Logical Methods in Computer Science
Vol. 7 (1:10) 2011, pp. 1–37
www.lmcs-online.org

Submitted Oct. 12, 2009
Published Mar. 28, 2011

EQUALITY SATURATION: A NEW APPROACH TO OPTIMIZATION

ROSS TATE, MICHAEL STEPP, ZACHARY TATLOCK, AND SORIN LERNER

Department of Computer Science and Engineering, University of California, San Diego
e-mail address: {rtate,mstepp,ztatlock,lerner}@cs.ucsd.edu

Abstract. Optimizations in a traditional compiler are applied sequentially, with each optimization destructively modifying the program to produce a transformed program that is then passed to the next optimization. We present a new approach for structuring the optimization phase of a compiler. In our approach, optimizations take the form of equality analyses that add equality information to a common intermediate representation. The optimizer works by repeatedly applying these analyses to infer equivalences between program fragments, thus saturating the intermediate representation with equalities. Once saturated, the intermediate representation encodes multiple optimized versions of the input program. At this point, a profitability heuristic picks the final optimized program from the various programs represented in the saturated representation. Our proposed way of structuring optimizers has a variety of benefits over previous approaches: our approach obviates the need to worry about optimization ordering, enables the use of a global optimization heuristic that selects among fully optimized programs, and can be used to perform translation validation, even on compilers other than our own. We present our approach, formalize it, and describe our choice of intermediate representation. We also present experimental results showing that our approach is practical in terms of time and space overhead, is effective at discovering intricate optimization opportunities, and is effective at performing translation validation for a realistic optimizer.

Introduction

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

1998 ACM Subject Classification: D.3.4.

Key words and phrases: Compiler Optimization, Equality Reasoning, Intermediate Representation.

An earlier version of this work appeared at the 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2009).

Supported in part by NSF CAREER grant CCF-0644306.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-7 (1:10) 2011

make their decisions one optimization at a time, and so it is difficult for these heuristics to account for the effect of future transformations.

In this paper, we present a new approach for structuring optimizers that addresses the above limitations of the traditional approach, and also has a variety of other benefits. Our approach consists of computing a set of optimized versions of the input program and then selecting the best candidate from this set. The set of candidate optimized programs is computed by repeatedly inferring equivalences between program fragments, thus allowing us to represent the effect of many possible optimizations at once. This, in turn, enables the compiler to delay the decision of whether or not an optimization is profitable until it observes the full ramifications of that decision. Although related ideas have been explored in the context of super-optimizers, as Section 10 on related work will point out, super-optimizers typically operate on straight-line code, whereas our approach is meant as a general-purpose compilation paradigm that can optimize complicated control flow structures.

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

The simplest form of equality analysis looks for ways to instantiate equality axioms like $a * 0 = 0$, or $a * 4 = a << 2$. However, our approach also supports arbitrarily complicated forms of equality analyses, such as inlining, 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 a new IR that compactly represents equality information, and as a result can simultaneously store multiple optimized versions of the input program. After a program is converted into our IR, we repeatedly apply equality analyses to infer new equalities until no more equalities can be inferred, a process known as saturation. Once saturated with equalities, our IR compactly represents the various possible ways of computing the values from the original program modulo the given set of equality analyses (and modulo some bound in the case where applying equality analyses leads to unbounded expansion).

Our approach 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 does not matter. The first benefit of our approach is that it removes the need to think about optimization ordering. When applying optimizations sequentially, ordering is a problem because one optimization, say A, may perform some transformation that will irrevocably prevent another optimization, say B, from triggering, when in fact running B first would have produced the better outcome. This so-called *phase ordering problem* is ubiquitous in compiler design. In our approach, however, the compiler writer does not need to worry about ordering, because optimizations do not destructively update the program – they simply add equality information. Therefore, after an optimization A

is applied, the original program is still represented (along with the transformed program), and so any optimization B that could have been applied before A is still applicable after A. Thus, there is no way that applying an optimization A can irrevocably prevent another optimization B from applying, and so there is no way that applying optimizations will lead the search astray. As a result, compiler writers who use our approach do not need to worry about the order in which optimizations run. Better yet, because optimizations are allowed to freely interact during equality saturation, without any consideration for ordering, our approach can discover intricate optimization opportunities that compiler writers may not have anticipated, and hence would not have implemented in a general purpose compiler.

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

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, and so it has a global view of all the transformations that were tried and what programs they generated.

There are many ways to implement this global profitability heuristic, and in our prototype compiler we have chosen to implement it using a Pseudo-Boolean solver (a form of Integer Linear Programming 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.

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

Contributions. The contributions of this paper can therefore be summarized as follows:

- We present a new approach for structuring optimizers. In our approach optimizations add equality information to a common IR that simultaneously represents multiple optimized versions of the input program. Our approach obviates the need to worry about optimization ordering, it enables the use of a global optimization heuristic (such as a Pseudo-Boolean solver), and it can be used to perform translation validation for any compiler. Sections 1 and 2 present an overview of our approach and its capabilities, Section 4 makes our approach formal, and Sections 5 through 7 describe the new IR that allows our approach to be effective.
- We have instantiated our approach in a new Java bytecode optimizer called Peggy (Section 8). Peggy uses our approach not only to optimize Java methods, but also to perform translation validation for existing compilers. Our experimental results (Section 9) show that our approach (1) is practical both in terms of time and space overhead, (2) is effective at discovering both simple and intricate optimization opportunities and (3) is effective at performing translation validation for a realistic optimizer – Peggy is able to validate 98% of the runs of the Soot optimizer [52], and within the remaining 2% it uncovered a bug in Soot.

1. Overview

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

To make equality reasoning *effective*, our IR needs to support the same kind of basic reasoning that one would expect from simple equality axioms like $a * (b + c) = a * b + a * c$, but with more complicated computations such as branches and loops. We have designed a representation for computations called Program Expression Graphs (PEGs) that meets these requirements. Similar to the *gated SSA* representation [51, 31], PEGs are *referentially transparent*, which intuitively means that the value of an expression depends only on the value of its constituent expressions, without any side-effects. As has been observed previously in many contexts, referential transparency makes equality reasoning simple and effective. However, unlike previous SSA-based representations, PEGs are also *complete*, which means that there is no need to maintain any additional representation such as a control flow graph (CFG). Completeness makes it easy to use equality for performing transformations: if two PEG nodes are equal, then we can pick either one to create a program that computes the same result, without worrying about the implications on any underlying representation.

In addition to being effective, equality reasoning in our IR must be *efficient*. The main challenge is that each added equality can potentially double the number of represented programs, thus making the number of represented programs exponential in the worst case. To address this challenge, we record equality information of PEG nodes by simply merging PEG nodes into equivalence classes. We call the resulting equivalence graph an 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

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

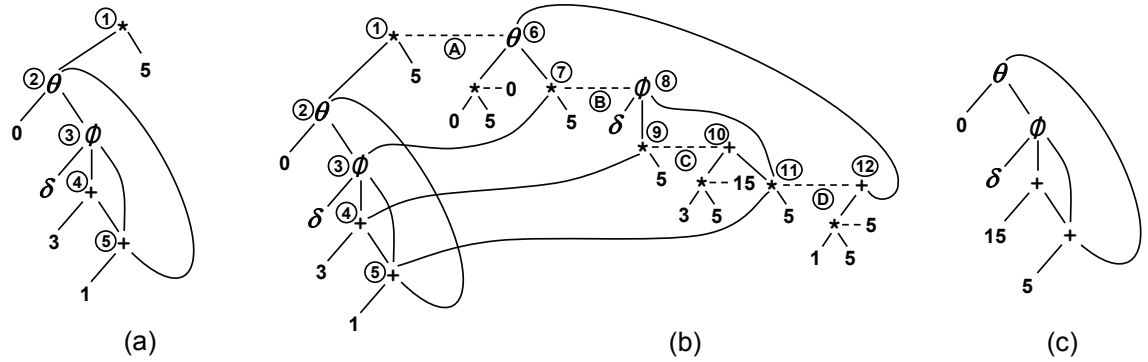

Figure 2: Loop-induction-variable Strength Reduction using PEGs: (a) shows the original PEG, (b) shows the E-PEG that our engine produces from the original PEG and (c) shows the optimized PEG, which results by choosing nodes 6, 8, 10, and 12 from (b).

