programmers typically write, which stores $\texttt{indexOf}(e)$ 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 $l.\texttt{contains}(e)$ to $(l.\texttt{indexOf}(e) \neq \texttt{-1})$ 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 E-PEG, 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 11.1 is presented in Appendix A.

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 12

Translation Validation

Equality saturation 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.

12.1 Overview

To illustrate this in more detail, we first present several examples demonstrating how Peggy performs translation validation. These examples are distilled versions of real examples that we found while doing translation validation for LLVM on SPEC 2006.

Example 1 Consider the original code in Figure 12.1(a) and the optimized code in Figure 12.1(b). There are two optimizations that LLVM applied here. First, LLVM performed copy propagation through the location `*p`, thus replacing `*p` with `t`. Second, LLVM removed the now-useless store `*p := t`.

```
int f(p,t) {
  *p := t
  *p := *p | (t & 4)
  return 0
}
```

(a)

(b)

```
int g(p,t) {
  *p := t | (t & 4)
  return 0
}
```

(c)

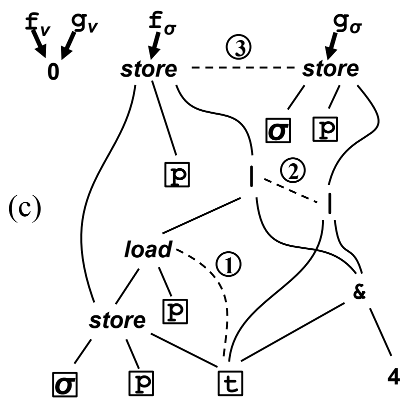


Figure 12.1. (a) Original code (b) Optimized code (c) Combined E-PEG

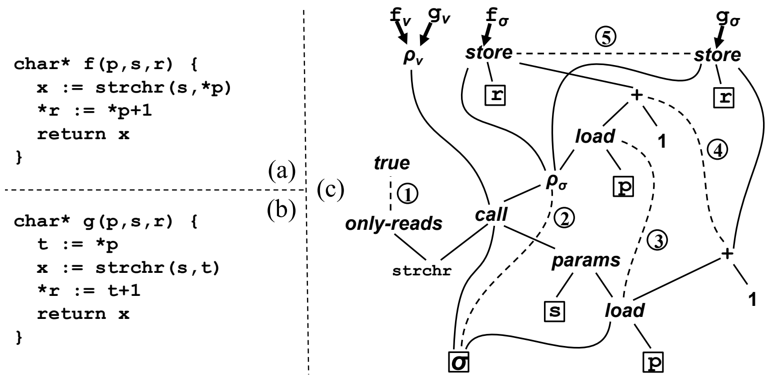
Figure 12.1(c) shows the E-PEG for `f` and `g` combined. The labels $\mathtt{f}_v$ and $\mathtt{f}_\sigma$ point to the value and effect witness returned by `f` respectively, and likewise for `g` – for now, let us ignore the dashed lines. Nodes with a square around them represent parameters to the function.

Our approach to translation validation builds the PEGs for both the original and the optimized programs in the same E-PEG, reusing nodes when possible. In particular, note how `t & 4` is shared. Once this combined E-PEG has been constructed, we apply equality saturation. If through this process Peggy infers that node $\mathtt{f}_\sigma$ is equivalent to node g_σ and that node $\mathtt{f}_v$ is equivalent to node g_v, then Peggy has shown that the original and optimized programs are equivalent.

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

1. Peggy adds equality ① using axiom $load(store(\sigma, p, v), p) = v$
2. Peggy adds equality ② by congruence closure: $a = b \Rightarrow f(a) = f(b)$
3. Peggy adds equality ③ using axiom $store(store(\sigma, p, v_1), p, v_2) = store(\sigma, p, v_2)$

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



```
char* f(p,s,r) {
  x := strchr(s,*p)
  *r := *p+1
  return x
}
```

```
char* g(p,s,r) {
  t := *p
  x := strchr(s,t)
  *r := t+1
  return x
}
```

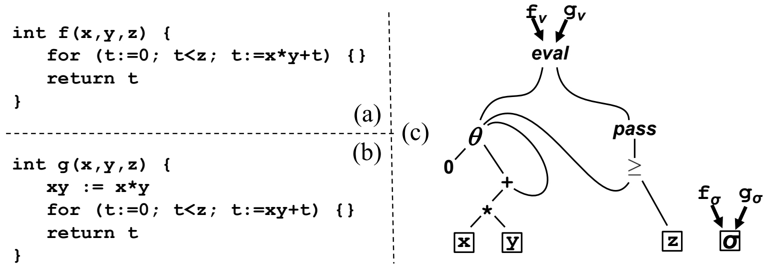
Figure 12.2. (a) Original code (b) Optimized code (c) Combined E-PEG

Example 2 As a second example, consider the original function `f` in Figure 12.2(a) and the optimized version g in Figure 12.2(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 E-PEG are shown in Figure 12.2(c). The call to `strchr` is represented using a *call* node, which has three children: the name of the function, the incoming effect witness, 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 effect witness. We use projection operators ρ_v and ρ_σ to extract the return value and the effect witness from the pair returned by a *call* node.

To give Peggy the knowledge that standard-library functions such as `strchr` do not modify the heap, we have annotated such standard-library functions with an *only-reads* annotation. Whenever a function *foo* is annotated with *only-reads*, Peggy adds the equality $\textit{only-reads}(\textit{foo}) = \textbf{true}$ to the E-PEG. Equality ① in Figure 12.2(c) is

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



```
int f(x,y,z) {
   for (t:=0; t<z; t:=x*y+t) {}
   return t
}
```

```
int g(x,y,z) {
   xy := x*y
   for (t:=0; t<z; t:=xy+t) {}
   return t
}
```

Figure 12.3. (a) Original code (b) Optimized code (c) Combined E-PEG

added in this way.

Peggy adds equality ② using: $\mathit{only\text{-}reads}(n) = \textbf{true} \Rightarrow \rho_\sigma(\mathit{call}(n,\sigma,p)) = \sigma$. This axiom encodes the fact that a read-only function call does not modify the heap. Equalities ③, ④, and ⑤ are added by congruence closure.

In these five steps, Peggy has identified that the heaps $\mathtt{f}_\sigma$ and g_σ are equal, and since the returned values $\mathtt{f}_v$ and 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 12.3(a) and the optimized code in Figure 12.3(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 12.3. As it turns out, `f` and g will produce the exact same PEG, so let us focus on understanding the PEG itself. 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.

12.2 Implementation

Axioms Peggy uses a variety of axioms to infer equality information. Some of these axioms were previously developed and state properties of built-in PEG operators like θ, *eval*, *pass*, and ϕ. We also implemented LLVM-specific axioms to reason about *load* and *store*, some of which we have already seen. An additonal such axiom is important for moving unaliased loads/stores across each other: $p \neq q \Rightarrow load(store(\sigma, q, v), p) = load(\sigma, p)$.

Alias Analysis The axiom above can only fire if $p \neq q$, requiring alias information. Our first attempt was to encode an alias analysis using axioms and the saturation engine. However, this added a significant run-time overhead, and so we instead took the approach from [84], which is to pre-compute alias information. We then used this information when applying axioms.

Generating Proofs After Peggy validates a transformation it can use the resulting E-PEG to generate a proof of equivalence of the two programs. This proof has already helped us determine how often axioms are useful. In the future, we could also use this proof to improve the run time of our validator: after a function f has been validated, we could record which axioms were useful for f, and enable only those axioms for subsequent validations of f (reverting back to all axioms if the validation fails).

12.3 Results

We used Peggy to perform translation validation for the Soot optimizer [86]. In particular, we used Soot to optimize a set of benchmarks with all of its intraprocedural optimizations turned on. The benchmarks included SpecJVM, 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 equality saturation 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 positives are a consequence of our coarse heap model, a finer-grained model can increase the effectiveness of translation validation and would also enable more optimizations.

Inspired by the recent results of Tristan et al. [84] on translation validation for LLVM, we used Peggy to perform translation validation for LLVM 2.8, a more aggressive and more widely used compiler than Soot, on SPEC 2006 C benchmarks. We enabled the following optimizations: dead-code elimination, global value numbering, partial-redundancy elimination, sparse conditional constant propagation, loop-invariant code motion, loop deletion, loop unswitching, dead-store elimination, constant propagation, and basic-block placement.

Figure 12.4 shows the results. "#Func" and "#Instr" are the number of functions and instructions. "% Success" is the percentage of functions whose compilation Peggy validated ("All" considers all functions. "OC", which stands for "Only Changed", ignoring functions for which LLVM's output is identical to the input). "To PEG" is the average time per function to convert from CFG to PEG. "Avg Eng Time" is the average time per function to run the equality saturation engine ("Success" is over successful runs, and "Failure" over failed runs).

Overall our results are comparable to [84]. However, because of implementation differences (including the set of axioms), an in-depth and meaningful experimental

Benchmark	**#Func**	**#Instr**	**% Success**		**To**	**Avg Eng Time**	
			All	**OC**	**PEG**	**Success**	**Failure**
`400.perlbench`	1,864	269,631	79.0%	73.3%	0.531s	1.028s	11s
`401.bzip2`	100	16,312	82.0%	76.9%	0.253s	0.733s	19s
`403.gcc`	5,577	828,962	80.8%	74.9%	0.558s	0.700s	19s
`429.mcf`	24	2,541	87.5%	87.0%	0.216s	0.500s	19s
`433.milc`	235	21,764	80.4%	75.0%	0.246s	0.188s	9s
`456.hmmer`	538	57,102	86.4%	84.6%	0.285s	0.900s	11s
`458.sjeng`	144	23,807	77.1%	72.5%	1.099s	0.253s	7s
`462.libquantum`	115	5,864	73.9%	64.3%	0.123s	0.167s	8s
`464.h264ref`	590	131,627	74.2%	70.5%	0.587s	0.846s	12s
`470.lbm`	19	3,616	78.9%	76.5%	0.335s	0.154s	3s
`482.sphinx3`	369	28,164	88.1%	86.0%	0.208s	0.480s	12s

Figure 12.4. Results for Peggy's translation validator on SPEC 2006 C benchmarks

comparison is difficult. Nonetheless, conceptually the main difference is that [84] uses axioms for destructive rewrites, whereas we use axioms to add equality information to the E-PEG, thus expressing multiple equivalent programs at once. Our approach has several benefits over [84]:

- We simultaneously explore an exponential number of paths through the space of equivalent programs, whereas [84] explores a single linear path – hence we explore more of the search space.
- We need not worry about axiom ordering, whereas [84] must pick a good ordering of rewrites for LLVM – hence it is easier to adapt our approach to new compilers, and a user can easily add or remove axioms (without worrying about ordering) to balance precision with speed or to specialize for a given code base.
- Our approach effectively reasons about loop-induction variables, which is more difficult using the techniques in [84].

However, the approach in [84] is faster, exploring a single linear path through the space of programs.

Failures were caused by: (1) incomplete axioms for linear arithmetic, (2) insufficient alias information, and (3) LLVM's use of pre-computed interprocedural information even in intraprocedural optimizations. These limitations point to several directions for future work, including incorporating SMT solvers and better alias analyses, as well as investigating interprocedural translation validation.

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

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*, volume 7, issue 1, 2011. The dissertation author was the primary 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*, 2011. The dissertation author was the secondary investigator and author of this paper.

Chapter 13

Learning Optimizations from Proofs

Our translation validator outputs a proof that the two input programs are equivalent. Similarly, our optimizer can output a proof that the optimized program is equivalent to the original program. These proofs of equivalence tell us precisely what parts of the programs mattered and how. If we are careful, we can generalize the programs in a way that retains the validity of the proof structure. Recognizing this, we developed a technique for learning optimizations from just one example of a transformed program given snapshots of the program before and after transformation. This enables us to learn optimizations from programmers without requiring them to understand anything about the compiler they are using.

Intuitively, our approach is to fix the proof structure and then try to find the most general optimization rule that a proof of that structure proves correct. Focusing on a given proof structure also has the added advantage that, once the structure is fixed, we will be able to show that there exists a unique most general optimization rule with that proof structure, something that may not exist ignoring the proof structure. For example, consider the optimization instance $0*0 \Mapsto 0$. This transformation has two incomparable generalizations, $X*0 \Mapsto 0$ and $0*X \Mapsto 0$, depending on whether one uses the axiom $\forall x.x*0=0$ or $\forall x.0*x=0$ to prove correctness. However, once we settle on a given proof of correctness, not only does there exist a most general optimization rule given the proof structure, but we can also show that our algorithm infers it.

In this chapter, we start by giving some examples of proof-based generalization (Section 13.1), explain some of the challenges behind generalization (Section 13.2), give an overview of our algorithm (Section 13.3), and finally describe a way of decomposing optimizations we generate into smaller independent ones (Section 13.4).

13.1 Generalization Examples

As we will show in Chapter 14, our approach is general and can be applied to many kinds of intermediate representations, and even to domains other than compiler optimizations. However, to make things concrete for our examples, we will use PEGs and E-PEGs.

Figure 13.1 shows an example of how our approach works. We describe the process at a high level, and then describe the details of each step. At a high level, we start with two concrete programs, presenting an example of what the desired transformation should do – parts (a) and (b). We convert these programs into PEGs – parts (c) and (d). We then prove that the two programs are equivalent using translation validation as in Chapter 12 – part (e). From the proof of equivalence we then generalize into optimization rules – parts (f) and (g) show two possible generalizations depending on the logic we use.

The starting point of generalization is the annotated E-PEG from Figure 13.1(e), which represents a proof that the PEGs from Figure 13.1(c) and Figure 13.1(d) are equivalent. In particular, edges ⓐ through ⓓ in the E-PEG represent the steps of the equivalence proof: edge ⓐ is added by applying the axiom $\theta(x,y)*z = \theta(x*z, y*z)$; edge ⓑ is added by applying the axiom $(x+y)*z = x*z+y*z$; edge ⓒ is added by applying the axiom $1*x = x$; and edge ⓓ is added by applying the axiom $0*x = 0$. This E-PEG represents many different versions of the original program, depending on how we choose to compute each equivalence class. By picking θ to compute the $\{*, \theta\}$

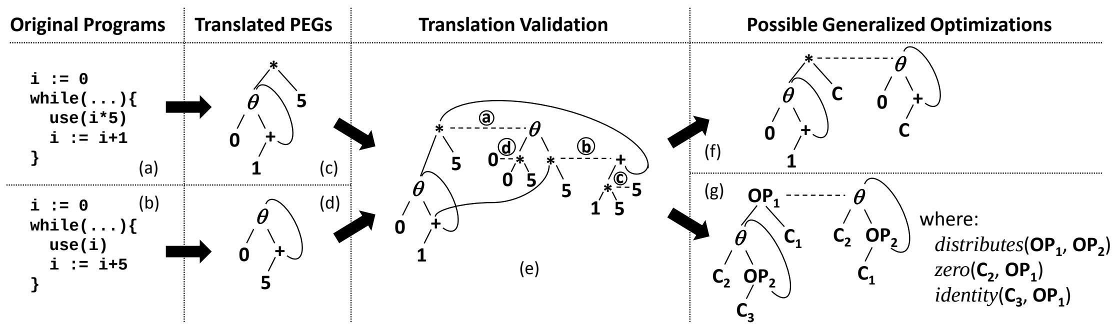


Figure 13.1. Learning loop-induction-variable strength reduction from an example

equivalence class, and $+$ to compute the $\{*,+\}$ equivalence class, we get the PEG from Figure 13.1(d). Thus, the E-PEG shows that the PEGs from parts (c) and (d) are equivalent, and the edge annotations give the proof.

Our goal is to take the conclusion of the proof – in this case edge ⓐ – and determine how one can generalize the E-PEG so that the proof encoded in the E-PEG is still valid. Figures 13.1(f) and 13.1(g) show two possible generalized optimizations that can result from this process. We represent a generalized optimization by an E-PEG containing a single equality edge, representing the conclusion of the proof. There are two ways of interpreting such E-PEGs. One is that it represents a transformation rule, with the single equality edge representing the transformation to perform. The direction of the rule is determined by which of the two programs in the instance was the original, and which was the transformed. Another way to interpret these rules is that they represent equality analyses used in Peggy. Chapter 18 will show that our generated optimizations, when used as equality analyses, make Peggy faster while still producing the same results.

Figure 13.1(f) shows a generalization where the constant 5 has been replaced with an arbitrary pure expression C. The key observation is that the particular choice of constant does not affect the proof – if we have a proof of LIVSR for 5, the same proof holds for an arbitrary constant. Note that PEGs abstract away the details of the control flow graph. As a result the generalizations of Figure 13.1(f) could be applicape to a PEG even if there were many other nodes in the PEG representing various kinds of loops or statements not affecting the induction variable being optimized.

Figure 13.1(g) shows a more sophisticated generalization, where instead of just generalizing constants, we also generalize operators. In particular, the $*$ and $+$ operators have been generalized to OP_1 and OP_2, with the added side condition that OP_1 distributes over OP_2 (there is no need to add a side condition stating that OP_1 distributes over θ since all operators distribute over θ). Furthermore, the constants 0 and 1 have been generalized to C_2 and C_3 with the additional side conditions that C_2 is a zero for OP_1 and C_3 is an identity for OP_2. The generalization in Figure 13.1(g) can apply to operators that have the same algebraic properties as integer plus/multiply, for example boolean OR/AND, vector plus/multiply, set union/intersect, or any other operators for which the programmer states that the side conditions from Figure 13.1(g) hold.

The choice of logic is what makes the difference between the above two generalizations. Figure 13.1(g) results from a proof expressed with a more general logic. Instead of the axiom $(x+y)*z = x*z+y*z$, the proof uses:

$$OP_1(OP_2(x,y),z) = OP_2(OP_1(x,z),OP_1(y,z)) \text{ where } \mathit{distributes}(OP_1,OP_2)$$

and instead of $0*x = 0$, the proof uses:

$$OP(C,x) = C \text{ where } \mathit{zero}(C,OP)$$

The LIVSR example therefore shows that the domain of axioms and proofs affects the kind of generalization that one can perform. More general logics typically lead to more general generalizations. By using category theory to formalize our algorithm, we will be able to abstract away the domain in which axioms and proofs are expressed, thus separating the particular choice of domains from the description of our algorithm. As a result, our algorithm as expressed in category theory will be general enough so that it can be instantiated with many different kinds of domains for proofs and axioms, including those that produce the different generalizations presented above.