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 Section 10.

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

As an example, consider the code snippet from Figure 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 1(b).

1.1. **Program Expression Graphs.** A Program Expression Graph (PEG) is a graph containing: (1) operator nodes, for example "plus", "minus", or any of our built-in nodes for representing conditionals and loops, and (2) "dataflow" edges that specify where operator nodes get their arguments from. As an example, consider the "use" statement in Figure 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(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 [41, 42, 21] 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 [19, 5]. 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 [51, 31]: they take an additional parameter (the first left-most one) which is used to select between the second and the third parameter. As a result, our ϕ nodes are executable by themselves, and so there is no need to explicitly encode a control-flow join. Our example doesn't use the branch condition in an interesting way, and so we just let δ represent the PEG sub-graph that computes the branch condition. Furthermore, since this PEG represents the value of `i` *inside* the loop, it does not contain any operators to describe the `while`-condition, since this information is only relevant for computing the value of `i` after the loop has terminated. We show how to express the value of variables after a loop in Section 2.

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, the nodes labeled 3 through 5 compute this sequence of remaining values in terms of the sequence produced by the θ node. In particular, nodes 3, 4 and 5 have been implicitly lifted to operate on this sequence. The fact that a single θ node represents the entire sequence of values that a loop produces allows us to represent that two loops compute the same sequence of values with a single equality between two θ nodes.

PEGs are well-suited for equality reasoning because all PEG operators, even those for branches and loops, are mathematical functions with no side effects. As a result, PEGs are *referentially transparent*, which allows us to perform the same kind of equality reasoning that one is familiar with from mathematics. Though PEGs are related to functional programs, in our work we have used PEGs to represent intra-procedural imperative code with branches and looping constructs. Furthermore, even though all PEG operators are pure, PEGs can

still represent programs with state by using heap summary nodes: stateful operations, such as heap reads and writes, can take a heap as an argument and return a new heap. This functional representation of stateful programs allows our Peggy compiler to use PEGs to reason about Java programs. The heap summary node can also be used to encode method/function calls in an intra-procedural setting by simply threading the heap summary node through special nodes representing method/function calls. There is however one big challenge with heap summary nodes, which we have not yet fully addressed yet. Although in the PEG domain, heap summary nodes can be reasoned about as if the heap can be duplicated, when a PEG is converted back to a CFG, heap summary nodes must obey a linear-typing discipline. We have developed a simple constraint solving technique for finding a linearization of heap operations in a PEG, but this technique is not complete (as in, even if there is a linearization, we are not guaranteed to find it). In our Peggy implementation, after optimizations have been applied, this incompleteness affects 3% of Java methods (in which case we do not optimize the method). Section 8 explains in more detail how we represent several features of Java programs in PEGs (including the heap and method calls) and what the issues are with our linearization incompleteness. We also present some ideas on how to address this incompleteness in future work.

1.2. **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(b) shows the E-PEG that our engine produces from the PEG of Figure 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 application, our trigger and callback mechanism is much more general, and has allowed us to implement equality analyses such as inlining, tail-recursion elimination, and constant folding.

The following three axioms are the equality analyses required to perform loop-induction-variable strength reduction. They state that multiplication distributes over addition, θ, and

ϕ:

$$(a + b) * m = a * m + b * m \tag{1.1}$$

$$\theta(a, b) * m = \theta(a * m, b * m) \tag{1.2}$$

$$\phi(a, b, c) * m = \phi(a, b * m, c * m) \tag{1.3}$$

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 Section 9 will describe in more detail, our experiments show that 84% of methods can be completely saturated, without any bounds being imposed.

Figure 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 (1.2) adds equality edge A, axiom (1.3) edge B, axiom (1.1) edge C, and axiom (1.1) 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 (1.2) adds edge A, it also adds node 7, which then enables axiom (1.3). Thus, equality analyses essentially communicate with each other by propagating equalities through the E-PEG. Furthermore, note that the instantiation of axiom (1.1) 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(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(c), which corresponds exactly to performing loop-induction-variable strength reduction in Figure 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 Programming 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, the Pseudo-Boolean solver picks the nodes described above. Section 8.3 describes our technique for selecting nodes in more detail.

1.3. **Benefits of our approach.**

Optimization order does not matter. 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 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 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 [60] have shown experimentally that enabling and disabling interactions between optimizations occur frequently in practice, and furthermore that the patterns of interaction vary not only from program to program, but also within a single program. Thus, no one order is best across all compilation.

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

At high levels of optimizations, some compilers may even run optimizations in a loop until no more changes can be made. Even so, if the compiler picks the wrong optimization to start with, then no matter what optimizations are applied later, in any order, any number of times, the compiler will not be able to reverse the disabling consequences of the first optimization.

In our approach, the compiler writer does not need to worry about the order in which optimizations are applied. The previous peephole optimization would be expressed as the axiom `i * 5 = i << 2 + i`. However, unlike in a traditional compilation system, applying this axiom in our approach does not remove the original program from the representation — it only adds information — and so it cannot disable other optimizations. Therefore, the code from Figure 1(b) would still be discovered, even if the peephole optimization was run first. In essence, our approach is able to simultaneously explore all possible sequences of optimizations, while sharing work that is common across the various sequences.

In addition to reducing the burden on compiler writers, removing the need to think about optimization ordering has two additional benefits. First, because optimizations interact freely with no regard to order, our approach often ends up combining optimizations in unanticipated ways, leading to surprisingly complicated optimizations given how simple our equality analyses are — Section 2 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, and 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 effect of future optimizations. For example, consider inlining. Although it is straightforward to estimate the *direct cost* of

inlining (the code-size increase) and the *direct benefit* of inlining (the savings from removing the call overhead), it is far more difficult to estimate the potentially larger *indirect benefit*, namely the additional optimization opportunities that inlining exposes.

To see how inlining would affect our running example, consider again the code from Figure 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* [20], where the compiler simulates the inlining transformation, along with the effect of subsequent optimizations, in order to decide whether or not to actually inline. However, in the face of multiple inlining decisions (or more generally multiple optimization decisions), there can be exponentially many possible outcomes, each one of which has to be compiled separately.

In our approach, on the other hand, inlining simply adds an equality to the 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.

Translation Validation. Unlike traditional compilation frameworks, our approach can be used not only to optimize programs, but also to establish equivalences between programs. In particular, if we convert two programs into an E-PEG, and then saturate it with equalities, then we can conclude that the two programs are equivalent if they belong to the same equivalence class in the saturated E-PEG. In this way, our approach can be used to perform translation validation for any compiler (not just our own), by checking that each function in the input program is equivalent to the corresponding optimized function in the output program.

For example, our approach would be able to show that the two program fragments from Figure 1 are equivalent. Furthermore, it would also be able to validate a compilation run in which `i * 5 = i << 2 + i` was applied first to Figure 1(a). This shows that we are able to perform translation validation regardless of what optimized program our own profitability heuristic would choose.

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

2. Reasoning about loops

This section 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.

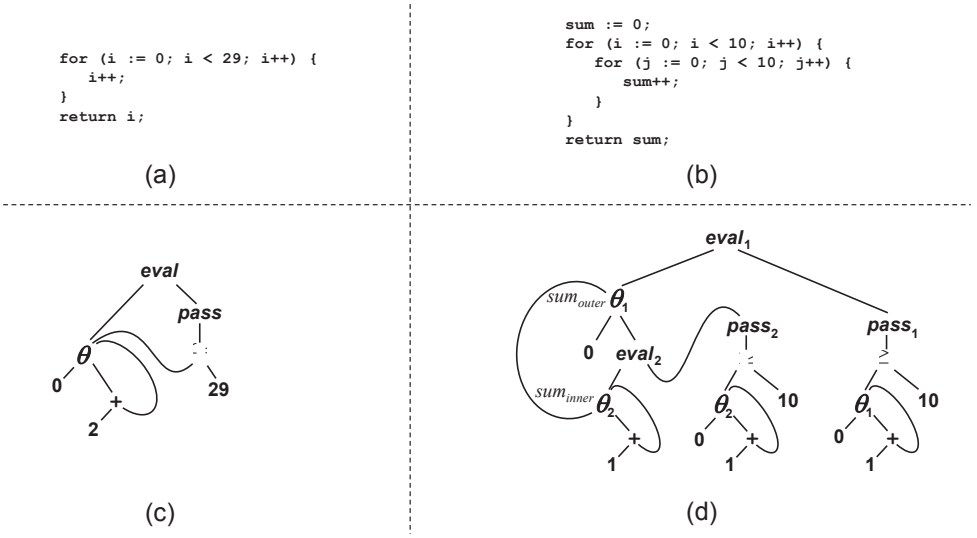

Figure 3: Two loops and their PEG representations.

We start in Sections 2.1 and 2.2 by describing all PEG constructs used to represent loops. We then show in Section 2.3 how our approach can perform an inter-loop strength reduction optimization.

2.1. **Single loop.** Consider the simple loop from Figure 3(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(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 taken on by the boolean expression $\texttt{i} \geq 29$ throughout the loop. This sequence is then sent to *pass*, which in this case produces the value 15, since the 15^{th} value (counting from 0) of `i` in the loop (which is 30) is the first one to make $\texttt{i} \geq 29$ true. The *eval* node then selects the 15^{th} element of the sequence produced by the θ node, which is 30. In our previous example from Section 1, 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.

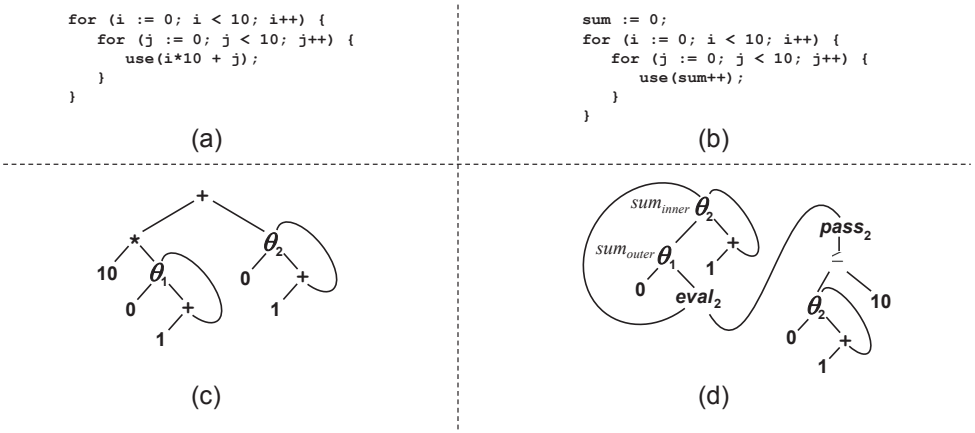

Figure 4: Two equivalent loops and their PEG representations. The PEGs for the expressions inside the `use` statements in (a) and (b) are shown in (c) and (d), respectively.

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 3.3) we use ϕ nodes and other special-purpose nodes as the second argument. Furthermore, Section 8.4 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.

2.2. **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(b), which has two nested loops. The PEG for this code snippet is shown in Figure 3(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(c), as is the value of `sum` after the outer loop terminates.

2.3. **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 4(a) and (b). The code in Figure 4(b) is equivalent to the code in Figure 4(a), but it is faster

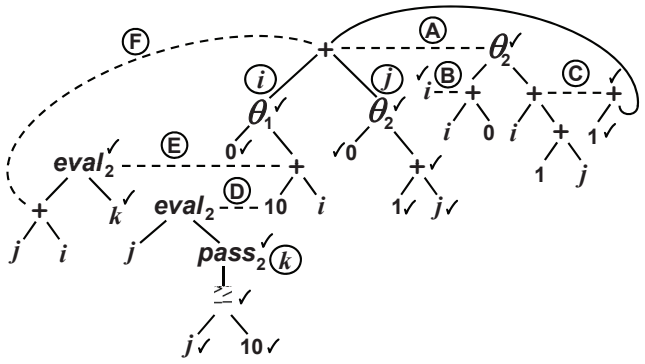

Figure 5: E-PEG that results from running the saturation engine on the PEG from Figure 4(c). By picking the nodes that are checkmarked, we get the PEG from Figure 4(d). To make the graph more readable, we sometimes label nodes, and then connect an edge directly to a label name, rather than connecting 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.

because `sum++` is cheaper than `i * 10 + j`. We show how our approach can transform the code in Figure 4(a) to the code in Figure 4(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 4(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 4(d). This is the PEG for the `sum++` expression from Figure 4(b). Note that the code snippet in Figure 4(b) is the same as Figure 3(b), and as a result Figure 4(d) is just the sum_{inner} node from Figure 3(d), along with its children. To summarize, in terms of PEGs, our optimization will replace the `+` node from Figure 4(c), which occurs in some larger PEG context, with the sum_{inner} node from Figure 4(d). The surrounding PEG context, which we do not show, remains unchanged.

Figure 5 shows the saturated E-PEG that results from running the saturation engine on the PEG from Figure 4(c). The checkmarks indicate which nodes will eventually be selected – they can be ignored for now. In drawing Figure 5, we have already performed loop-induction variable strength reduction on the left child of the topmost + from Figure 4(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 1.2.

Figure 5 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 5 to highlight the nodes that Peggy would select using its Pseudo-Boolean profitability heuristic. These nodes constitute exactly the PEG from Figure 4(d), meaning that Peggy optimizes the code in Figure 4(a) to the one in Figure 4(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 super-optimizer-based techniques.

3. Local changes have non-local effects

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

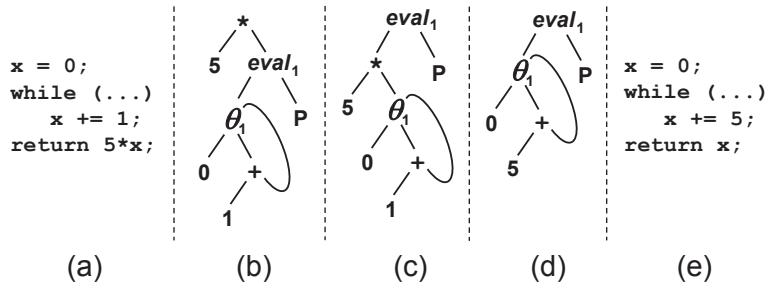

Figure 6: An example of loop-based code motion from simple axiom applications; (a) the original source code, (b) the original PEG, (c) the PEG after distributing $*$ through $eval_1$, (d) the PEG after performing loop-induction-variable strength reduction, (e) the resulting source code.