13.2 Obtaining the Most General Form

Looking at the LIVSR example, one may think that generalization is as simple as replacing all nodes and operators in an E-PEG with meta-variables, and then constraining the meta-variables based on the axioms that were applied. Although this approach is very simple, it does not always produce the most general optimization rule for a given proof. Consider for example the E-PEG from Figure 13.2(a), where α is some PEG expression. Edge ⓐ is produced by axiom $x+0=x$, edge ⓑ by $x-x=0$, edge ⓒ by $(x+y)-y=x$, edge ⓓ by $0+x=x$, and edge ⓔ by transitivity of edges ⓒ and ⓓ. This E-PEG therefore represents a proof that the sum at the top is equivalent to α. If we replace nodes with meta-variables and constrain the meta-variables based on axiom applications, one would simply generalize α to a meta-variable. However, the most general optimization rule from the proof encoded in Figure 13.2(a) is shown in Figure 13.2(b). The key difference is that by duplicating the shared $+$ node, one can constrain the arguments of the two new $+$ nodes differently. However, because PEGs can contain cycles, one cannot simply duplicate every node that is shared, as this would lead

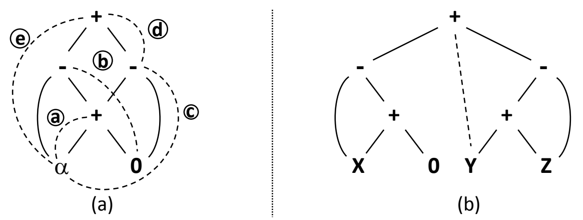


Figure 13.2. Example showing the need for splitting

to an infinite expansion. The main challenge then with getting the most general form is determining precisely how much to split.

13.3 Our Approach

Instead of generalizing the operators in the final E-PEG to meta-variables and then constraining the meta-variables, our approach is to start with a near empty E-PEG, and step through the proof *backwards*, augmenting and constraining the E-PEG as each axiom is applied in the backward direction. This allows us to solve the above splitting problem by essentially turning the problem on its head: instead of starting with the final E-PEG and splitting, we gradually add new nodes to a near-empty E-PEG, constraining and merging as needed. Our algorithm therefore merges only when required by the proof structure, keeping nodes separate when possible.

We illustrate our approach on a very simple example so that we can show all the steps. Consider the E-PEG of Figure 13.3(a), where two axioms have been applied to determine equality edges ⓐ and ⓑ. The axioms are shown in Figure 13.3(c), with each axiom being an E-PEG where one edge has been labeled “P” for “Premise”, and one edge has been labeled “C” for “Conclusion”. The • in the axioms represent meta-variables to be instantiated. The first axiom states $x - x = 0$, and the second axiom states $x + 0 = x$.

Our process is shown in Figure 13.3(b). We start at the top with a single equality edge representing the conclusion of the proof, and then work our way downward by applying the proof in reverse order: in step 1 we apply the second axiom backwards, and then in step 2 we apply the first axiom backwards. Each time we apply an axiom backwards, we create and/or constrain E-PEG nodes in order to allow that axiom to be applied. Figure 13.3 shows using fine-dotted edges how the “Premise” and “Conclusion” edges of the axioms map onto the E-PEGs being constructed. For example, note that in step 1, when the second axiom is applied backwards, we remove the final conclusion

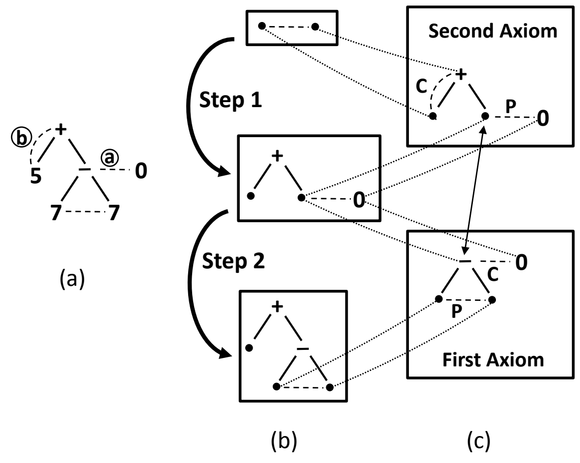


Figure 13.3. Example showing how generalization works

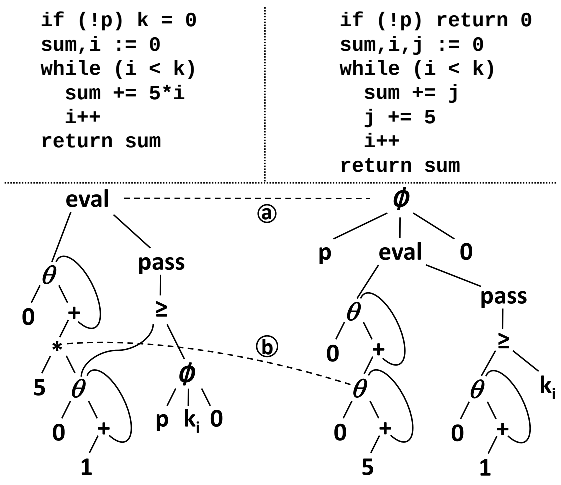
edge, and instead replace it with an E-PEG that essentially represents $\bullet + 0$.

There is an alternate way of viewing our approach. In this alternate view, we instantiate all the axioms that have been applied in the proof with fresh meta-variables, and then use unification to stitch these freshly instantiated axioms together so that they connect in the right way to make the proof work. With this view in mind, we show in Figure 13.3 how the first and second axioms would be stitched together using a bidirectional arrow.

This section has only given an overview of how our approach works. Chapters 14 and 15 will formalize our approach using category theory and revisit the above example in much more detail.

13.4 Decomposition

Even though our algorithm finds the most general transformation rule for a given proof, the produced rule may still be too specific to be reused. This can happen if the input-output example has several conceptually different optimizations happening at the same time. Consider for example the optimization instance shown in Figure 13.4. The



```
if (!p) k = 0
sum,i := 0
while (i < k)
  sum += 5*i
  i++
return sum
```

```
if (!p) return 0
sum,i,j := 0
while (i < k)
  sum += j
  j += 5
  i++
return sum
```

Figure 13.4. One E-PEG with conceptually two optimizations

top of the Figure shows the original and transformed code. There are two independent high-level optimizations. The first is LIVSR, which replaces the `i*5` with a variable `j` that is incremented by `5` each time around the loop; the second is specialization for the true case of the `if(!p)` branch, so that it immediately returns.

The corresponding E-PEG is shown at the bottom of the Figure. The E-PEG does not show all the steps – instead it just displays the final equality edge ⓐ, and an additional edge ⓑ that we will discuss shortly. In the E-PEG, the two optimizations manifest themselves as follows: LIVSR happens using steps similar to those from Figure 13.1 on the PEG rooted at the $*$ node (producing edge ⓑ), and the specialization optimization happens by pulling the ϕ node up through the $\geq$, *pass*, and *eval* nodes (producing edge ⓐ). Each of these optimizations takes several axiom applications to perform, introducing various temporary nodes that are not shown in Figure 13.4.

If we simply apply the generalization algorithm outlined in Section 13.3, we will get a single rule (although generalized) that applies the two optimizations together. However, these two optimizations are really independent of each other in the sense that each can be applied fruitfully in cases where the other does not apply. Thus, in order to learn optimizations that are more broadly applicable, we further *decompose* optimizations that have been generalized into smaller optimizations. One has to be careful, however, because decomposing too much could just produce the axioms we started with.

To find a happy medium, we decompose optimizations as much as possible, subject to the following constraint: we want to avoid generating optimization rules that introduce and/or manipulate temporary nodes (i.e. nodes that are *not* in the original or transformed PEGs). The intuition is that these temporary nodes really embody intermediate steps in the proof, and there is no reason to believe that these intermediate steps individually would produce a good optimization.

To achieve this goal, we pick decomposition points to be equalities between nodes in the generalized original and transformed PEGs (and not intermediate nodes). In particular, we perform decomposition in two steps. In the first step, we generalize the entire proof without any decomposition, which allows us to identify the nodes that are part of the generalized original or final terms. We call such nodes *required*, and equalities between them represent decomposition points. In the second step, we perform generalization again, but this time, if we reach an equality between two required nodes, we take that equality as an assumption for the current generalization and start another generalization beginning with that equality.

In the example of Figure 13.4, we would find one such equality edge, namely edge ⓑ. As a result, our decomposition algorithm would perform two generalizations. The first one starts at the conclusion ⓐ, going backwards from there, but stops when edge ⓑ is reached (i.e. edge ⓑ is treated as an assumption). This would produce a branch-lifting optimization. Separately, our decomposition algorithm would perform a generalization starting with ⓑ as the conclusion, which would essentially produce the LIVSR optimization.

Acknowledgements

This chapter contains material taken from "Generating Compiler Optimization from Proofs", by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

Chapter 14

Proofs in Categories

Having seen an overview of how our approach works, we now give a formal description of our framework for generalizing proofs using category theory. The generality of our framework not only gives us flexibility in applying our algorithm to the setting of compiler optimizations by allowing us to choose the domain of axioms and proofs, but it also makes our framework applicable to settings beyond compiler optimizations. After a quick overview of category theory (Section 14.1), we show how axioms (Section 14.2) and inference (Section 14.3) can be encoded in category theory. We then define what a generalization is (Section 14.4), and finally we show how to construct the most general one (Section 14.5).

14.1 Overview of Category Theory

A category is a collection of objects and morphisms from one object to another. For example, the objects of the commonly used **Set** category are sets, and its morphisms are functions between sets. Not all categories use functions for morphisms, and the concepts we present here apply to categories in general, not only to those where morphisms are functions. Nonetheless, thinking of the case where morphisms are functions is a good way of gaining intuition.

Given two objects $\mathcal{A}$ and $\mathcal{B}$ in a category, the notation $f : \mathcal{A} \to \mathcal{B}$ indicates that f

is a morphism from $\mathcal{A}$ to $\mathcal{B}$. This same information is displayed graphically as follows:

$$\begin{tikzcd}\mathcal{A} \arrow[r, "f"] & \mathcal{B}\end{tikzcd}$$

In addition to defining the collection of objects and morphisms, a category must also define how morphisms *compose*. In particular, for every $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B} \to \mathcal{C}$ the category must define a morphism $f\,;g : \mathcal{A} \to \mathcal{C}$ that represents the composition of f and g. The composition $f\,;g$ is also denoted $g \circ f$. Morphism composition in the **Set** category is simply function composition. Morphism composition must be associative. Also, each object $\mathcal{A}$ of a category must have an identity morphism $id_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ such that $id_{\mathcal{A}}$ is an identity for composition (that is to say, $id_{\mathcal{A}}\,;f = f\,;id_{\mathcal{A}} = f$ for any morphism f).

Information about objects and morphisms in category theory is often displayed graphically in the form of *commuting diagrams*. Consider for example the following diagram:

$$\begin{tikzcd}\mathcal{A} \arrow[r, "f"] \arrow[d, "g"'] & \mathcal{B} \arrow[d, "h"] \\ \mathcal{C} \arrow[r, "i"'] & \mathcal{D}\end{tikzcd}$$

By itself, this diagram simply states the existence of four objects and the appropriate morphisms between them. However, if we say that the above diagram *commutes* then it also means that $f\,;h = g\,;i$. In other words, the two paths from $\mathcal{A}$ to $\mathcal{D}$ are equivalent. The above diagram is known as a *commuting square*. In general, a diagram commutes if all paths between any two objects in the diagram are equivalent. Commuting diagrams are a useful visual tool in category theory, and in our exposition all diagrams we show commute.

Although there are many kinds of categories, we will be focusing on structured sets. In such categories, objects are sets with some additional structure and the morphisms are structure-preserving functions. The **Set** category is the simplest example of such a category, since there is no structure imposed on the sets. A more structured example is the **Rel**(2) category of binary relations. An object in this category is a binary relation (represented, say, as a set of pairs), and a morphism is a relation-preserving function. In particular, the morphism $f : R_1 \to R_2$ is a function from the domain of R_1 to the domain of R_2 satisfying: $\forall x, y.\ x\, R_1\, y \implies f(x)\, R_2\, f(y)$. Informally, there are also categories of expressions, even recursive expressions, and the morphisms are substitutions of variables. As shown in more detail in Chapter 15, in the setting of compiler optimizations, we will use a category in which objects are E-PEGs and morphisms are substitutions.

14.2 Encoding Axioms in Category Theory

Many axioms can be expressed categorically as morphisms [1]. For example, transitivity ($\forall x, y, z.\ x\, R\, y \wedge y\, R\, z \Rightarrow x\, R\, z$) can be expressed as the following morphism in the **Rel**(2) category:

$$\begin{tikzcd} \boxed{\begin{array}{cc} x & y \\ y & z \end{array}} \arrow[r, "trans"] & \boxed{\begin{array}{cc} x & y \\ y & z \\ x & z \end{array}} \end{tikzcd}$$

where $trans$ is the function $(x \mapsto x, y \mapsto y, z \mapsto z)$ (we display a relation graphically as a listing of pairs – the left object above is the relation $\{(x,y),(y,z)\}$). In this case, the axiom is *identity-carried*, meaning the underlying function is the identity function, but that need not be the case in general.

Now consider an object $\mathcal{A}$ in the **Rel**(2) category. We will see how to state that this object (relation) is transitive. In particular, we say that $\mathcal{A}$ *satisfies* $trans$ if for every morphism $f : \{(x,y),(y,z)\} \to \mathcal{A}$ there exists a morphism $f' : \{(x,y),(y,z),(x,z)\} \to \mathcal{A}$

such that the following diagram commutes:

$$\begin{tikzcd}[ampersand replacement=\&]
\boxed{\begin{matrix} x & y \\ y & z \end{matrix}} \arrow[r, "trans"] \arrow[d, "f"'] \& \boxed{\begin{matrix} x & y \\ y & z \\ x & z \end{matrix}} \arrow[dl, "f'"] \\
\mathcal{A} \&
\end{tikzcd}$$

To see how this definition of $\mathcal{A}$ satisfying *trans* implies that $\mathcal{A}$ is a transitive relation, consider a morphism f from $\{(x,y),(y,z)\}$ to $\mathcal{A}$. This morphism is a function that selects three elements a,b,c in the domain of $\mathcal{A}$ such that $a\ \mathcal{A}\ b$ and $b\ \mathcal{A}\ c$. Since *trans* is the identity function, a morphism f' will exist if and only if $a\ \mathcal{A}\ c$ also holds. Since this has to hold for any f (i.e. any three elements a,b,c in the domain of $\mathcal{A}$ with $a\ \mathcal{A}\ b$ and $b\ \mathcal{A}\ c$), $\mathcal{A}$ will satisfy *trans* precisely when the relation defined by $\mathcal{A}$ is transitive.

Similarly, our E-PEG axioms can be encoded as identity-carried morphisms which that add an equality. The axiom $x*0=0$ is encoded as the identity-carried morphism from the E-PEG $x*y$, with y equivalent to 0, to the E-PEG $x*y$, with y equivalent to 0 *and* $x*y$ equivalent to 0. Thus, an E-PEG satisfies this axiom if for every $x*y$ node where y is equivalent to 0, the $x*y$ node is also equivalent to 0. More details on our E-PEG category can be found in Chapter 15.

14.3 Encoding Inference in Category Theory

Inference is the process of taking some already known information and applying axioms to learn additional information. In the E-PEG setting, inference consists of applying axioms to learn equality edges. To start with a simpler example, consider the relation $\{(a,b),(b,c),(c,d)\}$, and suppose we want to apply transitivity to (a,b) and (b,c) to learn (a,c). Applying transitivity first involves selecting the elements on which we want to apply the axiom. This can be modeled as a morphism from $\{(x,y),(y,z)\}$ to $\{(a,b),(b,c),(c,d)\}$, specifically $(x \mapsto a, y \mapsto b, z \mapsto c)$. This produces the following diagram:

$$\begin{tikzcd}
\boxed{\begin{array}{cc} x & y \\ y & z \end{array}} \arrow[r, "trans"] \arrow[d, "(x \mapsto a, y \mapsto b, z \mapsto c)"'] & \boxed{\begin{array}{cc} x & y \\ y & z \\ x & z \end{array}} \\
\boxed{\begin{array}{cc} a & b \\ b & c \\ c & d \end{array}} &
\end{tikzcd} \tag{14.1}$$

Adding (a,c) completes the diagram into a commuting square:

$$\begin{tikzcd}
\boxed{\begin{array}{cc} x & y \\ y & z \end{array}} \arrow[r, "trans"] \arrow[d, "(x \mapsto a, y \mapsto b, z \mapsto c)"'] & \boxed{\begin{array}{cc} x & y \\ y & z \\ x & z \end{array}} \arrow[d, "(x \mapsto a, y \mapsto b, z \mapsto c)"] \\
\boxed{\begin{array}{cc} a & b \\ b & c \\ c & d \end{array}} \arrow[r] & \boxed{\begin{array}{cc} a & b \\ b & c \\ c & d \\ a & c \end{array}}
\end{tikzcd} \tag{14.2}$$

The above commuting diagram therefore encodes that transivity was used to learn information, in particular (a,c), but unfortunately, it does not state that nothing *more* than transivity was learned. For example, the bottom-right object (relation) in the above commuting square could acually contain more entries, say (a,a), and the diagram would still commute. To address this issue, we use the concept of *pushouts* from category theory.

Definition 14.1 (Pushout)**.** *A commuting square* $[\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D}]$ *is said to be a* pushout square *if for any object* $\mathcal{E}$ *that makes* $[\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{E}]$ *a commuting square, there exists a unique morphism from* $\mathcal{D}$ *to* $\mathcal{E}$ *such that the following diagram commutes:*

$$\begin{tikzcd}
\mathcal{A} \arrow[r, "f"] \arrow[d, "g"'] & \mathcal{B} \arrow[d, "h"'] \arrow[ddr] & \\
\mathcal{C} \arrow[r, "i"] \arrow[drr] & \mathcal{D} \arrow[dr, dashed] & \\
 & & \mathcal{E}
\end{tikzcd}$$

Furthermore, given $\mathcal{A}$*,* $\mathcal{B}$*,* $\mathcal{C}$*,* f*, and* g *in the diagram above, the* pushout operation *constructs the appropriate* $\mathcal{D}$*,* i*, and* h *that makes* $[\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D}]$ *a pushout square. When the morphisms are obvious from context, we omit them from the list of arguments to the pushout operation, and in such cases we use the notation* $\mathcal{B} +_{\mathcal{A}} \mathcal{C}$ *for the result* $\mathcal{D}$ *of the pushout operation.*

Pushouts in general are useful for imposing additional structure. Intuitively, when constructing pushouts, $\mathcal{A}$ represents “glue” that will join together $\mathcal{B}$ and $\mathcal{C}$: f says where to apply the glue in $\mathcal{B}$; g says where to apply the glue in $\mathcal{C}$. The pushout produces $\mathcal{D}$ by gluing $\mathcal{B}$ and $\mathcal{C}$ together where indicated by f and g. For example, in a category of expressions, pushouts can be used to accomplish unification: if $\mathcal{A}$ is the expression consisting of a single variable x, and f and g map x to the root of expressions $\mathcal{B}$ and $\mathcal{C}$ respectively, then the pushout $\mathcal{D}$ is the unification of $\mathcal{B}$ and $\mathcal{C}$. If the expressions cannot be unified, then no pushout exists.

Going back to our example, to encode the application of the transitivity axiom, we require that the commuting square in Diagram (14.2) be a pushout square. The pushout square property applied to Diagram (14.2) ensures that, for any relation $\mathcal{E}$ such that $a\ \mathcal{E}\ c$, there will be a morphism from the bottom-right object in the diagram (call it $\mathcal{D}$) to $\mathcal{E}$, meaning that $\mathcal{E}$ contains as much or more information than $\mathcal{D}$, which in turn means that

$\mathcal{D}$ encodes the least relation that includes (a,c). This is exactly the result we want from applying transitivity on our example.