3.1. **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 6. Part (a) shows the source code for a loop where the counter variable is multiplied by 5 at the end, and part (e) shows equivalent code where the multiplication is removed and the increment has been changed to 5. Essentially, this optimization moves the ($*5$) from the end of the loop and applies it to the increment and the initial value instead. This constitutes code motion into a loop, and is a non-local transformation in the CFG.

Peggy can perform this optimization using local axiom applications, without requiring any additional non-local reasoning. Figure 6(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 Section 1). 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.

3.2. **Restructuring the CFG.** In addition to allowing non-local optimizations, small changes in the PEG can cause large changes in the program's CFG. Consider the program in Figure 7. Parts (a) and (f) show two CFGs that are equivalent but have very different structure. Peggy can use several local axiom applications to achieve this same restructuring. Figure 7(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

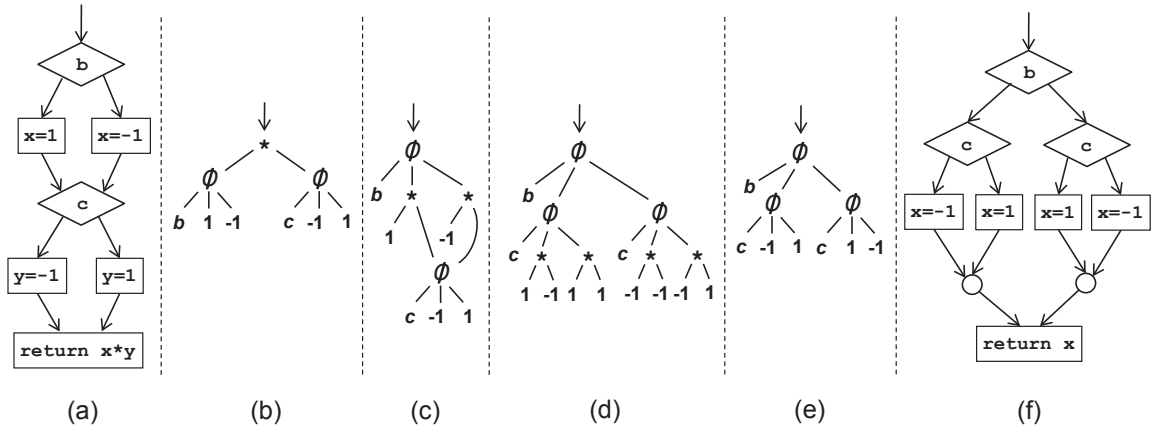

Figure 7: An example of how local changes in the PEG can cause large changes in the CFG: (a) the original CFG, (b) the original PEG, (c) the PEG after distributing $*$ through the left-hand ϕ, (d) the PEG after distributing $*$ through the bottom ϕ, (e) the PEG after constant folding, (f) the resulting CFG.

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 effects on the program.

3.3. **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 8(a). We want to perform a loop peeling on this code, which will result in the code shown in Figure 8(i). This can be done through axiom application through the following steps, depicted in Figure 8 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.

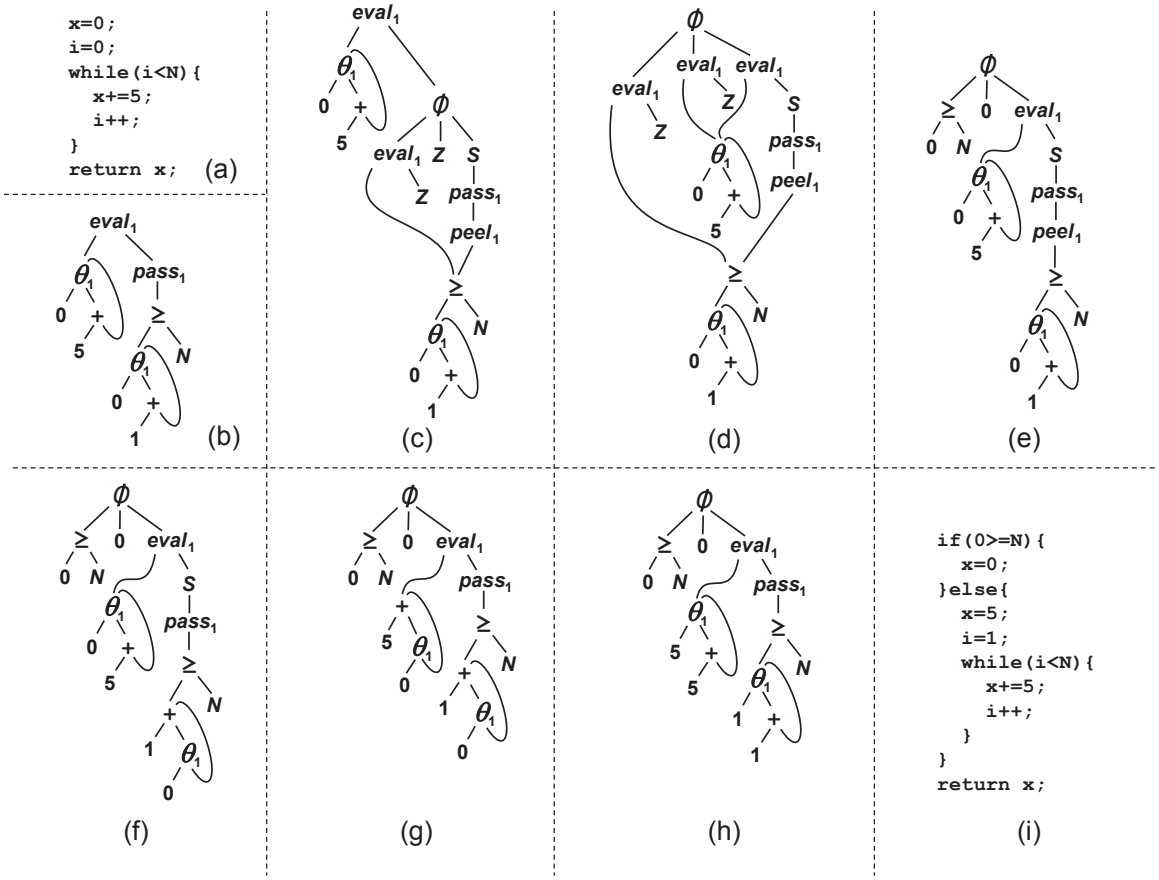

Figure 8: An example of axiom-based loop peeling: (a) the original loop, (b) the PEG for part (a), (c)-(h) intermediate steps of the optimization, (i) the final peeled loop, which is equivalent to (h).

The third step, shown in part (e), is the result of propagating 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 that $eval_1(C, \mathrm{Z}) = C$ for any constant or parameter C, which is why $eval_1(\mathtt{N}, \mathrm{Z}) = \mathtt{N}$.

The fourth step, shown in part (f), involves propagating 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 $peel$ operator meets a θ, it simplifies with the axiom $peel_1(\theta_1(A, B)) = B$. Furthermore, we also use the axiom that $peel_1(C) = C$ 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 θ's and doing constant folding afterward, to yield the PEG in part (h). This PEG is equivalent to the final peeled source code in part (i).

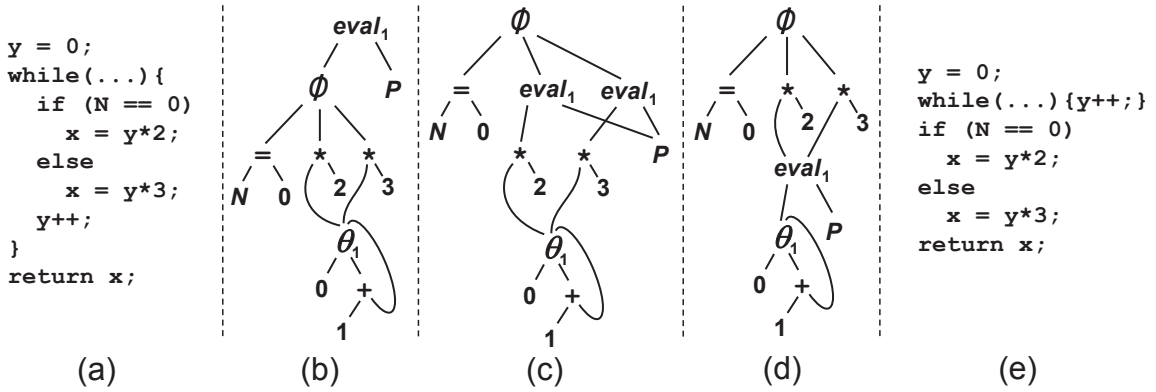

Figure 9: An example of branch hoisting: (a) the original program, (b) the PEG for part (a), (c) the PEG after distributing *eval* through ϕ, (d) the PEG after distributing *eval* through $*$, (e) the code resulting from (d).

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

The resulting PEG shown in Figure 8(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.

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

Consider the code in Figure 9(a). We assume that `N` is a parameter or a variable initialized elsewhere, and is clearly not altered inside the loop. Hence the condition on the if-statement is loop-invariant. Also we see that `x` is never read inside the loop, so the value it holds at the end of the loop can be expressed entirely in terms of the final values of the other variables (i.e. `y`). Hence, this code is equivalent to the code seen in part (e), where the branch is moved outside the loop and `x` is assigned once, using only the final value of `y`.

Our saturation engine can perform this optimization using simple axioms, starting with the PEG shown in part (b) corresponding to the code in part (a). In part (b), we display the

pass condition as P, since we never need to reason about it. Parts (c) and (d) depict the relevant intermediate steps in the optimization. Part (c) results from distributing the *eval* operator through the ϕ operator using the axiom $op(\phi(A, B, C), D) = \phi(A, op(B, D), op(C, D))$ with $op = eval_1$. 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.

3.5. **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 Section 2, the ideal output would be a single loop that increments the `sum` variable. One option for doing this kind of optimization is to add build-in axioms for fusing these kinds of loops together into one. Another optimization that we have not fully explored is loop unrolling. By adding a few additional higher-level operators to our PEGs, we were able to perform loop unrolling on paper using just equational reasoning. Furthermore, using similar higher-level operators, we believe that we could also perform loop interchange (which changes a loop `for i in R1, for j in R2` into `for j in R2 for i in R1`). However, both of these optimizations do require adding new operators to the PEG, which would require carefully formalizing their semantics and axioms that govern them. Finally, these more sophisticated loop optimizations would also require a more sophisticated cost model. In particular, because our current cost model does not take into account loop bounds (only loop depth), it has only a coarse approximation of the number of times a loop executes. As a result, it would assign the same cost to the loop before and after interchange, and it would assign a higher cost to an unrolled loop than the original. For our cost model to see these optimizations as profitable, we would have to update it with more precise information about loop bounds, and a more precise modeling of various architectural effects like caching and scheduling. We leave all of these explorations to future work.