Furthermore, we can obtain the bottom-right corner of Diagram (14.2) by taking the pushout of Diagram (14.1). Thus, inference is the process of repeatedly identifying points where axioms can apply and pushing out to add the learned information. This produces a sequence of pushout squares whose bottom edges all chain together. For example, in the diagram below, *app₁* states where to apply axiom_1 in $\mathcal{E}_0$, and the pushout $\mathcal{E}_0 +_{\mathcal{A}_1} \mathcal{C}_1$ produces the result $\mathcal{E}_1$; in the second step, app_2 states where to apply axiom_2 in $\mathcal{E}_1$, and the pushout $\mathcal{E}_1 +_{\mathcal{A}_2} \mathcal{C}_2$ produces $\mathcal{E}_2$; this process can continue to produce an entire sequence $(\mathcal{E}_i)_{i\in\{0\dots n\}}$, where each $\mathcal{E}_i$ encodes more information than the previous one.

$$\begin{tikzcd}
\mathcal{A}_1 \arrow[r, "axiom_1"] \arrow[d, "app_1"] & \mathcal{C}_1 \arrow[dr] & \mathcal{A}_2 \arrow[r, "axiom_2"] \arrow[d, "app_2"] & \mathcal{C}_2 \arrow[dr] & & \\
\mathcal{E}_0 \arrow[rr] & & \mathcal{E}_1 \arrow[rr] & & \mathcal{E}_2 & \cdots
\end{tikzcd} \tag{14.3}$$

In the E-PEG setting, each $\mathcal{E}_i$ will be an E-PEG, and each axiom application will add an equality edge. The entire sequence above constitutes a proof in our formalism: it encodes both the axioms being applied (axiom_1, axiom_2, etc.), how they are applied (app_1, app_2, etc.), and the sequence of conclusions that are made ($\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{E}_2$, etc.). Traditional tree-style proofs (such as derivations) can be linearized into our categorical encoding of proofs (see Chapter 17 for more details on how this can be done).

14.4 Defining Generalization in Category Theory

Proof generalization involves identifying a property of the result of an inference process and determining the minimal information necessary for that proof to still infer that property. We represent a property as a morphism to the final result of the inference process. For example, in the **Rel**(2) category, a morphism from $\{(x,y)\}$ to the final result of inference would identify a related a and b whose inferred relationship we are interested in generalizing. For E-PEGs, a morphism from $\alpha \approx \beta$ to an E-PEG $\mathcal{E}$ identifies two equivalent nodes in $\mathcal{E}$, phrasing the property “these two nodes are equivalent”. Generalization applied to this property will produce a generalized E-PEG for which the proof will make those two nodes equivalent.

We start by looking at the *last* axiom application in the inference process, the one that produces the final result. In this case we have:

$$\begin{tikzcd}
\mathcal{A} \arrow[r, "axiom"] \arrow[d, "app"'] & \mathcal{C} \arrow[d, "app'"] & \mathcal{P} \arrow[dl, "prop"] \\
\mathcal{E} \arrow[r] & \mathcal{E}' &
\end{tikzcd}$$

$\mathcal{A} \xrightarrow{axiom} \mathcal{C}$ is the (last) axiom being applied. $\mathcal{A} \xrightarrow{app} \mathcal{E}$ is where the axiom is being applied. $\mathcal{E}'$ is the result of pushing out $axiom$ and app. $\mathcal{P} \xrightarrow{prop} \mathcal{E}'$ is the property of $\mathcal{E}'$ for which we want a generalized proof.

Next we need to identify which parts of $\mathcal{P}$ the last axiom application contributes to. This step is necessary because, in general, $\mathcal{P}$ may only be partially established by the last step of inference. For example, in the **Rel**(2) category, we may be interested in generalizing a proof whose conclusion is that the final relation includes (a,b) and (b,c). In this case, it is entirely possible that the last step of inference produced (a,b) whereas an earlier step produced (b,c).

To identify which parts of $\mathcal{P}$ the last axiom application contributed, we use the concept of *pullbacks* from category theory. Pullbacks are the dual concept to pushouts.

Definition 14.2 (Pullback)**.** *A commuting square* $[\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D}]$ *is said to be a* pullback square *if for any object* $\mathcal{E}$ *that makes* $[\mathcal{E},\mathcal{B},\mathcal{C},\mathcal{D}]$ *a commuting square, there exists a* *unique morphism from $\mathcal{E}$ to $\mathcal{A}$ such that the following diagram commutes:*

$$\begin{tikzcd}
\mathcal{E} \arrow[dr, dashed] \arrow[drr, bend left] \arrow[ddr, bend right] & & \\
& \mathcal{A} \arrow[r, "f"'] \arrow[d, "g"] & \mathcal{B} \arrow[d, "h"] \\
& \mathcal{C} \arrow[r, "i"'] & \mathcal{D}
\end{tikzcd}$$

Furthermore, given $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, i, and h in the diagram above, the pullback operation *constructs the appropriate $\mathcal{A}$, f, and g that makes $[\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D}]$ a pullback square. When the morphisms are obvious from context, we omit them from the list of arguments to the pullback operation, and in such cases we use the notation $\mathcal{B} \times_{\mathcal{D}} \mathcal{C}$ for the result $\mathcal{A}$ of the pullback operation.*

Whereas pushouts are good for imposing additional structure, pullbacks are good for identifying common structure. For example, in the **Set** category with injective functions, $\mathcal{B} \times_{\mathcal{D}} \mathcal{C}$ would intuitively be the intersection of the images of $\mathcal{B}$ and $\mathcal{C}$ in $\mathcal{D}$.

Returning to our diagram, we take the pullback $\mathcal{C} \times_{\mathcal{E}'} \mathcal{P}$:

$$\begin{tikzcd}
\mathcal{A} \arrow[r, "axiom"] \arrow[d, "app"'] & \mathcal{C} \arrow[d, "app'"] & \mathcal{O} \arrow[l] \arrow[d] \\
\mathcal{E} \arrow[r] & \mathcal{E}' & \mathcal{P} \arrow[l, "prop"]
\end{tikzcd}$$

$\mathcal{O}$ now identifies where the result of the application and the property overlap.

A generalization of $\mathcal{E}$ is an object $\mathcal{G}$ with a morphism $gen : \mathcal{G} \to \mathcal{E}$ (see Diagram (14.4) below). A generalization of app is a morphism $app_{\mathcal{G}} : \mathcal{A} \to \mathcal{G}$ such that $app_{\mathcal{G}} ; gen = app$. We apply $axiom$ to the generalized application $app_{\mathcal{G}}$ by taking the pushout to produce $\mathcal{G}'$. Lastly, we want our property to hold in $\mathcal{G}'$ for the same reason that it holds in $\mathcal{E}'$; that is, any information added by $axiom$ to make the property $prop$ hold in $\mathcal{E}'$ should also make the property hold in $\mathcal{G}'$. We enforce this by requiring an additional morphism

$\mathit{prop}_{\mathcal{G}} : \mathcal{P} \to \mathcal{G}'$. To summarize, a generalization of applying *axiom* via *app* to produce *prop* is an object $\mathcal{G}$ with morphisms *gen*, $\mathit{app}_{\mathcal{G}}$, and $\mathit{prop}_{\mathcal{G}}$ making the following diagram commute:

$$\begin{tikzcd}
 & & & & \mathcal{O} \arrow[dll] \arrow[dd] \\
\mathcal{A} \arrow[rr, "\mathit{axiom}"] \arrow[dr, "\mathit{app}_{\mathcal{G}}"] \arrow[dd, "\mathit{app}"'] & & \mathcal{C} \arrow[dr] \arrow[dd] & & \\
 & \mathcal{G} \arrow[rr, crossing over] \arrow[dl, "\mathit{gen}"] & & \mathcal{G}' \arrow[dl, dashed] & \mathcal{P} \arrow[l, "\mathit{prop}_{\mathcal{G}}"'] \arrow[dll, "\mathit{prop}"] \\
\mathcal{E} \arrow[rr] & & \mathcal{E}' & &
\end{tikzcd} \tag{14.4}$$

Recall that $\mathcal{G}'$ in the above diagram is the pushout of *axiom* and $\mathit{app}_{\mathcal{G}}$. The dashed line from $\mathcal{G}'$ to $\mathcal{E}'$ is the unique morphism induced by that pushout (note that there is a morphism from $\mathcal{G}$ to $\mathcal{E}'$ passing through $\mathcal{E}$).

The above diagram defines a generalization for the *last* step of the inference process. A generalization for an entire sequence of steps – such as Diagram (14.3) – is an initial $\mathcal{G}_0$ and a morphism *gen* from $\mathcal{G}_0$ to $\mathcal{E}_0$ with a sequence of generalized axiom applications such that the property holds in the final $\mathcal{G}_n$.

14.5 Constructing Generalizations using Categories

Above we have defined what a generalization is, but not how to construct one. Furthermore, our goal is not just to construct some generalization; after all $\mathcal{E}$ is a trivial generalization of itself. We would like to construct the most general generalization, meaning that not only does it generalize $\mathcal{E}$, but it also generalizes any other generalization of $\mathcal{E}$.

In order to express our generalization algorithm, we introduce a new category-theoretic operation we call a *pushout completion*.

Definition 14.3 (Pushout completion)**.** *Given a diagram*

$$\begin{tikzcd}
\mathcal{A} \arrow[r, "f"] & \mathcal{B} \arrow[d, "g"] \\
& \mathcal{D}
\end{tikzcd}$$

the pushout completion of $[\mathcal{A},\mathcal{B},\mathcal{D},f,g]$ *is a pushout square* $[\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D}]$ *with the property that, for any other pushout square* $[\mathcal{A},\mathcal{B},\mathcal{E},\mathcal{F}]$ *in which the morphism from* $\mathcal{B}$ *to* $\mathcal{F}$ *passes through* $\mathcal{D}$*, there is a unique morphism from* $\mathcal{C}$ *to* $\mathcal{E}$ *(shown below with a dashed arrow) such that the following diagram commutes:*

$$\begin{tikzcd}
\mathcal{A} \arrow[rr, "f"] \arrow[dr] \arrow[dd] & & \mathcal{B} \arrow[d, "g"] \\
& \mathcal{C} \arrow[r] \arrow[dl, dashed] & \mathcal{D} \arrow[d] \\
\mathcal{E} \arrow[rr] & & \mathcal{F}
\end{tikzcd}$$

When the morphisms are obvious from context, we omit them from the list of arguments to the pushout completion, and in such cases we use the notation $\mathcal{D} -_{\mathcal{A}} \mathcal{B}$ *for the result* $\mathcal{C}$ *of the pushout completion (the minus notation is used because in the above diagram we have* $\mathcal{D} = \mathcal{B} +_{A} \mathcal{C}$*).*

The intuition is that $\mathcal{C}$ captures the structure of $\mathcal{D}$ minus the structure of $\mathcal{B}$ (reflected into $\mathcal{D}$ through g) while keeping the structure of $\mathcal{A}$ (reflected into $\mathcal{D}$ through $f\,;g$). For example, in the **Rel**(2) category, intuitively we would have: $\mathcal{C} = (\mathcal{D} \setminus \mathcal{B}) \cup \mathcal{A}$ (where $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ are sets of tuples that represent relations).

Our key requirement for constructing generalizations is that, for every axiom $\mathcal{A} \xrightarrow{axiom} \mathcal{C}$, there is a pushout completion for any morphism f from $\mathcal{C}$ to any object. There is encouraging evidence that axioms with pushout completions are quite common. In particular, all of our PEG axioms satisfy this condition. More generally, all identity-carried morphisms in **Rel**(2) or the category of expressions satisfy this condition. Furthermore, in Section 14.6 we show how to loosen this condition in a way that allows *all* morphisms in **Set** and **Rel**(2) to qualify as generalizable axioms.

Now that we have all the necessary concepts, the diagram below and the subsequent description explain the steps that our algorithm takes to construct the best generalization:

$$\begin{tikzcd}
& & & & \mathcal{O} \arrow[lld] \arrow[d] \\
\mathcal{A} \arrow[rr, "\mathit{axiom}"] \arrow[dd, "\mathit{app}"'] \arrow[rd] & & \mathcal{C} \arrow[rd] \arrow[dd] & & \\
& \mathcal{P}' \arrow[rr] \arrow[ld, dashed] & & \bar{\mathcal{P}} \arrow[ld, dashed] & \mathcal{P} \arrow[l] \arrow[lld, "\mathit{prop}"] \\
\mathcal{E} \arrow[rr] & & \mathcal{E}' & &
\end{tikzcd} \qquad (14.5)$$

1. $\mathcal{O}$ is constructed by taking the pullback $\mathcal{C} \times_{\mathcal{E}'} \mathcal{P}$.

2. $\bar{\mathcal{P}}$ is constructed by taking the pushout $\mathcal{C} +_{\mathcal{O}} \mathcal{P}$. The pushout $\bar{\mathcal{P}}$ intuitively represents the unification of property $\mathcal{P}$ with the assumptions and conclusions of axiom (which are represented by $\mathcal{C}$). The dashed morphism from $\bar{\mathcal{P}}$ to $\mathcal{E}'$ is induced by the pushout property of $\bar{\mathcal{P}}$; it identifies how this unified structure fits in $\mathcal{E}'$.

3. $\mathcal{P}'$ is constructed by taking the pushout completion $\bar{\mathcal{P}} -_{\mathcal{A}} \mathcal{C}$. The pushout completion $\mathcal{P}'$ intuitively represents the information in $\bar{\mathcal{P}}$ but with the inferred conclusion of axiom removed. The dashed morphism from $\mathcal{P}'$ to $\mathcal{E}$ is induced by the pushout property of $\bar{\mathcal{P}}$; it identifies the minimal information necessary in $\mathcal{E}$ so that applying axiom produces property $\mathcal{P}$.

Let us return to the larger context of a chain of axioms rather than applying just one axiom. In Diagram (14.5) above, $\mathcal{E}$ would be the result of an earlier axiom application. $\mathcal{P}'$ then identifies the property of $\mathcal{E}$ that needs to be generalized in that axiom application. This process of generalization can be repeated backwards through the chain of axioms until we arrive at the original $\mathcal{E}_0$ that the inference process started with. The generalized property of $\mathcal{E}_0$ at that point is then the best generalization of $\mathcal{E}_0$ such that the proof infers the property $\mathcal{P}$ we started with in Diagram (14.5). The fact that this solution is the most general falls immediately from the pushout and pushout-completion properties of the construction. In particular, suppose that in Diagram (14.5) there is another generalization $\mathcal{G}$, which essentially means that we merge diagrams (14.4) and (14.5) together. We need to show that there exists a morphism from $\mathcal{P}'$ to $\mathcal{G}$. First, the pushout property on $\bar{\mathcal{P}}$ induces a morphism from $\bar{\mathcal{P}}$ to $\mathcal{G}'$. Second, the pushout-completion property on $\mathcal{P}'$ induces the desired morphism from $\mathcal{P}'$ to $\mathcal{G}$.

The above process provides a categorical algorithm for generalizing proofs. The algorithm is parameterized by the choice of category used to represent the inference process, along with the details of implementing pushouts, pullbacks, and pushout completions on this category.

14.6 Subpushout Completions

The requirement that all axioms always have pushout completions is more restrictive than necessary. Although all the axioms in our implementation satisfy this requirement, other applications of proof generalization may need more freedom in their choice of axioms. Here we formalize the actual requirement of all axioms needed to apply our proof-generalization technique.

In our proof-generalization process, we only apply proof generalization in the following situation:

$$\begin{tikzcd}
 & & & \mathcal{O} \arrow[dll] \arrow[dd] \\
\mathcal{A} \arrow[r, "\mathit{axiom}"] \arrow[dd, "\mathit{app}"'] & \mathcal{C} \arrow[dr, "\overline{\mathit{app}}"] \arrow[dd, "\mathit{app}'"'] & & \\
 & & \bar{\mathcal{P}} \arrow[dl, dashed, "\overline{\mathit{prop}}"'] & \mathcal{P} \arrow[l] \arrow[dll, "\mathit{prop}"] \\
\mathcal{E} \arrow[r] & \mathcal{E}' & &
\end{tikzcd}$$

where $\mathcal{O}$ is the pullback of app' and prop, and $\bar{\mathcal{P}}$ is the pushout of this pullback.

Whenever this situation arises, there needs to be a *subpushout completion* of the pushout square $[\mathcal{A},\mathcal{C},\mathcal{E},\mathcal{E}']$ and $\bar{\mathcal{P}}$, which we define in steps. A subpushout of the pushout square $[\mathcal{A},\mathcal{C},\mathcal{E},\mathcal{E}']$ is a pushout square $[\mathcal{A},\mathcal{C},\mathcal{G},\mathcal{G}']$ with a morphism $gen_{\mathcal{G}} : \mathcal{G} \to \mathcal{E}$ making the following diagram commute:

$$\begin{tikzcd}
\mathcal{A} \arrow[rr, "axiom"] \arrow[dd, "app"'] \arrow[dr, "app_{\mathcal{G}}"] & & \mathcal{C} \arrow[dd] \arrow[dr, "app'_{\mathcal{G}}"] & \\
& \mathcal{G} \arrow[rr] \arrow[dl, "gen_{\mathcal{G}}"] & & \mathcal{G}' \arrow[dl, dashed, "gen'_{\mathcal{G}}"] \\
\mathcal{E} \arrow[rr] & & \mathcal{E}' &
\end{tikzcd}$$

A subpushout through $\bar{\mathcal{P}}$ also has a morphism $ext_{\mathcal{G}} : \bar{\mathcal{P}} \to \mathcal{G}'$ with $app'_{\mathcal{G}} = \overline{app}\,; ext_{\mathcal{G}}$ and $ext_{\mathcal{G}}\,; gen'_{\mathcal{G}} = \overline{prop}$.