4. Formalization of our Approach

Having given an intuition of how our approach works through examples, we now move to a formal description. Figure 10 shows the Optimize function, which 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.

```
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 10: Optimization phase in our approach. We assume a global set A of equality analyses to be run.

An instantiation 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 sections will show how we instantiate these three components in our Peggy compiler.

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 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. Note also that at this point in the presentation, when saying $ir_1 \xrightarrow{a} ir_2$ we keep ir_1 and ir_2 as abstract representations of an E-PEG (which includes a PEG graph and a set of equalities over PEG nodes). Later in Section 5 we will formally define what PEGs and E-PEGs are.

We define a partial order $\sqsubseteq$ on IRs, based on the nodes and equalities they encode: $ir_1 \sqsubseteq ir_2$ 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. Immediately from this definition, we get:

$$(ir_1 \xrightarrow{a} ir_2) \Rightarrow ir_1 \sqsubseteq ir_2 \tag{4.1}$$

We define an equality analysis a to be monotonic iff:

$$(ir_1 \sqsubseteq ir_2) \wedge (ir_1 \xrightarrow{a} ir_1') \Rightarrow \exists ir_2'.[(ir_2 \xrightarrow{a} ir_2') \wedge (ir_1' \sqsubseteq ir_2')] \tag{4.2}$$

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 (4.1) and (4.2) immediately imply the following property:

$$(ir_1 \xrightarrow{a} ir_1') \wedge (ir_1 \xrightarrow{b} ir_2) \Rightarrow \exists ir_2'.[(ir_2 \xrightarrow{a} ir_2') \wedge (ir_1' \sqsubseteq ir_2')] \tag{4.3}$$

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

We now define $ir_1 \rightarrow ir_2$ as:

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

The $\rightarrow$ relation formalizes one step taken by the saturation engine. We also define $\rightarrow^*$ to be the reflexive transitive closure of $\rightarrow$. The $\rightarrow^*$ 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 $ir_1 \rightarrow^* ir_2$ and there is no ir_3 such that $ir_2 \rightarrow ir_3$. 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. (4.4)

In essence, property (4.4) 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 (4.4) 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 E-PEG and produce new nodes and new equalities. 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 (4.3) 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).

5. PEGs and E-PEGs

The first step in instantiating our approach from the previous section 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 5.1), then we present its benefits (Section 5.4), and finally we give a detailed description of how to translate from CFGs to our IR and back (Sections 6 and 7).

5.1. **Formalization of PEGs.** A PEG is a triple $\langle N, L, C\rangle$, where N is a set of nodes, $L : N \rightarrow F$ is a labeling that maps each node to a semantic function from a set of semantic functions F, and $C : N \rightarrow N^*$ 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 fails or does not terminate. Formally, for each type τ, we define $\tau_\perp = \tau \cup \{\perp\}$.

Second, values are loop-lifted, which means that instead of representing the value at a particular iteration, PEG nodes represent values for all iterations at the same time. Formally, we let $\mathcal{L}$ be a set of loop identifiers, with each $\ell \in \mathcal{L}$ 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 $\ell \in \mathcal{L}$ 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 5^{th} iteration and loop ℓ_2 is on the 3^{rd} iteration. We let $\mathbb{I} = \mathcal{L} \to \mathbb{N}$ 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 $\mathbf{i}[\ell \mapsto v]$ to denote a function that returns the same value as $\mathbf{i}$ on all inputs, except that it returns v 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 11 gives the definition of the primitive functions $\mathit{Prims} = \{\phi, \theta_\ell, \mathit{eval}_\ell, \mathit{pass}_\ell\}$. These functions are polymorphic in τ, in that they can be instantiated for various τ's, ranging from basic types like integers and strings to complicated types like the heap summary nodes that Peggy uses to represent Java objects. The definitions of eval_ℓ and pass_ℓ make use of the function $\mathit{monotonize}_\ell$, whose definition is given in Figure 11. The $\mathit{monotonize}_\ell$ function transforms a sequence so that, once an indexed value is undefined, all following indexed values are undefined. The $\mathit{monotonize}_\ell$ 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 $\perp$-lifted, and then loop-lifted version of *op*. Intuitively, the $\perp$-lifted version of an operator works like the original operator except that it returns $\perp$ if any of its inputs are $\perp$, 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.

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

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

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

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

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

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

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

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

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

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

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:

$$[\![n]\!] = L(n)([\![C(n)]\!]) \tag{5.1}$$

Equation 5.1 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 5.1 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 5.1 (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's 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 1. *If a PEG is well-formed, then for each node n in the PEG there is a unique semantic value $[\![n]\!]$ satisfying Equation 5.1.*

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

Evaluation Semantics. The meaning 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've 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 Section 7) can use a single loop to compute both the *pass* result and the *eval* result.

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 assume that a PEG $\langle N, L, C\rangle$ has 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 as $L : N \to F \cup P$, 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 11 to pass around a value context Σ 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 $\widetilde{c}$. We use $n[x_1 \mapsto c_1, \ldots, x_k \mapsto c_k]$ to denote $n[x_1 \mapsto c_1] \ldots [x_k \mapsto c_k]$. Figure 12 shows an example with parameter nodes and an example of the substitution notation.

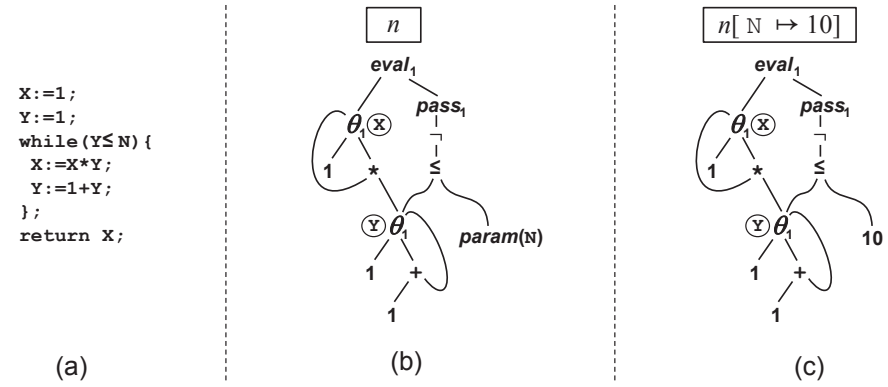

Figure 12: Example showing PEG with parameter nodes. (a) shows code for computing the factorial of N, where N is a parameter (b) shows the PEG with n being the value returned and (c) shows $n[\mathtt{N} \mapsto 10]$, which is now a PEG whose semantics is well-defined in our formalism.

5.2. **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 Section 8, our implementation strategy keeps track of these equivalence classes, rather than the set E.

5.3. **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 "don't care":

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

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 7.9 to optimize the PEG-to-imperative-code translation, and in Section 8.3 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 5.2 (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 $invariant_\ell$ for all ℓ. Also, since no rule restricts $pass_\ell$ nodes, such nodes will always be $invariant_\ell$. This syntactic definition of $invariant_\ell$ is best computed using an optimistic dataflow analysis.

Having defined $invariant_\ell$, the following built-in axioms hold if $invariant_\ell(A)$ holds:

$$\begin{array}{rcll} eval_\ell(A, \bullet) & = & A & \\ x & = & A & \text{where } x = \theta_\ell(A, x) \\ peel_\ell(A) & = & A & \end{array}$$

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 Sections 1 through 3. Appendix A contains the much longer list of all axioms that we have used in Peggy.

5.4. **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, 36, 46]. In our PEG representation, referential transparency can be formalized as follows:

$$\forall (n, n') \in N^2 \; . \; \left(\begin{array}{l} L(n) = L(n') \wedge \\ [\![C(n)]\!] = [\![C(n')]\!] \end{array} \right) \Rightarrow [\![n]\!] = [\![n']\!]$$

This property follows from the definition in Equation (5.1), 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.

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

Figure 13: Grammar for SIMPLE programs

6. Representing Imperative Code as PEGs

In this section 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 6.1) and then present ML-style functional pseudocode for converting any program written in this language into a PEG (Section 6.2). Next, we present a formal account of the conversion process using type-directed translation rules in the style of [37] (Section 6.3). Finally, we outline a proof of the semantic preservation of the translation (Section 6.4). Our technical report [50] shows how to extend the techniques in this section for converting SIMPLE programs to PEGs in order to convert more complex control flow graphs to PEGs.

6.1. **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 13. 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 (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 type-checking rules for SIMPLE programs are shown in Figure 14. There are three kinds of judgments: (1) judgment $\vdash p$ (where p is a program) states that p is well-typed; (2) judgment $\Gamma \vdash s : \Gamma'$ (where s is a statement) states that starting with context Γ, after s the context will be Γ'; (3) judgment $\Gamma \vdash e : \tau$ (where e is an expression) states that in type context Γ, expression e has type τ.

For program judgments, 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 judgments, 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.

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

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

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

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

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

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

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

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

Figure 14: Type-checking rules for SIMPLE programs

6.2. **Translating SIMPLE Programs to PEGs.** Here we use ML-style functional pseudo-code to describe the translation from SIMPLE programs to PEGs. Figure 15 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 10, then the ConvertToIR function from Figure 10 is implemented using a call to TranslateProg.

In the pseudo-code (and in all of Sections 6 and 7), 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 pseudo-code we want to avoid confusion between for example applying negation in the pseudo-code $\neg(\ldots)$, vs. creating a PEG node labeled with negation $\overline{\neg}(\ldots)$. For parameter nodes, there can't be any confusion because when we write $param(x)$, $param$ is actually not a function – instead $param(x)$ as a whole is label. Still, to be consistent in the notation, we use $\overline{param}(x)$ for constructing parameter 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 6.3). For our pseudo-code, we implement node contexts as an immutable map data structure that has the following operations defined on it

```
1: function TranslateProg(p : Prog) : N =
2:   let m = InitMap(p.params, λx. param(x))
3:   in TS(p.body, m, 0)(retvar)

4: function TS(s : Stmt, Ψ : map[V, N], ℓ : ℕ) : map[V, N] =
5:   match s with
6:     "s1; s2" ⇒ TS(s2, TS(s1, Ψ, ℓ), ℓ)
7:     "x := e" ⇒ Ψ[x ↦ TE(e, Ψ)]
8:     "if (e) {s1} else {s2}" ⇒ PHI(TE(e, Ψ), TS(s1, Ψ, ℓ), TS(s2, Ψ, ℓ))
9:     "while (e) {s}" ⇒
10:      let vars = Keys(Ψ)
11:      let Ψt = InitMap(vars, λv. TemporaryNode(v))
12:      let Ψ' = TS(s, Ψt, ℓ + 1)
13:      let Ψθ = THETAℓ+1(Ψ, Ψ')
14:      let Ψ'θ = InitMap(vars, λv. FixpointTemps(Ψθ, Ψθ(v)))
15:      in EVALℓ+1(Ψ'θ, pass_ℓ+1(¬(TE(e, Ψ'θ))))

16: function TE(e : Expr, Ψ : map[V, N]) : N =
17:   match e with
18:     "x" ⇒ Ψ(x)
19:     "op(e1, ..., ek)" ⇒ op(TE(e1, Ψ), ..., TE(ek, Ψ))

20: function PHI(n : N, Ψ1 : map[V, N], Ψ2 : map[V, N]) : map[V, N] =
21:   Combine(Ψ1, Ψ2, λ t f . φ(n, t, f))

22: function THETAℓ:ℕ(Ψ1 : map[V, N], Ψ2 : map[V, N]) : map[V, N] =
23:   Combine(Ψ1, Ψ2, λ b n . θℓ(b, n))

24: function EVALℓ:ℕ(Ψ : map[V, N], n : N) : map[V, N] =
25:   InitMap(Keys(Ψ), λv . evalℓ(Ψ(v), n)