A subpushout completion of $[\mathcal{A},\mathcal{C},\mathcal{E},\mathcal{E}']$ and $\overline{app}$ is a *universal* subpushout $[\mathcal{A},\mathcal{C},\mathcal{P}',\hat{\mathcal{P}}]$ of $[\mathcal{A},\mathcal{C},\mathcal{E},\mathcal{E}']$ through $\bar{\mathcal{P}}$, meaning that for any other subpushout $[\mathcal{A},\mathcal{C},\mathcal{G},\mathcal{G}']$ of $[\mathcal{A},\mathcal{C},\mathcal{E},\mathcal{E}']$ extending $\bar{\mathcal{P}}$ there is a unique morphism from $\mathcal{P}'$ to $\mathcal{G}$ making the following diagram commute:

$$\begin{tikzcd}
\mathcal{A} \arrow[rr, "axiom"] \arrow[ddd, "app"'] \arrow[dr, "app_{\mathcal{P}}"] \arrow[ddr, "app_{\mathcal{G}}"'] & & \mathcal{C} \arrow[rr, "\overline{app}"] \arrow[ddd] \arrow[dr] \arrow[ddr] & & \bar{\mathcal{P}} \arrow[dl, "ext_{\mathcal{P}}"'] \arrow[ddl, "ext_{\mathcal{G}}"] \\
& \mathcal{P}' \arrow[rr] \arrow[d, dashed] \arrow[ddl, "gen_{\mathcal{P}}"'] & & \hat{\mathcal{P}} \arrow[d, dashed] \arrow[ddl, dashed] & \\
& \mathcal{G} \arrow[rr] \arrow[dl, "gen_{\mathcal{G}}"] & & \mathcal{G}' \arrow[dl, dashed] & \\
\mathcal{E} \arrow[rr] & & \mathcal{E}' & &
\end{tikzcd}$$

This is admittedly complicated, but this more precise requirement actually makes many more axioms valid for generalization. For example, in (classical) **Set** only the surjective functions (and the unique function from the empty set to the singleton set) satisfy the less precise requirement. However, all morphisms in **Set** satisfy the more precise requirement. Because of this, all morphisms in **Rel**(2) also satisfy the more precise requirement. In fact, in both categories, the extension morphism $\mathit{ext}_{\mathcal{P}}$ is always an identity morphism (and $\hat{\mathcal{P}}$ is always equal to $\bar{\mathcal{P}}$).

14.7 Proof of Generality

Before constructing the most general proof, we have to define what a generalized proof is. Suppose we have a concrete proof with objects $(\mathcal{E}_i)_{i\in\{0\ldots n\}}$, axioms $(\mathit{axiom}_i : \mathcal{A}_i \to \mathcal{C}_i)_{i\in\{1\ldots n\}}$, and applications $(\mathit{app}_i : \mathcal{A}_i \to \mathcal{E}_{i-1})_{i\in\{1\ldots n\}}$ such that $\mathcal{E}_0$ is the concrete assumption and $\mathcal{E}_n$ is the concrete conclusion. $\mathit{app}'_i : \mathcal{C}_i \to \mathcal{E}_i$, for i in $\{1\ldots n\}$, is defined as the appropriate morphism in the pushout diagram $[\mathcal{A}_i, \mathcal{C}_i, \mathcal{E}_{i-1}, \mathcal{E}_i]$. Also suppose we have a desired property of our concrete conclusion, $\mathit{prop} : \mathcal{P} \to \mathcal{E}_n$, that we want to have hold in our generalized proof. Then a generalized proof is a proof using the same axioms but different objects $(\mathcal{X}_i)_{i\in\{0\ldots n\}}$ and applications $(\mathit{app}^{\mathcal{X}}_i : \mathcal{A}_i \to \mathcal{X}_{i-1})_{i\in\{1\ldots n\}}$ along with morphisms $(\mathit{gen}^{\mathcal{X}}_i : \mathcal{X}_i \to \mathcal{E}_i)_{i\in\{0\ldots n\}}$ and $\mathit{prop}^{\mathcal{X}} : \mathcal{P} \to \mathcal{X}_n$ evidencing generalizations satisfying a few properties. First, $\mathit{app}^{\mathcal{X}}_i ; \mathit{gen}^{\mathcal{X}}_i$ equals app_i, for i in $\{1\ldots n\}$, so that applications in proof $\mathcal{X}$ are generalizations of the applications in proof $\mathcal{E}$. Second, $\mathit{prop}^{\mathcal{X}} ; \mathit{gen}^{\mathcal{X}}_n$ equals prop, so that the property of the generalized conclusion $\mathcal{X}_n$ is a generalization of the desired property of the concrete conclusion $\mathcal{E}_n$. Lastly, $[\mathcal{O}, \mathcal{P}, \mathcal{C}_n, \mathcal{X}_n]$ must form a commutative square (where $\mathcal{O}$ is the pullback of prop and app_n), so that the desired property is still a result of the final axiom application. Now we procede to construct a generalized proof and prove that it is the most general generalization.

First, we use the process for generalizing axiom applications in order to construct the significant property $(\mathit{prop}_i : \mathcal{P}_i \to \mathcal{E}_i)_{i\in\{0\ldots n\}}$ at each stage. $\mathcal{P}\ n$ is defined as $\mathcal{P}$, and $\mathit{prop}_n : \mathcal{P}_n \to \mathcal{E}_n$ is defined as $\mathit{prop} : \mathcal{P} \to \mathcal{E}_n$. $\mathcal{O}_i$, for i in $\{1\ldots n\}$, is the pullback of app'_i and prop_i. $\bar{\mathcal{P}}_i$, for i in $\{1\ldots n\}$, is the pushout of $\mathcal{O}_i$'s pullback diagram, with $\overline{\mathit{app}}_i : \mathcal{C}_i \to \bar{\mathcal{P}}_i$ as the appropriate morphism in the pushout square. $\mathcal{P}_{i-1}$, $\hat{\mathcal{P}}_i$, $\mathit{ext}^{\mathcal{P}}_i : \bar{\mathcal{P}}_i \to \hat{\mathcal{P}}_i$, $\mathit{app}^{\mathcal{P}}_i : \mathcal{A}_i \to \mathcal{P}_{i-1}$, and $\mathit{prop}_{i-1} : \mathcal{P}_{i-1} \to \mathcal{E}_{i-1}$, for i in $\{1\ldots n\}$, are all defined as the appropriate objects and morphisms resulting from the subpushout completion of $[\mathcal{A}_i, \mathcal{C}_i, \mathcal{E}_{i-1}, \mathcal{E}_i]$ and $\overline{app}_i$.

From this we construct a generalized proof $(\mathcal{G}_i)_{i\in\{0\ldots n\}}$ with generalizations $(gen^{\mathcal{G}}_i : \mathcal{G}_i \to \mathcal{E}_i)_{i\in\{0\ldots n\}}$ and applications $(app^{\mathcal{G}}_i : \mathcal{A}_i \to \mathcal{G}_{i-1})_{i\in\{1\ldots n\}}$. We will also construct a morphism $prop^{\mathcal{G}} : \mathcal{P} \to \mathcal{G}_n$ such that $prop^{\mathcal{G}} ; gen^{\mathcal{G}}_n$ equals $prop$ and $[\mathcal{O}, \mathcal{P}, \mathcal{C}_n, \mathcal{G}_n]$ is a commuting square (where $\mathcal{O}$ is the pullback of $prop$ and app_n), demonstrating that the desired property is concluded by this generalized proof using the final axiom application. $\mathcal{G}_0$ is defined as $\mathcal{P}_0$ with $gen^{\mathcal{G}}_0 : \mathcal{G}_0 \to \mathcal{E}_0$ defined as $prop_0$, thus the generalized property we constructed is the starting point of our generalized proof. As we construct the rest of our proof, we will also construct a sequence of morphisms $(prop^{\mathcal{G}}_i : \mathcal{P}_i \to \mathcal{G}_i)_{i\in\{0\ldots n\}}$, with $prop^{\mathcal{G}}_i ; gen^{\mathcal{G}}_i = prop_i$ always holding, showing that the generalized property for each stage holds in our generalized object at the stage. $prop^{\mathcal{G}}_0 : \mathcal{P}_0 \to \mathcal{G}_0$ is defined simply as the identity morphism, automatically satisfying the above property. $app^{\mathcal{G}}_i : \mathcal{A}_i \to \mathcal{G}_i$, for i in $\{1\ldots n\}$, is simply defined as $app^{\mathcal{P}}_i ; prop^{\mathcal{G}}_i$. $\mathcal{G}_i$, for i in $\{1\ldots n\}$, is then defined as the pushout of $axiom_i$ and $app^{\mathcal{G}}_i$. $prop^{\mathcal{G}}_i : \mathcal{P}_i \to \mathcal{G}_i$ and $gen^{\mathcal{G}}_i : \mathcal{G}_i \to \mathcal{E}_i$, for i in $\{1\ldots n\}$, are defined using the morphisms induced by pushouts in the following commutative diagram:

$$\begin{tikzcd}
 & & & & \mathcal{O}_i \arrow[dll, "\overline{app}_i"] \arrow[dr] & \\
\mathcal{A}_i \arrow[rr, "axiom_i"] \arrow[ddr, "app^{\mathcal{P}}_i"] \arrow[dddr, "app^{\mathcal{G}}_i"] \arrow[dddd, "app_i"'] & & \mathcal{C}_i \arrow[dr] \arrow[dddrr] \arrow[dddd] & & & \mathcal{P}_i \arrow[dll] \arrow[dddl, "prop^{\mathcal{G}}_i"] \arrow[ddddlll, bend left, "prop_i"] \\
 & & & \bar{\mathcal{P}}_i \arrow[dr, "ext^{\mathcal{P}}_i"] & & \\
 & \mathcal{P}_{i-1} \arrow[rrr] \arrow[d, "prop^{\mathcal{G}}_{i-1}"] \arrow[ddl, "prop_{i-1}"'] & & & \hat{\mathcal{P}}_i \arrow[d, dashed] \arrow[ddll, "prop'_{i-1}"] & \\
 & \mathcal{G}_{i-1} \arrow[rrr] \arrow[dl, "gen^{\mathcal{G}}_{i-1}"] & & & \mathcal{G}_i \arrow[dll, dashed, "gen^{\mathcal{G}}_i"] & \\
\mathcal{E}_{i-1} \arrow[rr] & & \mathcal{E}_i & & &
\end{tikzcd}$$

Although most components of this diagram commute by either assumption or construction, the two paths from $\hat{\mathcal{P}}_i$ to $\mathcal{E}_i$ commute due to the uniqueness property of pushout squares (specifically of the pushout square $[\mathcal{A}_i, \mathcal{C}_i, \mathcal{P}_{i-1}, \mathcal{P}_i]$). Because of this, $\mathit{prop}_i^{\mathcal{G}};\mathit{gen}_i^{\mathcal{G}}$ is equal to prop_i, maintaining our inductive property. In particular, $\mathit{prop}_n^{\mathcal{G}};\mathit{gen}_n^{\mathcal{G}}$ equals prop_n and $[\mathcal{O}_n, \mathcal{C}_n, \mathcal{P}_n, \mathcal{G}_n]$ is a commuting square, so by defining $\mathit{prop}^{\mathcal{G}}$ as $\mathit{prop}_n^{\mathcal{G}}$ we demonstrate that the desired property holds appropriately in the result of our proof (since prop_n is defined as prop and $\mathcal{O}_n$ is $\mathcal{O}$). Thus, we have constructed a valid generalized proof.

Lastly we need to prove that this is the most general proof of the given concrete proof. So suppose there is another generalized proof $(\mathcal{X}_i)_{i\in\{0\ldots n\}}$ with generalizations $(\mathit{gen}_i^{\mathcal{X}} : \mathcal{X}_i \to \mathcal{E}_i)_{i\in\{0\ldots n\}}$ and applications $(\mathit{app}_i^{\mathcal{X}} : \mathcal{A}_i \to \mathcal{X}_{i-1})_{i\in\{1\ldots n\}}$. This generalized proof also has a morphism $\mathit{prop}^{\mathcal{X}} : \mathcal{P} \to \mathcal{X}_n$ such that $\mathit{prop}^{\mathcal{X}};\mathit{gen}_n^{\mathcal{X}}$ equals prop and $[\mathcal{O}, \mathcal{P}, \mathcal{C}_n, \mathcal{X}_n]$ is a commuting square, demonstrating that the desired property is concluded by this generalized proof using the final axiom application. First we demonstrate that the generalized property prop_i we constructed at each stage holds in $\mathcal{X}_i$ at each stage by constructing a suitable morphism $\mathit{prop}_i^{\mathcal{X}} : \mathcal{P}_i \to \mathcal{X}_i$ such that $\mathit{prop}_i^{\mathcal{X}};\mathit{gen}_i^{\mathcal{X}}$ equals prop_i and $[\mathcal{O}_i, \mathcal{P}_i, \mathcal{C}_i, \mathcal{X}_i]$ is a commuting square. We define $\mathit{prop}_n^{\mathcal{X}}$ as $\mathit{prop}^{\mathcal{X}}$ which satisfies the relevant properties by assumption, since prop_n is defined as prop and $\mathcal{O}_n$ is $\mathcal{O}$. We define $\mathit{prop}_{i-1}^{\mathcal{X}}$, for i in $\{1\ldots n\}$, using the morphisms induced in the following commutative diagram:

$$\begin{tikzcd}
 & & & & & \mathcal{O}_i \arrow[dlll] \arrow[dr] & \\
 & & & & & & \mathcal{P}_i \arrow[ddlll] \arrow[ddddll, "\mathit{prop}_i^{\mathcal{X}}"] \arrow[dddddllll, bend left, "\mathit{prop}_i"] \\
\mathcal{A}_i \arrow[rr, "\mathit{axiom}_i"] \arrow[ddr, "\mathit{app}_i^{\mathcal{P}}"] \arrow[dddr, "\mathit{app}_i^{\mathcal{X}}"] \arrow[dddd, "\mathit{app}_i"'] & & \mathcal{C}_i \arrow[dr, "\overline{\mathit{app}}_i"] \arrow[dddrr] \arrow[dddd] & & & & \\
 & & & \bar{\mathcal{P}}_i \arrow[dr, "\mathit{ext}_i^{\mathcal{P}}"] \arrow[ddr, dashed] & & & \\
 & \mathcal{P}_{i-1} \arrow[rrr] \arrow[d, dashed, "\mathit{prop}_{i-1}^{\mathcal{X}}"] \arrow[ddl, "\mathit{prop}_{i-1}"] & & & \hat{\mathcal{P}}_i \arrow[d, dashed] \arrow[ddll, "\mathit{prop}'_{i-1}"'] & & \\
 & \mathcal{X}_{i-1} \arrow[rrr] \arrow[dl, "\mathit{gen}_{i-1}^{\mathcal{X}}"] & & & \mathcal{X}_i \arrow[dll, "\mathit{gen}_i^{\mathcal{X}}"] & & \\
\mathcal{E}_{i-1} \arrow[rr] & & \mathcal{E}_i & & & &
\end{tikzcd} \tag{14.6}$$

The morphism from $\bar{\mathcal{P}}_i$ to $\mathcal{X}_i$ is induced by the pushout property of $\bar{\mathcal{P}}_i$ since $[\mathcal{O}_i, \mathcal{P}_i, \mathcal{C}_i, \mathcal{X}_i]$

is a commuting square by inductive assumption; call this morphism $\mathit{ext}_i^{\mathcal{X}}$. The morphism from $\mathcal{P}_{i-1}$ to $\mathcal{X}_{i-1}$ and the morphism from $\hat{\mathcal{P}}_i$ to $\mathcal{X}_i$ are induced by the subpushout-completion property of $[\mathcal{A}_i, \mathcal{C}_i, \mathcal{P}_{i-1}, \hat{\mathcal{P}}_i]$ since $\mathit{gen}_i^{\mathcal{X}}$ demonstrates that $[\mathcal{A}_i, \mathcal{C}_i, \mathcal{X}_{i-1}, \mathcal{X}_i]$ is a subpushout of $[\mathcal{A}_i, \mathcal{C}_i, \mathcal{E}_{i-1}, \mathcal{E}_i]$ and $\mathit{ext}_i^{\mathcal{X}}$ demonstrates that this subpushout extends $\bar{\mathcal{P}}_i$. Lastly, we construct a sequence of morphisms $(\mathit{gen}_i : \mathcal{G}_i \to \mathcal{X}_i)_{i \in \{0 \ldots n\}}$ demonstrating that $(\mathcal{G}_i)_{i \in \{0 \ldots n\}}$ is a generalized proof of $(\mathcal{X}_i)_{i \in \{0 \ldots n\}}$. For each i in $\{0 \ldots n\}$, we will maintain the property that $\mathit{prop}_i^{\mathcal{G}} ; \mathit{gen}_i$ equals $\mathit{prop}_i^{\mathcal{X}}$, demonstrating that $\mathcal{G}$ generalizes the significant property in $\mathcal{X}$ in each stage. gen_0 is defined as $\mathit{prop}_0^{\mathcal{X}}$ (since $\mathcal{G}_0$ is defined as $\mathcal{P}_0$), satisfying the above property since $\mathit{prop}_0^{\mathcal{G}}$ is simply the identity morphism. gen_i, for i in $\{1 \ldots n\}$, is induced by the pushout property of $[\mathcal{P}_{i-1}, \hat{\mathcal{P}}_i, \mathcal{G}_{i-1}, \mathcal{G}_i]$ using gen_{i-1}, satisfying the above property by construction and the fact that $\mathit{prop}_i^{\mathcal{G}}$ is defined in terms of the unique morphism from $\hat{\mathcal{P}}_i$ to $\mathcal{G}_i$ ($[\mathcal{P}_{i-1}, \hat{\mathcal{P}}_i, \mathcal{G}_{i-1}, \mathcal{G}_i]$ is a pushout square using the pushout lemma, not repeated here as it is standard, since $[\mathcal{A}_i, \mathcal{C}_i, \mathcal{P}_{i-1}, \hat{\mathcal{P}}_i]$ and $[\mathcal{A}_i, \mathcal{C}_i, \mathcal{G}_{i-1}, \mathcal{G}_i]$ are both pushout squares). From this, we can also easily deduce that $\mathit{prop}^{\mathcal{G}} ; \mathit{gen}_n$ equals $\mathit{prop}^{\mathcal{X}}$ and, for i in $\{1 \ldots n\}$, $\mathit{app}_i^{\mathcal{G}} ; \mathit{gen}_{i-1}$ equals $\mathit{app}_i^{\mathcal{X}}$, demonstrating that $\mathcal{G}$ is a generalized proof of $\mathcal{X}$. Also, for i in $\{0 \ldots n\}$, $\mathit{gen}_i^{\mathcal{G}}$ equals $\mathit{gen}_i ; \mathit{gen}_i^{\mathcal{X}}$ due to the uniqueness property of pushouts, demonstrating that $\mathcal{G}$ is a generalization of the generalization $\mathcal{X}$.

The proof we have presented is actually flawed. We have made a major over-simplification for sake of explanation. The flaw is that, in Diagram (14.6), it is not necessarily the case that the square $[\mathcal{O}_i, \mathcal{P}_i, \mathcal{C}_i, \mathcal{X}_i]$ commutes. The reason is that we have not imposed enough constraints on the generalized proof $\mathcal{X}$. In particular, we have not required a generalized proof to make the conclusions of earlier axiom applications to contribute to the assumptions of later axioms in the same way they do in the original proof. In fact, as it stands often it can be the case that $\mathcal{X}_0$ is the coproduct (e.g. disjoint union) of all the axiom assumptions, completely disconnected, and $\mathcal{X}_n$ is the coproduct of all the axiom conclusions, again completely disconnected, with the conclusion of no axiom application contributing to the assumption of another. The reason is that a proof is more than just a sequence of pushout squares. A proof also details how the conclusion of one axiom application contributed to the conclusion of another. Our use of pullbacks in the generalization process was an approximation of this. Should one include objects and morphisms mapping into the concrete proof along with required equivalences indicating the glue of the proof across axioms, then one can use pushouts along those morphisms instead to get a proof generalization process very similar to the one we presented. Then one requires that the required equivalences factor through a generalized proof to indicate that it glues together appropriately, in which case the square analogous to $[\mathcal{O}_i, \mathcal{P}_i, \mathcal{C}_i, \mathcal{X}_i]$ in Diagram (14.6) will commute and the proof we presented will be valid. Nonetheless, the algorithm and arguments we gave convey the critical concepts behind our proof-generalization technique.

Acknowledgements

This chapter contains material taken from “Generating Compiler Optimization from Proofs”, by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission

and/or a fee.

Chapter 15

E-PEG Instantiation

We now show how to instantiate the categorical algorithm from Chapter 14 with E-PEGs. The category of E-PEGs can be formalized in a straghtforward way using well established categories such as partial algebras $\mathbf{PAlg}(\Omega)$ and relations $\mathbf{Rel}(2)$ [1]. The full formal description, however, is lengthy to expose, so we provide here a semi-formal description.

An object in the E-PEG category is simply an E-PEG. These E-PEGs can have free variables, and a morphism f from one E-PEG $\mathcal{A}$ to another E-PEG $\mathcal{B}$ is a map from the free variables of $\mathcal{A}$ to nodes of $\mathcal{B}$ such that, when the substitution f is applied to $\mathcal{A}$, the resulting E-PEG is a sub-graph of $\mathcal{B}$. The substitution is also required to map expressions equivalent in $\mathcal{A}$ to expressions equivalent in $\mathcal{B}$.

The three operations that need to be defined on the E-PEG category intuitively work as follows:

- The pullback $\mathcal{A} \times_{\mathcal{C}} \mathcal{B}$ treats $\mathcal{A}$ and $\mathcal{B}$ as sub-E-PEGs of $\mathcal{C}$ and takes their intersection.
- The pushout $\mathcal{A} +_{\mathcal{C}} \mathcal{B}$ treats $\mathcal{C}$ as a sub-E-PEG of both $\mathcal{A}$ and $\mathcal{B}$ and unifies $\mathcal{A}$ and $\mathcal{B}$ along the common substructure $\mathcal{C}$.
- The pushout completion $\mathcal{C} -_{\mathcal{A}} \mathcal{B}$ removes from $\mathcal{C}$ the equalities in $\mathcal{B}$ that are not

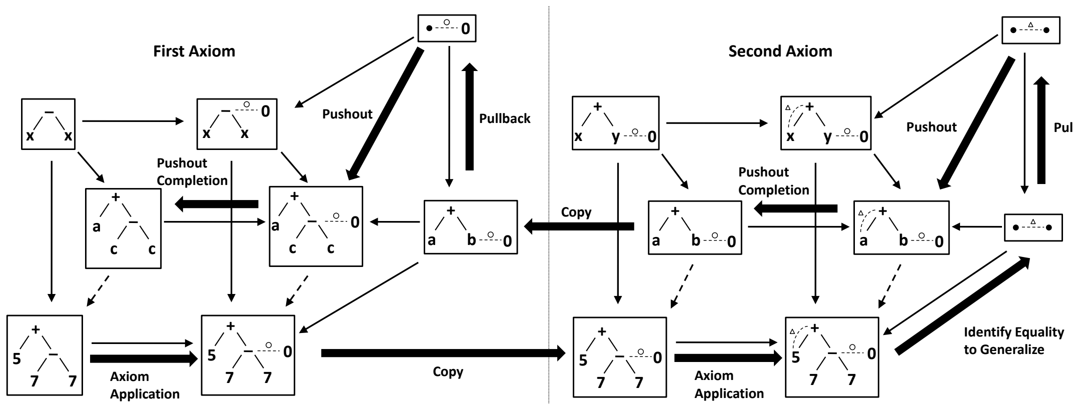


Figure 15.1. Example of generalization using E-PEGs

present in $\mathcal{A}$.

We now revisit the example from Figure 13.3 and this time explain it using our category-theoretic algorithm. Figure 15.1 shows the generalization process for this example using E-PEGs. The objects (boxes) in this figure are E-PEGs, and the thin arrows between them are morphisms. The example has two axiom applications, and so Figure 15.1 consists essentially of two side-by-side instantiations of Diagram (14.5). The thick arrows identify the steps taken by inference and generalization. The equality edges in each E-PEG are labeled with either a $\bigcirc$ or a $\triangle$ to show how these equality edges map from one E-PEG to another through the morphisms.

In this example the inference process uses two axioms to infer that $5+(7-7)$ is equal to 5. First, it applies the axiom $x-x=0$ to learn that $7-7=0$, and then $x+0=x$ to learn that $5+(7-7)=5$. We use the "Copy" arrows just for layout reasons – these copy operations are not actually performed by our algorithm.

Once the inference process is complete, we identify the equality we are interested in generalizing by creating an E-PEG containing the equality $\alpha \approx \beta$ (with α and β fresh) and a morphism from this E-PEG to the final result of inference which maps α to 5 and β to the $+$ node. Thus we are singling out the $\triangle$ equality in the final result of inference to generalize.

Having singled out which equality we want to focus on, we generalize the second axiom application using our three-step approach from Chapter 14: a pullback, a pushout, and then a pushout completion. The pullback identifies how the axiom is contributing to the equalities we are interested in – in this case it contributes through the $\triangle$ equality. The pushout then unifies the equality we are interested in generalizing with a freshly instantiated version of the axiom's conclusion (with a and b being fresh). Finally, the pushout completion essentially runs the axiom in reverse, removing the axiom's conclusion. In particular, it takes the substitution from the unification and applies it to the premise of the axiom to produce the E-PEG $[a+b\cdots 0]$.

This E-PEG, which is the result of generalizing the second axiom application, is then used as a starting point for generalizing the first axiom application, again using our three steps. The pullback identifies that the first axiom establishes the $\bigcirc$ equality edge. The pushout unifies $[a+b\cdots 0]$ with a freshly instantiated version of the axiom's conclusion (with c being fresh). Note that the $\bigcirc$ equality edge in $[a+b\cdots 0]$ must unify with the corresponding equality edge in the axiom's conclusion, and so b gets unified with the minus node. Finally, the pushout completion runs the first axiom in reverse, essentially removing the axiom's conclusion. The result is our generalized starting E-PEG for that proof. We then generate a rule stating that whenever this starting E-PEG is found, the final conclusion of the proof is added, in this case the $\triangle$ equality.

The details of how our E-PEG category is designed affects the optimizations that our approach can learn. For example, the category described above has free variables, but they only range over E-PEG nodes. For additional flexiblity, we can also introduce free variables that range over node operators, such as variables OP_1 and OP_2 in Figure 13.1. This would allow us to generate optimizations that are valid for any operator, for example pulling an operation out of a loop if its arguments are invariant in the loop. For even more flexibility, we can augment our E-PEG catgory with domain-specific relationships on operator variables, which could be used to indicate that one operator distributes over another. With this additional flexiblity, we can learn the more general version of LIVSR show in Figure 13.1. In all these cases, to learn the more general optimizations, one has to not only add flexiblity to the category, but also re-express the axioms so that they take advantage of the more general category (as was shown in Section 13.1). The E-PEG category can also be augmented with new structure in order to accommodate analyses not based on equalities. For example, an alias analysis could add a distinctness relation to identify when two references point to different locations. This would allow our generalization technique to apply beyond the kinds of equality-based optimizations that Peggy currently performs.

Acknowledgements

This chapter contains material taken from “Generating Compiler Optimization from Proofs”, by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

Chapter 16

More Applications of Generalization

The main advantage of having an abstract framework for proof generalization is that it separates the domain-independent components of proof generalization — how to combine pullbacks, pushouts, and pushout completions — from the domain-specific components of the algorithm — how to compute pullbacks, pushouts, and pushout completions. As a result, not only does this abstraction provide us with a significant degree of flexibility within our own domain of E-PEGs, as described in Chapter 15, but it also enables applications of proof generalization to problems unrelated to E-PEGs. We illustrate this point by showing how our generalization framework from Chapter 14 can be used to learn efficient query optimizations in relational databases (Section 16.1), assist programmers with debugging static type errors (Section 16.2), and improve polymorphism in programs that have already been type checked (Section 16.3).

16.1 Database Query Optimization

In relational databases, a small optimization in a query can produce massive savings. However, these optimizations become more expensive to find as the query size grows and as the database schema grows. We focus here on the setting of conjunctive queries, which are existentially quantified conjunctions of the predicates defined in the database schema. For example, the query $\exists y.\ R(x,y) \wedge R(y,z)$ returns all elements x and z

for which there exists a y such that (x,y) and (y,z) are in the R table (relation). For sake of brevity, we discuss only conjunctive queries without existential quantification.

A conjuctive query can itself be represented as a small database. For example, the query $q := R(x,y,z,1) \wedge R(x',y,0,1)$ can be represented by the following database (our notation assumes there is one table in the database called R and just lists the tuples in R):

$$Q := \boxed{\begin{array}{cccc} x & y & z & 1 \\ x' & y & 0 & 1 \end{array}}$$

Any result produced by q on a database instance I corresponds to a relation-preserving and constant-preserving function from Q to I. One nice property of this representation is that the number of joins required to execute a query is exactly one less than the number of rows in the small database representing the query. Thus, reducing the number of rows means reducing the number of joins.

Most databases have some additional structure known by the designer. One such structure could be that the first column of R determines the third column (we will use A, B, C, and D to refer to the columns of R). This is known as a functional dependency, noted by $A \to C$. Functional dependencies fit into the broader class of equality-generating dependencies since they can be used to infer equalities. A query optimizer can exploit this information to reduce the number of variables in a query, identify better opportunities for joins, or even identify redundant joins. Unfortunately, the functional dependency $A \to C$ provides no additional information for our example query, at least not yet.

Another form of dependencies is known as tuple-generating dependencies. These dependencies take the form “if these tuples are present, then so are these”. One common example is known as multi-valued dependencies. Suppose in our example database, the designer knows that, for a fixed element in B, column A is completely independent of C and D. In other words, $R(a,b,c,d) \wedge R(a',b,c',d')$ implies $R(a,b,c',d')$, as well as

$R(a',b,c,d)$. This is denoted as $B \twoheadrightarrow A$ or equivalently as $B \twoheadrightarrow CD$.

Adding tuples to a query in general is harmful because each added tuple represents an additional join. However, combined with equality-generating dependencies, these additional tuples can be used to infer useful equalities, which can then simplify the query. Let us apply an algorithm known as "the chase" [25] to optimize our example query using $A \rightarrow C$ and $B \twoheadrightarrow A$:

$$\boxed{\begin{array}{cccc} x & y & z & 1 \\ x' & y & 0 & 1 \end{array}} \overset{B \twoheadrightarrow A}{\mid\!\Longrightarrow} \boxed{\begin{array}{cccc} x & y & z & 1 \\ x' & y & 0 & 1 \\ x & y & 0 & 1 \end{array}} \overset{A \rightarrow C}{\mid\!\Longrightarrow} \boxed{\begin{array}{cccc} x & y & 0 & 1 \\ x' & y & 0 & 1 \end{array}}$$

The added tuple was used to infer that z must equal 0, which then simplifies the rightmost database above into two tuples. The optimizer can use this to select only tuples with C equal to 0 before joining, a potentially huge savings. Although this example was beneficial, many times adding tuples is harmful because it adds additional joins which can be inefficient. Thus, a query optimizer prefers to infer equalities without introducing unnecessary tuples.

Our framework from Chapter 14, instantiated to the database setting, can use instances of optimized queries to identify general rules for when adding tuples to a query is helpful. In particular, in the above example, it could identify exactly what properties of the original query led to the inferred equality. The category we will use in this example is **Rel**(4): quaternary relations and relation-preserving functions. The "axiom" $A \rightarrow C$ can be expressed categorically by the morphism

$$\boxed{\begin{array}{cccc} a & b & c & d \\ a & b' & c' & d' \end{array}} \xrightarrow{c,\, c' \mapsto \bar{c}} \boxed{\begin{array}{cccc} a & b & \bar{c} & d \\ a & b' & \bar{c} & d' \end{array}}$$

The “axiom” $B \twoheadrightarrow A$ can be expressed using the morphism

$$\boxed{\begin{array}{cccc} a & b & c & d \\ a' & b & c' & d' \end{array}} \rightarrow \boxed{\begin{array}{cccc} a & b & c & d \\ a' & b & c' & d' \\ a & b & c' & d' \end{array}}$$

Applying our framework to our sample query optimization sequence will produce the theorem

$$\boxed{\begin{array}{cccc} a & b & c & d \\ a' & b & c' & d' \end{array}} \xrightarrow{c,\, c' \mapsto \bar{c}} \boxed{\begin{array}{cccc} a & b & \bar{c} & d \\ a' & b & \bar{c} & d' \end{array}}$$

or simply $B \rightarrow C$. Thus, our framework can be used to learn equality-generating dependencies, removing the need for the intermediate generated tuples. This was possible because the dependencies involved, namely $A \rightarrow C$ and $B \twoheadrightarrow A$, could be expressed categorically as morphisms. We have proven that our learning technique can be used so long as all the dependencies can be expressed in this manner. Although the primary purpose of applying our framework to database optimizations was to demonstrate the flexibility of our framework, discussions with an expert in the database community [24] have revealed that our technique is in fact a promising approach that would merit further investigation.

16.2 Type Debugging

As type systems grow more complex, it also becomes more difficult to understand why a program does not type check. Type systems relying on Hindly-Milner type inference [56] are well known for producing obscure error messages since a type error can be caused by an expression far removed from where the error was finally noticed by the compiler. Below we show how to apply our framework as a type-debugging assistant that is similar to [39], but is also easily adaptable to additional language features such as type classes.

In Haskell, heap state is an explicit component of a type. For example, `readSTRef` is the function used to read references. This is a stateful operation, so it has type $\forall s\ a.\ \texttt{STRef}\ s\ a \to \texttt{ST}\ s\ a$. `STRef` $s\ a$ is the type for a reference to an a in heap s. `ST` $s\ a$ stands for a stateful computation using heap s to produce a value of type a. In order to use this stateful value, Haskell uses the type class `Monad` to represent sequential operations such as stateful operations. Thus `ST` s is an instance of `Monad` for any heap s. A problem that quickly arises is that operations such as `+` take two `Int`s, not two `ST` s `Int`s. Thus, `+` has to be lifted to handle effects. To do this, there is a function `liftM2` that lifts binary functions to handle effects encoded using any `Monad`. Likewise, `liftM` lifts unary functions.

Now consider the task of computing the maximum value from a list of references to integers. If the list is empty, the returned value should be $-\infty$. In Haskell, integers with $-\infty$ are encoded using the `Maybe Int` type: the `Nothing` case represents $-\infty$ and the `Just n` case represents the integer n. Conveniently, `max` defined on `Int` automatically extends to `Maybe Int`. The following program would seem to accomplish our goal:

```
maxInRefList refs
  = case refs of
      []         -> Nothing
      ref : tail -> liftM2 max
                      (liftM Just (readSTRef ref))
                      (maxInRefList tail)
```

Since `readSTRef` is a stateful operation, the lifting functions `liftM2` and `liftM` allow `max` and `Just` to handle this state. Unfortunately, this program does not type check.

The Glasgow Haskell Compiler, when run on the above program using `do` notation for the recursive call, produces the error "`readSTRef ref` has inferred type `ST` s a but is expected to have type `Maybe` a". This error message does not point directly to the problem, so the programmer has to examine the program, possibly even callers of `maxInRefList`, to understand why the compiler expects `readSTRef ref` to have a different type. Within `maxInRefList` alone there are many possiblities, such as the lifting operations, dealing with `Maybe` correctly, and the recursive call. Here we can apply proof generalization to limit the scope of where the programmer has to search, thereby helping identify the cause of the type error.

Type inference can be encoded categorically using a category of typed expressions. An object is a set of program expressions and a map from these program expressions to type expressions, although this map is not required to be a valid typing. Program expressions can have program variables, and type expressions can have type variables. A morphism from $\mathcal{A}$ to $\mathcal{B}$ is a type-preserving substitution of program and type variables in $\mathcal{A}$ to program and type expressions in $\mathcal{B}$ such that when the substitution is applied to $\mathcal{A}$, the resulting expressions are subexpressions of the ones in $\mathcal{B}$. In this category, typing rules can be encoded as morphisms. For example, function application can be encoded as:

$$\boxed{((f:\alpha)\ (x:\beta)):\gamma} \xrightarrow{\alpha \mapsto (\beta \to \gamma)} \boxed{((f:\beta \to \gamma)\ (x:\beta)):\gamma}$$

This states that, for any program expressions f and x where x has type β, f must have type $\beta \to \gamma$ for $f\ x$ to have type γ. Hence, the type α of f is mapped to $\beta \to \gamma$ by the morphism. In effect, applying this axiom unifies the type of f with $\beta \to \gamma$.

Rules for polymorphic values can also be encoded as morphisms. For example, the rule for `Nothing` can be encoded as:

$$\boxed{\texttt{Nothing} : \alpha} \xrightarrow{\alpha \mapsto \texttt{Maybe}\ \beta} \boxed{\texttt{Nothing} : \texttt{Maybe}\ \beta}$$

This states that, for the value `Nothing` to have type α, there must exist a type β such that α equals `Maybe` β. As before, applying this axiom unifies the type of `Nothing` with `Maybe` β.

Putting aside type classes for simplicity, the rule for `liftM` is:

$$\boxed{\texttt{liftM} : \alpha} \xrightarrow{\alpha \mapsto (\beta \to \gamma) \to M\ \beta \to M\ \gamma} \boxed{\texttt{liftM} : (\beta \to \gamma) \to M\ \beta \to M\ \gamma}$$

This rule uses a type variable M, which is treated like other type variables except it maps to unary type constructors, such as `Maybe` or the partially applied type constructor `ST` s.

Going back to the `maxInRefList` example, since the compiler expects the expression `readSTRef ref` to have type `Maybe` a, the type-inference process could be made to produce a proof that this fact must be true for the program to type check. This proof can be expressed categorically using the above encoding, which allows us to now apply our generalization technique. We ask the question "Why does `readSTRef ref` need to have type `Maybe` a?" categorically using a morphism from object (x : `Maybe` ζ) that maps x to `readSTRef ref` and ζ to a. We then proceed backwards through the inference process. For each step, we determine whether it contributes to the property; if it does, we generalize it, otherwise we skip the step entirely so as not to needlessly constrain the program.

The first useful step to generalize is the function-application rule where the function is `liftM Just` and the argument is `readSTRef ref`. During inference, before applying this axiom, `liftM Just` had type $M\ a \to M$ (`Maybe` α) for some M, a, and α;

`liftM Just (readSTRef ref)` had type `Maybe` β for some β; and `readSTRef ref` still had the unconstrained type γ. Applying the function-application rule during inference causes γ to be unified with $M\ a$ and M (`Maybe` α) with `Maybe` β. In turn, this forces M to unify with `Maybe`, contributing to the reason why `readSTRef ref` must have type `Maybe` a. Generalization can analyze this axiom application to determine that `readSTRef ref` has type `Maybe` a due to two key properties: (1) `liftM Just` had type $M\ a \rightarrow M\ \delta$ (where δ generalizes `Maybe` α), and (2) `liftM Just (readSTRef ref)` had type `Maybe` β (the same as the non-generalized type).