26: function Combine(m1 : map[a, b], m2 : map[a, c], f : b * c → d) : map[a, d] =
27:   InitMap(Keys(m1) ∩ Keys(m2), λk.f(m1[k], m2[k]))
```

Figure 15: ML-style pseudo-code for converting SIMPLE programs to PEGs

- *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 $f(k)$.
- *Keys of a map.* Given a map m, Keys(m) returns the set of keys of m.
- *Map read.* Given a map m and a key $k \in$ 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 pseudo-code 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 6,7 for statements, and 18, 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 $e_1, \ldots, e_k$, 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 $n_1, \ldots, n_k$.

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 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 node context that maps 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 16(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 16(b) through 16(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 16(g).

6.3. **Type-directed translation.** In this section we formalize the translation process described by the pseudo-code implementation with a type-directed translation from SIMPLE programs to PEGs, in the style of [37]. The type-directed translation in Figure 17 is more complicated than the implementation in Figure 15, 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 15 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 15, as they rely on the fact that the program being translated is well-typed. In the type-directed translation, these

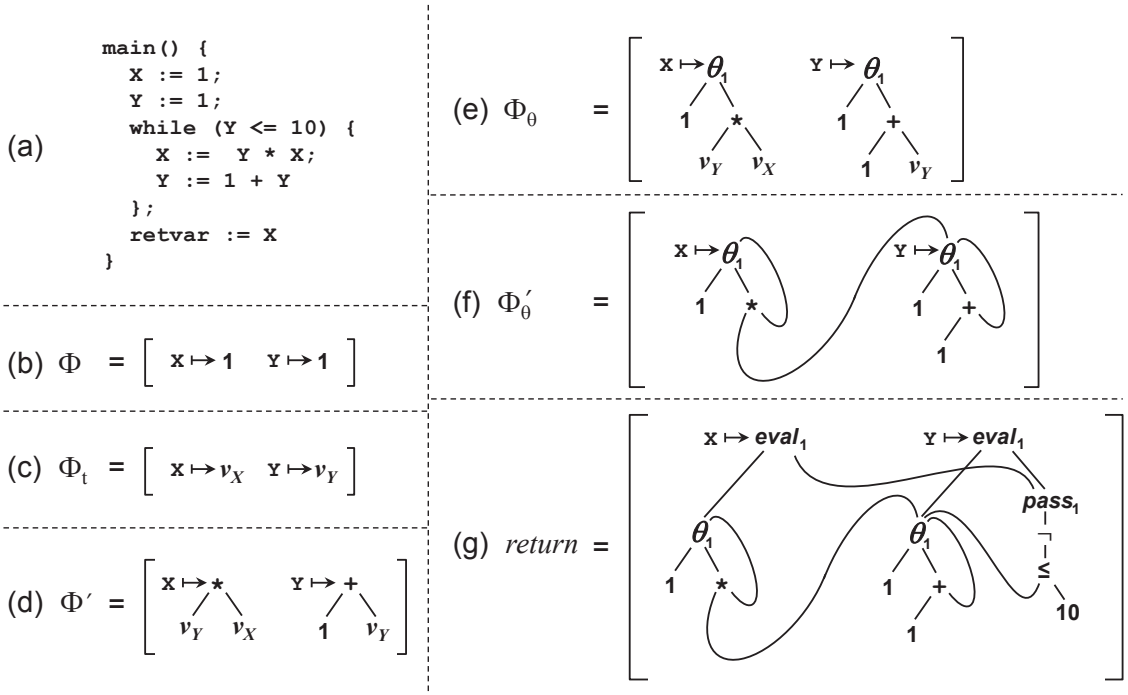

Figure 16: Steps of the translation process. (a) shows a SIMPLE program computing the factorial of 10; (b) through (f) show the contents of the variables in TS when processing the `while` loop; and (g) shows the return value of TS when processing the `while` loop.

properties are almost immediate since the translation operates on an actual proof that the program is well-typed.

In fact, the rules in Figure 17 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 judgment $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$ 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 judgment $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi'$ 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 judgment $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$, 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 $n_1 \ldots n_k$ are determined by translating the argument expressions $e_1 \ldots e_k$.

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

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

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

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

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

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

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

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

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

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

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

Figure 17: Type-directed translation from SIMPLE programs to PEGs

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 $\Gamma \vdash e : \tau \quad \rhd \quad \Psi \rightsquigarrow n$ 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 15, would explore a derivation of $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$ 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 17 and those in Figure 14. In particular, the formulas on the left of the $\triangleright$ correspond directly to the typing rules from Figure 14.

Statements. The translation for a statement s takes as input a derivation of the type-correctness of s, 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 judgment $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi'$, 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 judgment $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow \Psi'$ 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 judgment $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_{\ell'} \Psi'$ translates the body of the loop s 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 invariants 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 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 the `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 ensure 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. pseudo-code. The pseudo-code in Figure 15 follows the rules from Figure 17 very closely. Indeed, the code can be seen as using the rules from the type-directed translation to find a derivation of $\vdash p \quad \triangleright \quad n$. The search starts at the end of the derivation tree, and moves up from there. The entry-point of the pseudo-code is TranslateProg, which corresponds to rule Trans-Prog, the last rule to be used in a derivation of $\vdash p \quad \triangleright \quad n$. TranslateProg calls TS, which corresponds to finding a derivation for $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi'$. Finally, TS calls TE, which corresponds to finding a derivation for $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$. Each pattern-matching case in the pseudo-code corresponds to a rule from the type-directed translation.

The one difference between the pseudo-code and the type-directed translation is that in the judgments 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 pseudo-code does not manipulate any derivations. This can be explained by a simple erasure optimization in the pseudo-code: because of the structure of the type-checking 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 17 is essentially structural induction on the proof that the SIMPLE program is well typed, we can guarantee that its implementation in Figure 15 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.

6.4. **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 don't give full definitions, since these are standard. These functions are defined in terms of an *evaluation context* Σ, which is a map that for each variable x gives its value $\nu = \Sigma(x)$.

Definition 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 in Σ.*

Definition 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 3 (Semantics of programs)**.** *For a SIMPLE program* $\mathbf{main}(x_1 : \tau_1, \ldots, x_k : \tau_k)\{s\}$ *and an evaluation context Σ that maps each x_i to an appropriately typed value, we define* $[\![\mathbf{main}]\!](\Sigma) = [\![s]\!](\Sigma)(\texttt{retvar})$.

We will use these functions in our discussion below. For the translation defined in Section 6.3 we have proven the following theorem (recall the substitution notation $n[x \mapsto \nu]$ from Section 5.1).

Theorem 2. *If* (1) $\vdash \mathbf{main}(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau\ \{s\} \ \triangleright\ n$, (2) Σ *is an evaluation context mapping each x_i to an appropriately typed value ν_i, and* (3) $\hat{n} = n[x_1 \mapsto \nu_1, \ldots, x_k \mapsto \nu_k]$, *then* $[\![\mathbf{main}]\!](\Sigma) = \nu \implies [\![\hat{n}]\!](\lambda\ell.0) = \nu$.

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 that causes 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 these loops have *no* side-effects (aside from non-termination), and so 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 which do not contribute to the result [3]. In fact, the newly planned C++ standard allows the implementation to remove such IO-less loops, even if termination cannot be proven [2]. Nonetheless, at the end of this section, we give a brief overview of how to encode non-termination in PEGs.

In this theorem, we use both the function $[\![\cdot]\!]$ defined above for SIMPLE programs, as well as the function $[\![\cdot]\!]$ defined in Equation 5.1 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 thus 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 naturally 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 5.1, 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 $\mathit{invariant}_\ell(n)$. 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 $\mathit{invariant}_\ell$ to a PEG context as follows.