Generalizing property (1) eventually recognizes `liftM` as an important value in the program, whereas `Just` is not. Generalizing property (2) reaches similar kinds of conclusions in the rest of the program. In this manner, generalization identifies exactly which components of the program are causing the compiler to expect `readSTRef ref` to have type `Maybe` a. The resulting skeleton program is shown below, using dots for irrelevant expressions:

```
. = case . of
      . -> Nothing
      . -> liftM2 . (liftM . .) .
```

The skeleton program makes it clear that only the two cases, the lifting operations, and the use of `Nothing` are causing the incorrect expectation. Combining these three facts, the programmer can quickly realize that they forgot to lift the stateless value `Nothing` into the stateful effect `ST` s, easily fixed by passing `Nothing` to the `return` function. This mistake was hidden before because `Maybe` is coincidentally an instance of `Monad`, so the lifting functions were interpreted as lifting `Maybe` rather than `ST` s. The mistake was in a different case than where the error was reported, misleading the programmer into examining the wrong part of the program. Generalization, however, helps the programmer pinpoint the problem by removing parts of the program that do not contribute to the error.

16.3 Type Polymorphization

Applying proof generalization to type checking will automatically generalize the types of a program into polymorphic types. For example, take the function

```
int sum(l:int list, i:int) := foldr (+) i l
```

Now suppose type classes are added to the language and + is an operator defined for any instance of type class `Num`. Rather than have a programmer go through all prior code involving integers and determine which apply to arbitrary `Num`, we can generalize the proof that a typing is valid for a given program to determine the minimal type constraints necessary for the program to still type check.

In order to apply our framework from Chapter 14 to type derivations, we must encode type derivations as a category-theoretic inference process – Diagram (14.3). There are several ways of doing this, the most traditional of which would be to have an object in the category be a type derivation, and an axiom be an operation that adds one extra step at the end of the derivation. Thus, the sequence $\mathcal{E}_i$ from Diagram (14.3) would be a sequence in which each type derivation $\mathcal{E}_i$ adds one additional step to derivation $\mathcal{E}_{i-1}$. Here we use an encoding that is more direct and easier to illustrate, although less traditional. This alternate encoding also illustrates the flexibility of our category-theoretic framework, which can support many different encoding mechanisms for proofs.

An object in our category is a typed expression, which is a triple consisting of: a program expression, a function from subexpressions to types, and a unary predicate on subexpressions indicating whether the type of the given subexpression is known to be valid. We graphically represent an object in our category by writing down the expression, using ":" to show the types of subexpressions, and using a check mark ✓ to indicate that that the type of a subexpression is known to be valid. For example, the typed expression `(x:✓int + y:int):int` is the expression `x+y` where `x`, `y`, and `x+y` are all assigned type `int`, and only the typing of `x` is known to be valid. The inference process starts with a typed expression that has all the types but has no check marks, and each step in the inference process adds a check mark.

A morphism from a typed expression *p* to a typed expression *q* is a mapping *m* from the program variables in *p* to those in *q* and from the type variables in *p* to those in *q* such that *p* after substituting with *m* is a subexpression of *q*. This map needs to also preserve the subexpression-to-type mapping, as well as the type-validity predicate.

To see how a regular typing rule would be encoded as an axiom in our category, consider a typing rule of the following form, where $\mathtt{P}[e_1, e_2]$ is some expression composed of e_1 and e_2:

$$P\text{-rule}\,\frac{e_1 : \tau_1 \qquad e_2 : \tau_2}{\mathtt{P}[e_1, e_2] : \tau_3}$$

In our category theory formulation, this would be encoded as the following axiom:

$$\boxed{\mathtt{P}[e_1 :^{\checkmark} \tau_1,\, e_2 :^{\checkmark} \tau_2] : \tau_3} \xrightarrow{\mathcal{P}\text{-}rule} \boxed{\mathtt{P}[e_1 :^{\checkmark} \tau_1,\, e_2 :^{\checkmark} \tau_2] :^{\checkmark} \tau_3}$$

In order to type check the `sum` function, we must validate the typing of its body assuming that the type of the parameters are valid. In other words, starting with the object:

```
(foldr (+:int->int->int) (i:✓int) (l:✓int list)):int
```

we want to establish:

$$\texttt{(foldr (+:}^{\checkmark}\texttt{int->int->int) (i:}^{\checkmark}\texttt{int) (l:}^{\checkmark}\texttt{int list)):}^{\checkmark}\texttt{int}$$

To validate `+:int->int->int`, we apply the following axiom:

$$\boxed{\begin{array}{c} + : \tau \to \tau \to \tau \\ \text{where } \texttt{Num}\ \tau \text{ holds} \end{array}} \xrightarrow{\mathit{plus}} \boxed{\begin{array}{c} + :^{\checkmark} \tau \to \tau \to \tau \\ \text{where } \texttt{Num}\ \tau \text{ holds} \end{array}}$$

This axiom has been updated to use type classes rather than have one axiom for each elemental number type. Next, we apply the following axiom to validate the whole expression:

$$\boxed{\begin{array}{l} \texttt{(foldr} \\ \quad (f :^{\checkmark} \alpha \to \beta \to \beta) \\ \quad (b :^{\checkmark} \beta) \\ \quad (l :^{\checkmark} \alpha\ \texttt{list}) \\ \texttt{):}\ \beta \end{array}} \xrightarrow{\mathit{foldr}} \boxed{\begin{array}{l} \texttt{(foldr} \\ \quad (f :^{\checkmark} \alpha \to \beta \to \beta) \\ \quad (b :^{\checkmark} \beta) \\ \quad (l :^{\checkmark} \alpha\ \texttt{list}) \\ \texttt{):}^{\checkmark}\ \beta \end{array}}$$

Now that we have a proof of type validity expressed categorically, we can generalize this proof. We first ask the question “Why is the expression well typed?” using the morphism from the object $\texttt{e:}^{\checkmark}\tau$ mapping `e` to `foldr (+) i l` and τ to `int`. Applying the axiom *foldr* backwards will transform $\texttt{e:}^{\checkmark}\tau$ into

$$\texttt{(foldr (f:}^{\checkmark}\ \alpha\texttt{->}\beta\texttt{->}\beta\texttt{) (i:}^{\checkmark}\ \beta\texttt{) (l:}^{\checkmark}\texttt{list } \alpha\texttt{)):}\alpha$$

The axiom for + will replace `f` with +, unify α and β (say as τ), and add the constraint

Num τ:

$$\texttt{(foldr (+:}\tau\texttt{->}\tau\texttt{->}\tau\texttt{) (i:}^{\checkmark}\,\tau\texttt{) (l:}^{\checkmark}\texttt{list }\tau\texttt{)):}\tau \quad | \quad \texttt{Num}\,\tau$$

Thus, we have automatically added polymorphism from `int sum(list int, int)` to $\forall\tau \in \texttt{Num}.\ \tau\ \texttt{sum(list}\ \tau\texttt{,}\ \tau\texttt{)}$. Although this is just a toy example, it demonstrates a potential application of our framework to another domain very different from E-PEGs.

Acknowledgements

This chapter contains material taken from “Generating Compiler Optimization from Proofs”, by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

Chapter 17

Manipulating Proofs

Given a proof of correctness, our generalization technique produces the most general optimization for which the same proof applies. This still allows different proofs of the same fact to produce incomparable generalizations. However, by changing proofs intelligently, we can ensure better generalizations. Below we illustrate three classes of proof edits that we use to produce more broadly applicable optimizations: sequencing axiom applications, removing irrelevant axiom applications, and decomposing proofs.

17.1 Sequencing Axiom Applications

Our generalization technique requires proofs to be represented as a sequence of linear steps. However, proofs are often expressed as trees, in which case one needs to linearize the tree before our technique is applicable. The most faithful encoding of a proof tree is to use "parallel" axiom applications to directly encode the tree: each step in the linearized proof corresponds to the parallel application of all axioms in one layer of the proof tree. This encoding is the most faithful linearization of a proof tree because the tree can be reconstructed from the linearization.

In a traditional proof tree, two branches of a proof are essentially applying axioms in parallel, meaning their assumptions are checked independently and their conclusions are inferred independently. In certain logics, it is possible to combine two axioms into one “axiom” encoding both in parallel. For example, if we have the axioms $\forall x.x+0=x$ and $\forall x.x-x=0$, we can combine them into $\forall x,y.x+0=x\wedge y-y=0$. This process can be described categorically using coproducts, a concept closely related to pushouts.

For many, coproducts are best likened to sum types. A sum type defines a type with two cases where each case has its own type. Suppose the type of the first case is $\mathcal{A}$, and the type of the second case is $\mathcal{B}$, and we refer to their sum as $\mathcal{A}+\mathcal{B}$. This sum type comes with constructors, say $\iota_{\mathcal{A}}:\mathcal{A}\to\mathcal{A}+\mathcal{B}$ and $\iota_{\mathcal{B}}:\mathcal{B}\to\mathcal{A}+\mathcal{B}$. This sum type can also be destructed using case matching. Suppose we want to define some function which takes an inhabitant of $\mathcal{A}+\mathcal{B}$ and produce an inhabitant of some desired type $\mathcal{C}$. We can split our inhabitant of $\mathcal{A}+\mathcal{B}$ into two cases and produce an inhabitant of $\mathcal{C}$ in each case. Suppose $f:\mathcal{A}\to\mathcal{C}$ produces the inhabitant for the first case, and $g:\mathcal{B}\to\mathcal{C}$ produces the inhabitant for the second case. By using case matching, we can combine these into a function $[f,g]:\mathcal{A}+\mathcal{B}\to\mathcal{C}$. More than that, we also have the property that $\iota_{\mathcal{A}}\,;[f,g]$ is equivalent to f, and likewise $\iota_{\mathcal{B}}\,;[f,g]$ is equivalent to g. All these properties make the “sink” $\mathcal{A}\xrightarrow{\iota_{\mathcal{A}}}\mathcal{A}+\mathcal{B}\xleftarrow{\iota_{\mathcal{B}}}\mathcal{B}$ a coproduct. The categorical definition of a coproduct is defined below.

Definition 17.1 (Coproduct)**.** *A sink $\mathcal{A}\xrightarrow{\iota_{\mathcal{A}}}\mathcal{A}+\mathcal{B}\xleftarrow{\iota_{\mathcal{B}}}\mathcal{B}$ is said to be a* coproduct *if, for any object $\mathcal{C}$ and morphisms $f:\mathcal{A}\to\mathcal{C}$ and $g:\mathcal{B}\to\mathcal{C}$, there exists a unique morphism (denoted $[f,g]$) from $\mathcal{A}+\mathcal{B}$ to $\mathcal{C}$ such that the following diagram commutes:*

$$\begin{tikzcd}
\mathcal{A} \arrow[dr, "\iota_{\mathcal{A}}"'] \arrow[drr, "f"] & & \\
 & \mathcal{A}+\mathcal{B} \arrow[r, dashed] & \mathcal{C} \\
\mathcal{B} \arrow[ur, "\iota_{\mathcal{B}}"] \arrow[urr, "g"'] & &
\end{tikzcd}$$

The coproduct operation $+$ *constructs the coproduct of two objects, and the* $[,]$ *operation constructs the uniquely induced morphism.*

In addition, we define the coproduct of two morphisms $f : \mathcal{A} \to \mathcal{C}$ and $g : \mathcal{B} \to \mathcal{D}$ as $f + g = [f\,; \iota_{\mathcal{C}}, g\,; \iota_{\mathcal{D}}] : \mathcal{A} + \mathcal{B} \to \mathcal{C} + \mathcal{D}$. Intuitively, $f + g$ applies f to the first case and g to the second case.

Given two axioms expressed categorically as $\mathcal{A}_1 \xrightarrow{axiom_1} \mathcal{C}_2$ and $\mathcal{A}_2 \xrightarrow{axiom_2} \mathcal{C}_2$, we can encode $axiom_1$ and $axiom_2$ in parallel as $axiom_1 + axiom_2 : \mathcal{A}_1 + \mathcal{A}_2 \to \mathcal{C}_1 + \mathcal{C}_2$. Given an application $app_1 : \mathcal{A}_1 \to \mathcal{E}$ of $axiom_1$ and an application $app_2 : \mathcal{A}_2 \to \mathcal{E}$ of $axiom_2$ to the same instance $\mathcal{E}$, we can use the coproduct property to construct an application $[app_1, app_2] : \mathcal{A}_1 + \mathcal{A}_2 \to \mathcal{E}$ of the parellelized axioms $axiom_1 + axiom_2$. We have proven that the pushout of this axiom application produces the same result (up to isomorphism) of pushing out app_1 and then pushing out app_2 on the resulting instance (and likewise for the reverse order). Thus, the parallelized axiom does in fact encode both axioms together in an order-independent manner. Using this encoding, we can encode a proof tree by parallelizing all the axioms in the same layer of the proof and then sequencing the layers of the proof.

At this point, we consider the impact that this process has on proof generalization. Given two axiom applications for the same instance, we now have the choice of sequencing these two applications or parallelizing them. First, it is the case that different sequences of the same axiom applications produce different, even incomparable, generalizations. Interestingly though, we have proven that *any* sequential application of axioms always produces a more general (or isomorphic) proof generalization than the parallelized application. Thus, when given the choice, it is always better to apply axioms in sequence rather than in parallel, but unfortunately there may not be an optimal ordering.

To demonstrate how this might happen, suppose $axiom_1$ is $\phi_1 \Rightarrow \psi_1 \wedge \phi_2$ and $axiom_2$ is $\phi_2 \Rightarrow \psi_2 \wedge \phi_1$. The important property is that $axiom_1$ implies $axiom_2$'s premise and vice versa. The parallelized axiom would be $\phi_1 \wedge \phi_2 \Rightarrow \psi_1 \wedge \psi_2 \wedge \phi_1 \wedge \phi_2$. Now suppose in our proof instance we had assumed ϕ_1 and ϕ_2 and concluded with ϕ_1, ϕ_2, ψ_1, and ψ_2, and we want to generalize our proof in order to weaken our assumptions but still infer ψ_1 and ψ_2. If we use the parallelized axiom, our generalized proof is no better than our original proof. If we apply $axiom_1$ and then $axiom_2$, our generalized proof will only require ϕ_1 to be assumed, since $axiom_1$ infers ϕ_2 as required by $axiom_2$. If instead we apply $axiom_2$ and then $axiom_1$, our generalized proof will only require ϕ_2 to be assumed, since $axiom_2$ infers ϕ_1 as required by $axiom_1$. Thus, neither generalization of either sequential proof is better than the other, but both generalizations are better than that of the parallelized proof. The intuition is that the parallel axiom requires the premise of both axioms, whereas sequencing enables the first axiom to infer the premise of the following axiom so that the second premise is not required in the generalized proof.

This result suggests that our use of sequential proofs is not limiting but in fact enables better generalizations of proofs. In our implementation, our proofs have the property that sequential and parallel forms all produce the same generalization though. This is because the axioms used by our implementation can only interfere, like in the example above, when one application is redundant, and our implementation only produces proofs without redundant axiom applications. This property simplifies our implementation since it allows us to generalize axioms in any order and always produce the same result. We have not, however, researched categorical techniques for determining when axiom sets defined in a arbitrary logic will have this same non-interference property. We leave this to the research domain of proof theory.

17.2 Removing Irrelevant Axiom Applications

Sometimes certain axiom applications infer information that is irrelevant to the final property that we are interested in concluding. An irrelevant axiom application can overly restrict the generalized optimization by making certain equalities (those required by the axiom) seem important to the optimization when they are not. Prior to generalization, it is difficult to identify which steps of the proof are relevant to the optimization. However, since generalization proceeds backwards through the proof, each step of the algorithm can easily identify when an axiom application is not contributing to the current property being generalized and simply skip it. In essence, our algorithm edits the original proof on the fly, as generalization proceeds, to remove steps not useful for the end goal.

17.3 Decomposition

As mentioned in Section 13.4, we decompose generated optimizations into smaller optimizations that are more broadly applicable. We can view decomposition as taking the original proof and cutting it up into smaller lemmas before applying generalization. In the context of E-PEGs, performing decomposition requires us to determine the set of inferred equalities along which we want to cut the proof (the first step mentioned in Section 13.4). Formally, we represent the set of cut-points as an object $\mathcal{S}$ and a morphism $\mathit{sub} : \mathcal{S} \to \mathcal{E}_n$, where $\mathcal{E}_n$ is the final inferred result of the proof. Then, in each step of generalization, we check whether the current property prop_m being generalized is contained within sub by determining whether there exists a morphism from $\mathcal{P}_m$ to $\mathcal{S}$ such that the following diagram commutes:

$$\begin{tikzcd}
 & \mathcal{A}_m \arrow[r, "\mathit{axiom}_m"] \arrow[d, "\mathit{app}_m"'] & \mathcal{C}_m \arrow[d] & \mathcal{P}_m \arrow[r, "?"] \arrow[dl, "\mathit{prop}_m"] & \mathcal{S} \arrow[d, "\mathit{sub}"] \\
\cdots \arrow[r] & \mathcal{E}_{m-1} \arrow[r] & \mathcal{E}_m \arrow[r] & \cdots \arrow[r] & \mathcal{E}_n
\end{tikzcd}$$

This morphism essentially describes how to contain prop_m within sub. If this is possible, we conclude "prop_m implies prop_n" as a generalized lemma. We then continue generalizing, but now prop_m will be the conclusion of the next generalized lemma. We do this at each point where $prop_m$ can be contained within sub, thus splitting the proof into smaller lemmas, each of which is generalized.

Acknowledgements

This chapter contains material taken from “Generating Compiler Optimization from Proofs”, by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

Chapter 18

Evaluation of Learning

We used Peggy and E-PEGs to implement our technique for generating optimizations (as was illustrated in Chapters 13 and 15). In this chapter, we experimentally validate three hypotheses about our technique: (1) our technique allows a programmer to easily extend the compiler by sketching what an optimization looks like with a before-and-after example; (2) our technique can amortize the cost of expensive-to-run superoptimizers by generating fast-to-run optimizations; and (3) our technique can even learn optimizations that are significantly profitable on code that the compiler was *not* trained on.

18.1 Extending the Compiler through Examples

When the compiler does not perform an optimization that the programmer would like to see, our technique allows the programmer to train the compiler by providing a concrete example of what the desired optimization does. By using our implementation to learn a variety of optimizations in this way, we demonstrate experimentally that our technique enables the compiler to be extensible in an easy-to-program manner, without the programmer having to learn any new language or compiler interface.

The optimizations that our system learned from examples are listed in Figure 18.1. It took on average of 3.5 seconds to learn each example (including the time for translation

Optimization	Description
LIVSR	Loop-induction variable SR
Inter-Loop SR	SR across two loops
LIVSR Bounds	Optimizes loop bounds after LIVSR
ILSR Bounds	Optimizes loop bounds after inter-loop SR
Fun-Specific Opts	Function-specific optimizations
Spec Inlining	Inline only for special parameter values
Partial Inlining	Inline only part of the callee
Temp-Obj Removal	Remove temporary objects
Loop-op Factor	Factor op out of loop
Loop-op Distr	Distribute op into loop to cancel other ops
Entire-Loop SR	Replace entire loop with one op
Array-Copy Prop	Copy prop through array elements
Design-Patt Opts	Remove overhead of design patterns

Figure 18.1. Learned optimizations (SR = Strength Reduction)

validation). The only optimizations in this list that are performed by `gcc -O3` are LIVSR and Array-Copy Prop. This demonstrates the benefit of our system: it allows an end-user programmer to easily implement non-standard optimizations targeting application domains with high-performance needs, such as computer graphics, image processing, and scientific computing.

The LIVSR optimization was already shown in Chapter 13. Optimizations LIVSR Bounds through Loop-op Factor will be covered throughout the remainder of Chapter 18. We start with Inter-Loop SR (from Section 3.3) and ILSR Bounds. These optimizations apply to a common programming idiom in image processing, which is shown in the left part of Figure 18.2(a). The `img` variable is a two-dimensional image represented as a one-dimensional array in row-major form. Programmers commonly use this kind of representation to efficiently store dynamically sized two-dimensional images. The original code in Figure 18.2(a) uses a convenient way of iterating over such arrays, whereas the transformed code uses the more efficient, yet harder to program, formulation that programmers typically use (it removes the multiplication and addition from the

```
for (i=0; i<h; i++)
  for (j=0; j<w; j++)
    img[i*w+j] /= 2;
```

$\Longmapsto$

```
for (i=0; i<h*w; i+=w)
  for (j=i; j<i+w; j++)
    img[j] /= 2;
```

(a) Concrete example

```
for (I=E1; I<E2; I++)
  for (J=0; J<E3; J++)
    E4(I*E3+J)
```

$\Longmapsto$

```
for (I=E1; I<E2*E3; I+=E3)
  for (J=I; J<I+E3; J++)
    E4(J)
```

where E_1, E_2, E_3, and E_4 are any loop-invariant expressions

(b) Generated optimization rule

Figure 18.2. Generalized inter-loop strength reduction

inner loop). From the concrete example, our generalizer determines that the `img` array is actually insignificant (the only part that matters is the `i*w + j` computation); it determines that the starting point of the outer loop is insignificant; and that the bounds on both loops can be generalized into loop-invariant expressions. Furthermore, although we show the learned optimization as one rule, our decomposition algorithm would split this into two optimizations: one which optimizes the body of the loop (Inter-Loop SR), and one which optimizes the bounds of the loop (ILSR Bounds). A similar decomposition when learning LIVSR would also produce LIVSR Bounds. Also, all our generated optimization rules — including the one in Figure 18.2(b) — can apply to programs containing other statements in the loops that do not affect the significant program fragments being optimized.

When traditional compilers like `gcc -O3` do not perform the optimization described above, an image-processing programmer would be forced to use the more efficient but error-prone version of the code. With our system, the programmer can use the simpler version and rely on the compiler to optimize it into the more efficient one. Furthermore, if the programmer encounters a new instance of the programming pattern that the generated rule does not cover, the programmer can train the compiler with another example. This

```
len = array.length;
sum = 0;
for (i=0; i<len; i++)
   sum += array[i] * 7;
```

$\Longmapsto$

```
len = array.length;
sum = 0;
for (i=0; i<len; i++)
   sum += array[i];
sum *= 7;
```

(a) Concrete example

```
X = 0
while (E1)
   X += E2 * E3
```

$\Longmapsto$

```
X = 0
while (E1)
   X += E2
X *= E3
```

where $\mathtt{E}_3$ is a loop-invariant expression

(b) Generated optimization rule

Figure 18.3. Loop-operation factoring

scenario emphasizes how easy it is for non-compiler-experts to benefit from our system. Extending the compiler is as simple as providing a single concrete example, without having to worry about the side conditions required for correctness or having to learn a new language, such as the language in Figure 18.2(b) including expression variables like $\mathtt{E}_1$ and side conditions like "$\mathtt{E}_1$ is loop invariant".

We next show another example of an optimization not performed by `gcc -O3`, loop-operation factoring (Loop-op Factor in Figure 18.1). The concrete example that we used to sketch the optimization is shown in Figure 18.3(a). The multiplication inside the loop gets factored out of the loop through the additions. The generalization that our system generates is shown in Figure 18.3(b). Our generalizer determined that the use of the array is insignificant and the format of the loop is insignificant, as is the constant 7, which can in fact be any loop-invariant expression.

Finally, we show how our system can learn optimizations for the `pow` function in Figure 18.4. With the concrete example `pow(a,p) * pow(b,p)` $\Rrightarrow$ `pow(a * b,p)`, our generalizer shows that the two concrete programs are equivalent and then generalizes the example into the optimization $\forall x,y,z \in$ `int`. `pow`(x,z) `*` `pow`$(y,z) \mapsto$ `pow`$(x$ `*` $y,z)$. This is an example of Fun-Specific Opts from Figure 18.1. Similarly, we were able to get our generalizer to learn the non-trivial optimization: $\forall x \in$ `uint`. `pow`$(2,x) \mapsto 1 \ll x$ (which is an example of Spec Inlining in Figure 18.1). Here again, neither of these optimizations are performed by `gcc -O3`, whereas our approach allows the programmer to easily specify these optimizations by example.

18.2 Learning from Superoptimizers

Another possible use of our approach is to amortize the cost of running a superoptimizer. Given an input program, a superoptimizer performs a brute-force exploration through the large space of transformed programs to find the (near) optimal version of the input program. Our approach can mitigate the cost of running superoptimizers by learning optimizations from one run of the superoptimizer, and then applying only the learned optimizations to get much of the benefit of the superoptimizer at a fraction of the cost.

The Peggy compiler at its core performs a superoptimizer-style brute-force exploration by applying axioms to build an E-PEG that compactly represents exponentially many different versions of the input program, and then using a profitability heuristic to find the best PEG represented in this E-PEG. To evaluate our approach in the setting of superoptimizers, we used Peggy to superoptimize some microbenchmarks and SpecJVM, and used our technique on the original and transformed programs to learn optimizations.

Several of the optimizations learned using before-and-after examples in Section 18.1 cannot be learned by using the superoptimizer – they really do require a programmer to give an input-output example. For instance, if the Peggy superoptimizer were given `pow(a,p) * pow(b,p)` to optimize using basic axioms, it would not be able to find the desired transformed program `pow(a * b,p)` because, even though it can

```
int pow(int base, int power)
  int prod = 1;
  for (int i = 0; i < power; i++)
    prod *= base;
  return prod;
```

Figure 18.4. The power function for integers

decompose the original expression into smaller pieces, it cannot guess how to reassemble them into `pow(a * b,p)`. However, Peggy can prove the original and transformed programs equivalent because it sees the `pow(a * b,p)` term to which it can apply axioms. In essence it is easier to apply axioms on the original and transformed programs and meet in the middle, rather than derive the transformed program from the original. With the proof, our approach can learn a new optimization that the Peggy superoptimizer would not have performed.

Our superoptimizer experiments also show how inlining combined with generalization produces useful and unconventional optimizations. Inlining in Peggy simply adds the information that a call node is equivalent to the body of the called function. Adding this equality does not force Peggy to choose the inlined version; it just provides more options in the caller's E-PEG for the profitability heuristic to choose from. When running on code that uses the `pow` function from Figure 18.4, Peggy applied the inlining axiom on `pow`, thus pulling the body of `pow` into the E-PEG. Peggy then exploited the fact that `pow` does not affect the heap to optimize the *surrounding context*, but chose not to inline `pow` in the final result. From this our generalizer learned the optimization that `pow` is heap invariant. In future compilations, Peggy can then immediately use the fact that `pow` does not affect the heap to optimize the surrounding context without the expensive process of inlining `pow`. This is an example of what we call *partial inlining* (Partial Inlining in Figure 18.1), where the optimizer exploits the information learned from inlining without opting to inline the function. Another example of partial inlining we observed involved a large function that modifies the heap but always returns 0. By applying the inlining axiom, Peggy was able to optimize the surrounding context with the information that the return value was 0, but then chose not to inline the called function because it was too large. In this case the generalizer would learn that the function always returns 0, which can be used in future compilations without having to apply the inlining axiom.

We evaluate the effectiveness of amortizing the cost of a superoptimizer on the SpecJVM suite. For each class in SpecJVM, as a first step we used the Peggy superoptimizer to compile the class using basic axioms, and used our generalization technique to generate a set of optimizations for that class. As a second step, we removed all the basic axioms from Peggy and re-optimized the class using the axioms that were learned in the first step. Across all benchmarks, it took on average 11.15 seconds to generalize a method, and the average cost of compiling each method went down from 26.64 seconds in the first step to 1.47 seconds in the second step, while still producing the same output programs.

These experiments show that, for expensive superoptimizers, our technique can improve compilation time while still producing the same result. Furthermore, our benchmark-specific optimizations can still apply even if small changes are made to the code. As a result, we can avoid a superoptimizing compilation after each small change, while still retaining many of the benefits of the superoptimizer. Eventually, however, the learned optimizations will go out of date, at which point a superoptimizing compilation would be needed. Thus, we envision that our approach could be used to perform an expensive superoptimizing compilation every so often while using the generated optimizations in between.

18.3 Cross-Training

The above experiment illustrates the benefits of learning optimizations when compiling exactly the same code that was learned on. We now show some encouraging evidence that even cross-training is possible. Our hypothesis is that many libraries are used in stylized ways, and so training with Peggy's superoptimizer on some uses of a library can discover optimizations that would be useful on previously unseen code that uses the same library.

We conducted a preliminary evaluation of this hypothesis on the Java ray tracer from Section 11.2, which uses a pure vector library. Peggy's optimization phase improves this benchmark's performance by 7%, compared to only using Sun's Java 6 JIT, by removing the short-lived temporary objects (Temp-Obj Removal in Figure 18.1).

Most of these gains come from one important method, call it `m`. We identified another vector-intense method, call it `f`, to train our learning optimizer on. Using only the axioms learned from optimizing the expressions within `f`, Peggy was able to optimize `m` to produce a 3.1% run-time improvement on the ray tracer, instead of the 7.1% speed-up gained by fully optimizing `m`. Alternatively, using a slightly larger training set produces a 5.1% speed-up. Upon further investigation, we found that the learned optimizations perform large-step simplifications of common usage patterns of the vector library (for example, a vector scale followed by a vector add). Furthermore, if in addition to the learned optimizations from either training set, we allow Peggy to also use those original axioms which infer equalities without creating any new terms (41% of all axioms), Peggy produces the fully optimized `m`. The purpose of these axioms is to simplify a program, so they cannot lead the optimizer down a fruitless path. Using simplifying axioms alone on `m` produces only a 0.6% speed-up. Thus, the optimizations learned from either training set lead the optimizer in the right direction, and the remaining axioms simplify the resulting expressions into the fully optimized `m`. These findings show that our technique can be effective at cross-training even on a small training set.

Acknowledgements

This chapter contains material taken from “Generating Compiler Optimization from Proofs”, by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

Chapter 19

Related Work

Superoptimizers Our approach of computing a set of programs and then choosing from this set is related to the approach taken by many superoptimizers [9, 33, 37, 54]. Superoptimizers strive to produce truly optimal code, rather than simply improve programs. Although superoptimizers 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, as shown by the inter-loop optimization from Chapter 3.

Our approach was inspired by Denali [45], a superoptimizer 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's 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.

Rewrite-Based Optimizers Axioms or rewrite rules have been used in many compilation systems, for example TAMPR [13], ASF+SDF [87], the ML compilation system of Visser et al. [88], and Stratego [14]. 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. Even with no strategies, we can perform a variety of intricate optimizations.

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 [93, 94], automated techniques for generating good sequences [2, 17, 46], manual techniques for combining analyses and optimizations [16], and automated techniques for the same purpose [52]. 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. Aside from the theoretical guarantees from Chapter 5, our approach can do well even if every part of the input program requires a different ordering.

Translation Validation In contrast to previous approaches to translation validation have been explored [59, 64, 69, 96], our approach has the advantage that it can perform translation validation by using the same technique as for program optimization. It also adapts easily to new compilers and languages, unlike the strategy taken by Tristan et al. [84] which requires a thorough understanding of the specific compiler and even compiler configuration at hand.

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 E-PEG representation. There has been a long line of work on developing IRs that make analysis and optimizations easier to perform [3, 15, 20, 31, 41, 63, 71, 85, 92]. The key distinguishing feature of E-PEGs is that a single E-PEG 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 [20], gated SSA [85] and thinned gated SSA [41]. 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, as shown in Chapter 4. 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 predecessor CFG basic blocks. The fact that our PEG representation is not tied to a CFG makes the reversion from PEGs back to a CFG much more challenging, since it requires reconstructing explicit control information.

The program dependence graph [31] (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 [62] (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 reverting back to a CFG more complex.

Dependence flow graphs [63] (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 [92] (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 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 witness rather than using the techniques from the functional-languages community such as monads [57, 89–91] or continuation-passing style [4, 5, 32, 40, 47], 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 [44]. The most closely related is the Lucid programming language [7], 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. Furthermore, we incorporate a *monotonize* function into our semantics and axioms, which guarantees the correctness of our conversion to and from CFGs with loops.

Theorem Proving Because most of our reasoning is performed using simple axioms, our work is related to the broad area of automated theorem proving. The theorem prover that most inspired our work is Simplify [23], with its E-graph data structure for representing equalities [61]. Our E-PEGs are in essence specialized E-graphs for reasoning about PEGs. Furthermore, the way our analyses communicate through equality is conceptually similar to the equality-propagation approach used in Nelson-Oppen theorem provers [60].

Execution Indices Execution indices identify the state of progress of an execution [27, 95]. 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.

Extensible Optimizers There has been a long line of work on making optimizers extensible or easier to develop, including Sharlit [83], Whitfield and Soffa's Gospel system [94], the Broadway extensible compiler [38], and the Rhodium system for expressing optimizations [48, 53]. In all these systems, however, the programmer has to learn a new language or compiler interface to express optimizations. In contrast, our approach learns an optimization from a single example provided by the programmer in a language already familiar to them.

Explanation-Based Learning In the context of artificial intelligence, our approach is an instance of explanation-based learning (EBL) [30]. EBL refers to learning from a single example using an explanation of that example. EBL has been applied to a wide variety of domains, such as Prolog optimization [35], logic-circuit designs [29], and software reuse [11]. Many of these applications use algorithms based on unification or Prolog [26, 29, 35]. The declarative components of Prolog can be encoded in our framework by combining categories of expressions with categories of relations. Furthermore, most EBL implementations provide no guarantees on what is learned, while we can prove that our technique learns the most general lesson for a given explanation. Within EBL, our work is closely related that of Dietzen and Pfenning [26]. They use λProlog to extend EBL to higher-order and modal logic, and apply this framework to various settings including program transformations. However, they do not investigate ways to automatically train the optimizer, relying instead on the user to prove the transformation correct using tacticals. As a consequence, they do not attempt to decompose an optimization into suboptimizations, since a user manually proving an optimization would already do this. Furthermore, we experimentally demonstrate that generalization can be useful for extending compilers and amortizing the cost of superoptimizers.

Machine Learning There have been many uses of machine learning in the context of compiler optimizations. Techniques like genetic algorithms, reinforcement learning, and supervised learning, have been used to generate effective heuristics for instruction scheduling [55, 73], register allocation [73], prefetching [73], loop-unroll factors [72], and for optimization ordering [18]. In all these cases, the parts being learned are not the transformation rules themselves, but profitability heuristics, which are functions that decide when or where to apply certain transformations. As a result, these techniques are complementary to our technique: we generate the optimization rules themselves, but not the profitability heuristics (we use a single global profitability heuristic for all optimizations). Also, while modern machine learning techniques use statistical methods over large data sets, our EBL-based approach can learn from a very small dataset, even from just one example.

Optimization Inference The idea of discovering optimizations has also been been explored in the setting of superoptimizers [9, 45, 54]. Superoptimizers try to discover optimizations by a brute-force exploration of possible transformed programs for a given input program. Traditional superoptimizers find concrete optimization *instances*, whereas our approach *starts* with optimization instances, and tries to generalize the instances into reusable optimization rules. As such, our work is complementary to superoptimization techniques. However, Bansal and Aiken's recent superoptimizer [9] does achieve a simple form of generalization, namely abstraction of register names and constants. In contrast, we perform a more sophisticated kind of generalization based on the reasons why the original and transformed programs are equivalent.

Acknowledgements

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*, 2009. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specfiic permission and/or a fee.

This chapter contains material taken from “Generating Compiler Optimization from Proofs”, by Ross Tate, Michael Stepp, and Sorin Lerner, which appears in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010. The dissertation author was the primary investigator and author of this paper. Some of the material in these chapters is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Chapter 20

Conclusion

The key to this work is our algebraic representation of imperative functions: Program Expression Graphs. This representation enabled us to extend the algebraic and equational reasoning capabilities of theorem provers to entire control flow graphs so that we may discover and verify even complex loop optimizations. It also provided a powerful proof language enabling us to learn even advanced loop optimizations by generalizing proofs in this expressive language. Thus, in our experience the program representation is the key component of an optimizer.

In this thesis I have focused entirely on intraprocedural optimization. We have investigated interprocedural optimizations; however, we have come across two major obstacles. First, we have not been able to design an algebraic interprocedural representation of programs, meaning a representation that does not use some form of intermediate variables, such as parameters, which obstruct the flow of data and restructurings. Second, the program size grows too large as does the space to explore, since there is little information as to where interprocedural transformations may actually prove beneficial. The former seems fundamentally insurmountable, so the latter is where we believe efforts should be focused. The current common strategy is to collect interprocedural information, such as aliasing, and apply that information intraprocedurally. Another strategy may be to enable programmers to annotate code with information indicating where interprocedural optimization may be beneficial, and simultaneously provide guarantees or feedback to programmers so that they are encouraged to supply this information.

Possibly the greatest weakness of this work is its dramatic difference from conventional optimization techniques. At present we do not have a means for combining our approach with conventional approaches so that they may cooperate with each other. This would also address the more practical consideration of integrating our techniques into existing compilers. I believe such a combination would be fruitful for both optimization and correctness, so that is my next step for furthering this promising line of research and putting it into practice.

Lessons from Category Theory In doing this research, it is surprising how helpful category theory has been considering how application-oriented this work is. When I got stuck attempting to find a good and principled optimization-generalization algorithm [79], category theory allowed me to abstract away the overwhelming details, identify the core components of the problem at hand, and tackle those challenges at a high level, with the low-level details being just instantiations of the high-level algorithm. In fact, this same situation happened again when trying to design and implement an inferable typed assembly language [77] and being overwhelmed with details and, in that case, design choices. In that case, the high-level algorithm actually transferred to an entirely different domain I had not anticipated, which enabled me to quickly devise an algorithm for improving Java's handling of wildcards [78]. Similarly, when attempting to prove that our representation of effects [75, 80, 81] was in fact sound, which thankfully none of the reviewers required back when I had never heard of categories, category theory offered the tools necessary to find a proper argument, especially one that could adapt to new languages and effects as I expected PEGs should be able to. So, if I may offer a lesson from my own personal experience, even if you are not working in areas with well known connections to category theory, such as semantics, you should expose yourself to some raw category theory [1]; like me, you may be surprised when and where it will come to the rescue.

Appendix A

Axioms

In this appendix we describe the axioms used to produce the optimizations listed in Figure 11.1. 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 11.1.

A.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 E-PEG operators) 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. Some require OP to be lifted with respect to a certain loop. For example, domain operators, such as $+$, as well as ϕ are lifted with respect to all loops. On the other hand, loop operators such as θ_ℓ, $eval_\ell$, and $pass_\ell$ are only lifted with respect to loops distinct from ℓ. We formalize this with the definition of *lifted*$_\ell$.

Definition A.1. *(Loop-liftedness) Putting aside the issue of arity for simplicity, given a function* $f : ((\mathscr{L} \to \mathbb{N}) \to X) \to ((\mathscr{L} \to \mathbb{N}) \to Y)$, *lifted*$_\ell(f)$ *is defined as*

$$\forall x, x' \in (\mathscr{L} \to \mathbb{N}) \to X, i \in \mathbb{N}. \begin{pmatrix} \forall \mathbf{i} \in \mathscr{L} \to \mathbb{N}.\ x(\mathbf{i}[\ell \mapsto i]) = x'(\mathbf{i}[\ell \mapsto i]) \\ \Downarrow \\ \forall \mathbf{i} \in \mathscr{L} \to \mathbb{N}.\ f(x)(\mathbf{i}[\ell \mapsto i]) = f(x')(\mathbf{i}[\ell \mapsto i]) \end{pmatrix}$$
$$\bigwedge$$
$$\forall x \in (\mathscr{L} \to \mathbb{N}) \to X. \begin{pmatrix} \forall \mathbf{i} \in \mathscr{L} \to \mathbb{N}, i, i' \in \mathbb{N}.\ x(\mathbf{i}[\ell \mapsto i]) = x(\mathbf{i}[\ell \mapsto i']) \\ \Downarrow \\ \forall \mathbf{i} \in \mathscr{L} \to \mathbb{N}, i, i' \in \mathbb{N}.\ f(x)(\mathbf{i}[\ell \mapsto i]) = f(x)(\mathbf{i}[\ell \mapsto i']) \end{pmatrix}$$

- if $T = \theta_\ell(\mathbf{A}, T)$ and *invariant*$_\ell(\mathbf{A})$, then $T = \mathbf{A}$

  [*If a loop-varying value always equals its previous value, then it equals its initial value*]

- if *invariant*$_\ell(\mathbf{A})$, then *eval*$_\ell(\mathbf{A}, \mathbf{P}) = \mathbf{A}$

  [*Loop-invariant values have the same value regardless of the current loop iteration*]

- $$\begin{aligned} \textsc{op}(\mathbf{A_1}, \ldots, \theta_\ell(\mathbf{B}_i, \mathbf{C}_i), \ldots, \mathbf{A}_k) = \theta_\ell(&\textsc{op}(\mathit{eval}_j(\mathbf{A}_1, \mathrm{Z}), \ldots, \mathbf{B}_i, \ldots, \mathit{eval}_\ell(\mathbf{A}_k, \mathrm{Z})),\\ &\textsc{op}(\mathit{peel}_j(\mathbf{A_1}), \ldots, \mathbf{C_i}, \ldots, \mathit{peel}_j(\mathbf{A_k}))),\\ &\text{when } \mathit{lifted}_\ell(\textsc{op}) \end{aligned}$$

  [*Any loop-lifted operator can distribute through* θ_ℓ]

- $\phi(\mathbf{C}, \mathbf{A}, \mathbf{A}) = \mathbf{A}$

  [*If a* ϕ *node has the same value regardless of its condition, then it is equal to that value*]

- $\phi(\mathbf{C}, \phi(\mathbf{C}, \mathbf{T}_2, \mathbf{F}_2), \mathbf{F}_1) = \phi(\mathbf{C}, \mathbf{T}_2, \mathbf{F}_1)$

  [*A ϕ node in a context where its condition is true is equal to its true case*]

- $\text{OP}(\mathbf{A_1}, \ldots, \phi(\mathbf{B}, \mathbf{C}, \mathbf{D}), \ldots, \mathbf{A_k}) = \phi(\mathbf{B}, \text{OP}(\mathbf{A_1}, \ldots, \mathbf{C}, \ldots, \mathbf{A_k}), \text{OP}(\mathbf{A_1}, \ldots, \mathbf{D}, \ldots, \mathbf{A_k}))$

  [*All operators distribute through ϕ nodes*]

- $\text{OP}(\mathbf{A}_1, \ldots, eval_\ell(\mathbf{A}_i, \mathbf{P}), \ldots, \mathbf{A}_k) = eval_\ell(\text{OP}(\mathbf{A_1}, \ldots, \mathbf{A}_i, \ldots, \mathbf{A}_k), \mathbf{P})$, when $lifted_\ell(\text{OP})$ and all of $\mathbf{A}_1, \ldots, \mathbf{A}_{i-1}, \mathbf{A}_{i+1}, \ldots, \mathbf{A}_k$ are $invariant_\ell$

  [*Any loop-lifted operator can distribute through $eval_\ell$*]

(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*$[i \to 0]$ *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 11.1.

- $(\mathbf{A}*\mathbf{B})+(\mathbf{A}*\mathbf{C}) = \mathbf{A}*(\mathbf{B}+\mathbf{C})$
- if $\mathbf{C} \neq 0$, then $(\mathbf{A}/\mathbf{C})*\mathbf{C} = \mathbf{A}$
- $\mathbf{A}*\mathbf{B} = \mathbf{B}*\mathbf{A}$

- $\mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}$
- $\mathbf{A}*1=\mathbf{A}$
- $\mathbf{A}+0=\mathbf{A}$
- $\mathbf{A}*0=0$
- $\mathbf{A}-\mathbf{A}=0$
- $\mathbf{A}\,\%\,8=\mathbf{A}\,\&\,7$
- $\mathbf{A}+(-\mathbf{B})=\mathbf{A}-\mathbf{B}$
- $-(-\mathbf{A})=\mathbf{A}$
- $\mathbf{A}*2=\mathbf{A}\ll 1$
- $(\mathbf{A}+\mathbf{B})-\mathbf{C}=\mathbf{A}+(\mathbf{B}-\mathbf{C})$
- $(\mathbf{A}+\mathbf{B})+\mathbf{C}=\mathbf{A}+(\mathbf{B}+\mathbf{C})$
- if $\mathbf{A}\geq\mathbf{B}$ then $(\mathbf{A}+1)>\mathbf{B}$
- if $\mathbf{A}\leq\mathbf{B}$ then $(\mathbf{A}-1)<\mathbf{B}$
- $(\mathbf{A}>\mathbf{A})=\mathbf{false}$
- $(\mathbf{A}\geq\mathbf{A})=\mathbf{true}$
- $\neg(\mathbf{A}>\mathbf{B}) \;=\; (\mathbf{A}\leq\mathbf{B})$
- $\neg(\mathbf{A}\leq\mathbf{B}) \;=\; (\mathbf{A}>\mathbf{B})$
- $(\mathbf{A}<\mathbf{B}) \;=\; (\mathbf{B}>\mathbf{A})$
- $(\mathbf{A}\leq\mathbf{B}) \;=\; (\mathbf{B}\geq\mathbf{A})$
- 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 $\mathbf{I} \neq \mathbf{J}$, then $\textsc{getarray}(\textsc{setarray}(\mathbf{S},\mathbf{A},\mathbf{J},\mathbf{V}),\mathbf{A},\mathbf{I}) = \textsc{getarray}(\mathbf{S},\mathbf{A},\mathbf{I})$

  [*Reading* $\mathbf{A}[\mathbf{I}]$ *after writing* $\mathbf{A}[\mathbf{J}]$ *(where* $\mathbf{I} \neq \mathbf{J}$*) is the same as reading before the write*]

- $\textsc{setarray}(\textsc{setarray}(\mathbf{S},\mathbf{A},\mathbf{I},\mathbf{V}_1),\mathbf{A},\mathbf{I},\mathbf{V}_2) = \textsc{setarray}(\mathbf{S},\mathbf{A},\mathbf{I},\mathbf{V}_2)$

  [*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 the same as reading before the write*]

- $\textsc{setfield}(\textsc{setfield}(\mathbf{S},\mathbf{O},\mathbf{F},\mathbf{V}_1),\mathbf{O},\mathbf{F},\mathbf{V}_2) = \textsc{setfield}(\mathbf{S},\mathbf{O},\mathbf{F},\mathbf{V}_2)$

  [*Writing* $\mathbf{O}.\mathbf{F} \leftarrow \mathbf{V}_1$ *then* $\mathbf{O}.\mathbf{F} \leftarrow \mathbf{V}_2$ *is the same as only writing* $\mathbf{V}_2$]

A.2 Domain-Specific Axioms

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*]

(Read Only) It is very common for certain Java methods to be read only. 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 read only*]

- $\rho_\sigma(\text{INVOKE}(\mathbf{S},\mathbf{L},\texttt{[int List.size()]},\mathbf{P})) = \mathbf{S}$

  [`List.size` *is read only*]

- $\rho_\sigma(\text{INVOKESTATIC}(\mathbf{S}, \texttt{[double Math.sqrt(double)]}, \mathbf{P})) = \mathbf{S}$

  [`Math.sqrt` *is read only*]

(Vector axioms) In our ray-tracer 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*]

- $\texttt{construct(}\mathbf{A},\mathbf{B},\mathbf{C}\texttt{).mX} = \mathbf{A}$

  $\texttt{construct(}\mathbf{A},\mathbf{B},\mathbf{C}\texttt{).mY} = \mathbf{B}$

  $\texttt{construct(}\mathbf{A},\mathbf{B},\mathbf{C}\texttt{).mZ} = \mathbf{C}$

  [*Accessing the field of constructed vector* (A,B,C) *is equal to appropriate parameter*]

- $\texttt{construct(}\mathbf{A},\mathbf{B},\mathbf{C}\texttt{).difference(construct(}\mathbf{D},\mathbf{E},\mathbf{F}\texttt{))} =$

  $\texttt{construct(}\mathbf{A}-\mathbf{D},\mathbf{B}-\mathbf{E},\mathbf{C}-\mathbf{F}\texttt{)}$

  [*The difference of vectors* (A,B,C) *and* (D,E,F) *equals* $(A-D, B-E, C-F)$]

- `construct(`$\mathbf{A},\mathbf{B},\mathbf{C}$`).dot(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* $A*D+B*E+C*F$]

- `construct(`$\mathbf{A},\mathbf{B},\mathbf{C}$`).length2()` $= \mathbf{A}*\mathbf{A}+\mathbf{B}*\mathbf{B}+\mathbf{C}*\mathbf{C}$

  [*The squared length of vector* (A,B,C) *equals* $A^2+B^2+C^2$]

- `construct(`$\mathbf{A},\mathbf{B},\mathbf{C}$`).negative()` $=$ `construct(`$-\mathbf{A},-\mathbf{B},-\mathbf{C}$`)`

  [*The negation of vector* (A,B,C) *equals* $(-A,-B,-C)$]

- `construct(`$\mathbf{A},\mathbf{B},\mathbf{C}$`).scaled(`$\mathbf{D}$`)` $=$ `construct(`$\mathbf{A}*\mathbf{D},\mathbf{B}*\mathbf{D},\mathbf{C}*\mathbf{D}$`)`

  [*Scaling vector* (A,B,C) *by* D *equals* $(A*D,B*D,C*D)$]

- `construct(`$\mathbf{A},\mathbf{B},\mathbf{C}$`).sum(construct(`$\mathbf{D},\mathbf{E},\mathbf{F}$`))` $=$
  `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* $(A+D,B+E,C+F)$]

- `getZero().mX` $=$ `getZero().mY` $=$ `getZero().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 wrappers:

  $\mathbf{A}$`.plus(`$\mathbf{B}$`).getValue()` $=$ $\mathbf{A}$`.getValue()` $+$ $\mathbf{B}$`.getValue()`

  [*Where* `plus` *returns a new integer wrapper, and* `getValue` *returns the wrapped value*]

- Axiom about redundant method calls when using `java.util.List`:

  `l.contains(o)` $=$ `l.indexOf(o) >= 0`

[*A list contains an item iff the item has a valid index*]

(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 for removing a common yet inefficient snippet of code and replacing it with a more efficient library implementation.

- Body of selection sort replaced with `Arrays.sort(int[])`:

```
length = A.length;
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];                    = Arrays.sort(A);
      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 $\mathit{isSorted}(\rho_\sigma(I), \mathbf{A}) = \mathbf{true}$

  [*If you call* `sort` *on an array A, then A is sorted in the subsequent heap*]

- if $\mathit{isSorted}(\mathbf{S}, \mathbf{A}) = \mathbf{true}$, 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*]

Acknowledgements

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*, volume 7, issue 1, 2011. The dissertation author was the primary investigator and author of this paper.

Bibliography

[1] Jiří Adámek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. John Wiley & Sons, 1990.

[2] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES*, 2004.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *POPL*, 1988.

[4] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1991.

[5] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *JFP*, 7(5):515–540, 1997.

[6] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[7] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[8] John C. Baez and Mike Stay. Physics, topology, logic and computation: A Rosetta stone. http://arxiv.org/abs/0903.0340, 2009.

[9] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[10] Pete Becker. C++0x draft standard, 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf, page 130.

[11] Ralph Bergmann. Explanation-based learning for the automated reuse of programs. In *CompEuro*, 1992.

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[13] 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.

[14] 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.

[15] Cliff Click. Global code motion/global value numbering. In *PLDI*, 1995.

[16] Keith D. Cooper Cliff Click. Combining analyses, combining optimizations. *TOPLAS*, 17(2):181–196, 1995.

[17] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES*, 1999.

[18] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

[19] Pierre-Louis Curien. The joy of string diagrams. In *CSL*, 2008.

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. In *POPL*, 1989.

[21] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[22] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Conference on LISP and Functional Programming*, 1994.

[23] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[24] Alin Deutsch. Author of [25]. Personal communication, July 2009.

[25] Alin Deutsch, Alan Nash, and Jeff Remmel. The chase revisited. In *PODS*, 2008.

[26] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. *Machine Learning*, 9(1):23–55, 1992.

[27] Edsgar W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[28] Niklas Eén and Niklas Sörensson. MiniSat: A SAT solver with conflict-clause minimization. In *8th International Conference on Theory and Application of Satisfiability Testing (SAT)*, 2005.

[29] Thomas Ellman. Generalizing logic circuit designs by analyzing proofs of correctness. In *IJCAI*, 1985.

[30] Thomas Ellman. Explanation-based learning: a survey of programs and perspectives. *ACM Computing Surveys*, 21(2):163–221, 1989.

[31] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[32] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[33] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG – fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.

[34] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC*, 1983.

[35] Milind Gandhe and G. Venkatesh. Improving prolog performance by inductive proof generalizations. In *Knowledge Based Computer Systems*, 1990.

[36] Joseph C. Giarratano and Gary D. Riley. *Expert Systems – Principles and Programming*. PWS Publishing Company, 1993.

[37] Torbjorn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *PLDI*, 1992.

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

[39] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order

languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

[40] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *POPL*, 1994.

[41] Paul Havlak. Construction of thinned gated single-assignment form. In *Workshop on Languages and Compilers for Parallel Computing*, 1993.

[42] Chris Heunen and Bart Jacobs. Arrows, like monads, are monoids. *Theoretical Computer Science*, 158:219–236, 2006.

[43] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[44] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[45] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *PLDI*, 2002.

[46] Linda Torczon Keith D. Cooper, Devika Subramanian. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, pages 7–22, 2002.

[47] Andrew Kennedy. Compiling with continuations, continued. In *ICFP*, 2007.

[48] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.

[49] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[50] Saunders M. Lane. Natural associativity and commutativity. *Rice University Studies*, 49:28–46, 1963.

[51] Daan Leijen. A type directed translation of MLF to System F. In *ICFP*, 2007.

[52] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *POPL*, 2002.

[53] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[54] Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS*, 1987.

[55] Amy McGovern, J. Eliot B. Moss, and Andrew G. Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine Learning, Special Issue on Reinforcement Learning*, 49(2/3):141–160, 2002.

[56] Robin Millner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 7(3):348–375, 1978.

[57] Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, 1989.

[58] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

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

[60] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2):245–257, 1979.

[61] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[62] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, 1990.

[63] Keshav Pengali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodhgill. Dependence flow graphs: an algebraic approach to program dependencies. In *POPL*, 1991.

[64] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, 1998.

[65] A.J̃ohn Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, 1997.

[66] A.J̃ohn Power and Hayo Thielecke. Environments, continuation semantics and indexed categories. In *TACS*, 1997.

[67] Willard Van Orman Quine. *Word and Object*. Simon and Schuster, 1964.

[68] John Regehr. C compilers disprove Fermats last theorem, April 2010. http://blog.regehr.org/archives/140.

[69] Hanan Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, 1978.

[70] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:61–96, 2006.

[71] Bernhard Steffen, Jens Knoop, and Oliver Ruthing. The value flow graph: A program representation for optimal program transformations. In *ESOP*, 1990.

[72] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, 2005.

[73] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *PLDI*, 2003.

[74] Michael Stepp. *Equality Saturation: Engineering Challenges and Applications*. PhD thesis, University of California, San Diego, 2011.

[75] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for LLVM. In *CAV*, 2011.

[76] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbolic Computation*, 13(1-2):11–49, 2000.

[77] Ross Tate, Juan Chen, and Chris Hawblitzel. Inferable object-oriented typed assembly language. In *PLDI*, 2010.

[78] Ross Tate, Alan Leung, and Sorin Lerner. Taming wildcards in Java's type system. In *PLDI*, 2011.

[79] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimization from proofs. In *POPL*, 2010.

[80] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL*, 2009.

[81] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation:

a new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[82] Tachio Terauchi and Alex Aiken. Witnessing side-effects. In *ICFP*, 2005.

[83] Steven W. K. Tjiang and John L. Hennessy. Sharlit – A tool for building optimizers. In *PLDI*, 1992.

[84] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.

[85] Peng Tu and David Padua. Efficient building and placing of gating functions. In *PLDI*, 1995.

[86] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java optimization framework. In *CASCON*, 1999.

[87] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *TOPLAS*, 24(4), 2002.

[88] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, 1998.

[89] Philip Wadler. Comprehending monads. In *LFP*, 1990.

[90] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, 1995.

[91] Philip Wadler. The marriage of effects and monads. In *ICFP*, 1998.

[92] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *POPL*, 1994.

[93] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *PPOPP*, 1990.

[94] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *TOPLAS*, 19(6):1053–1084, 1997.

[95] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *PLDI*, 2008.

[96] 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, 2003.