Definition 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 $\mathit{invariant}_\ell(\Psi)$.*

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

Lemma 1. *If $\Gamma \vdash e : \tau \;\; \triangleright \;\; \Psi \rightsquigarrow n$, then $\forall \ell, \mathit{invariant}_\ell(\hat{\Psi}) \implies \mathit{invariant}_\ell(\hat{n})$*

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

Lemma 2. *For all loop-depths ℓ, if $\Gamma \vdash s : \Gamma' \;\; \triangleright \;\; \Psi \rightsquigarrow_\ell \Psi'$, then*

$$\forall \ell' > \ell, \mathit{invariant}_{\ell'}(\hat{\Psi}) \implies \forall \ell' > \ell, \mathit{invariant}_{\ell'}(\hat{\Psi}')$$

Proof. Proved using induction on the proof of $\Gamma \vdash s : \Gamma'$, with Lemma 1. □

Using the above lemmas, and the fact that $\mathit{invariant}(\cdot)$ 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 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 3. *If $\Gamma \vdash e : \tau \quad \triangleright \quad \Psi \rightsquigarrow n$ and $[\![\hat{\Psi}]\!](\mathbf{i}) = \Sigma$, then*

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

Proof. Proved using induction on the proof of $\Gamma \vdash e : \tau$, with Lemma 1. □

Lemma 4. *For any loop-depth ℓ, if (1) $\Gamma \vdash s : \Gamma' \quad \triangleright \quad \Psi \rightsquigarrow_\ell \Psi'$, and (2) for each $\ell' > \ell$, $\mathit{invariant}_{\ell'}(\hat{\Psi})$ holds, and (3) $[\![\hat{\Psi}]\!](\mathbf{i}) = \Sigma$, then*

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

Proof. Proved using induction on the proof of $\Gamma \vdash s : \Gamma'$, with Lemmas 2 and 3. For the **while** case, the proof relies on the fact that syntactic invariance of $\hat{\Psi}$ implies semantic invariance of $\hat{\Psi}$, which implies that $\hat{\Psi}$ corresponds to Σ at all loop vectors $\mathbf{i}'$ which 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 don't contribute to the result. In particular, we can use the concept of an effect token. For our task of preserving non-termination, the effect token will encode the non-termination effect, although one can use a similar strategy for other effects (and in fact we use a similar strategy for encoding heap modifications and exceptions using a σ heap summary node in Section 8). Any effectful operation must take an effect token. If the operation might also change the state of the effect, then it must produce an effect token as output. In SIMPLE, the $\div$ operation could modify our non-termination effect token, if we choose to encode division-by-zero as non-termination (since SIMPLE does not contain exceptions). In we added functions to SIMPLE, then function calls would also consume and produce a non-termination effect token, 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 token. 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

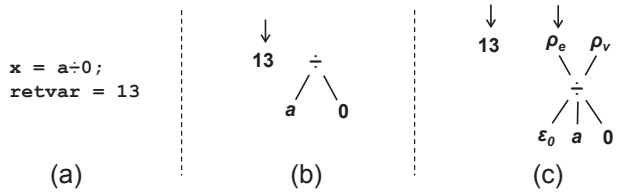

Figure 18: Representation of a division by zero: (a) the original source code, (b) the PEG produced without effect tokens, (c) the PEG produced with effect tokens. The arrows indicate the return values.

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 token through all the effectful operations (which includes *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 token 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 18 to demonstrate the use of effect tokens. The SIMPLE code in part (a) shows an attempt to divide by zero, followed by a return value of 13. Let's assume that $\div$ does not terminate when dividing by 0. A similar encoding works if $\div$ throws an exception instead of running forever – we describe our encoding of exceptions in Section 8, but in short the example would look essentially the same, except that we would use a σ heap summary node instead of an effect token. Part (b) shows the corresponding PEG without effect tokens. 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, and hence the PEG would be optimized to 13, which would remove the divide-by-zero (thus changing the termination behavior of the code).

Using a non-termination effect token 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 token 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. Reverting a PEG to Imperative Code

In this section we present the complement to the previous section: 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 translation in the forward direction. Because of the complexity involved in reversion, we start by presenting a simplified version of

the process in Sections 7.1 through 7.5. 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 7.6 through 7.9 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 Section 8), these optimizations are not optional anymore: they are needed to make sure that we don't incorrectly duplicate side-effecting operations. In our technical report [50],] 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 `break`s and `continue`s.

7.1. **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 6 (CFG-like PEG context)**.** *We say that a PEG context* Ψ *is CFG-like if* $\Gamma \vdash \Psi : \Gamma'$ *using the rules in Figure 19.*

The rules in Figure 19 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 5.1 from Section 5.1). Furthermore, all PEGs produced by our SIMPLE-to-PEG translation process from Section 6 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 8.3 ensures that the PEG selected for reversion is CFG-like.

In Figure 19, 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 an assumption context used to type-check 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 $x = \overline{\theta_2}(eval_1(eval_2(x, \dots), \dots), \dots)$, in which essentially the initializer for the nested loop is the final result of the outer loop.

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 20. Part (a) of the figure shows the syntactic form of the judgment (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.

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

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

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

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

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

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

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

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

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

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

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

$$\text{Type-Assume}\frac{(\ell \vdash n : \tau) \in \Theta}{\Gamma, \ell, \Theta \vdash n : \tau}$$

Figure 19: Rules for defining CFG-like PEGs

7.2. **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 19.

For the reversion process, we make two small changes to the type checking rules for SIMPLE programs. First, we want to allow the reversion process to introduce temporary

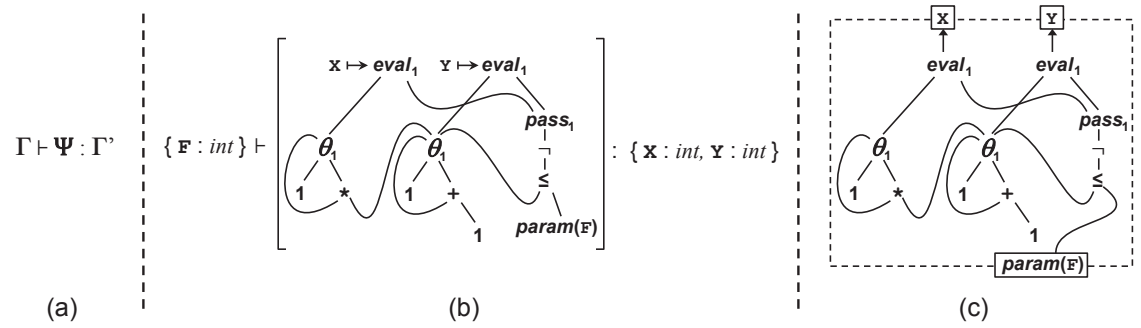

Figure 20: Visual representation of the judgment for CFG-like PEG contexts. (a) shows the syntactic form of the judgment; (b) shows an example of the syntactic form; and (c) shows the same example in visual form.

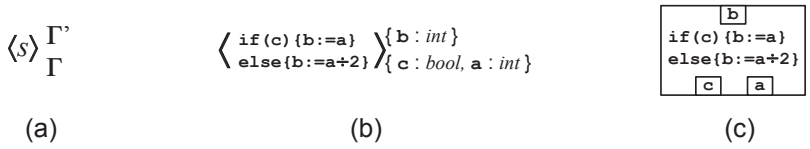

Figure 21: Diagrammatic representation of a statement node

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 14 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 19). 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 section 7, we assume that there is a PEG context $\overline{\Gamma_0}; \overline{\Gamma} \vdash \overline{\Psi} : \overline{\Gamma'}$ that we are currently reverting. Furthermore, we define Γ_0 to be:

$$\Gamma_0 = \overline{\Gamma_0} \cup \overline{\Gamma} \tag{7.1}$$

As will become clear in Section 7.3, the Γ_0 from Equation (7.1) 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 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 (7.1)). The node has many inputs, one for each variable in the domain Γ, and unlike any other node we've 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 21. 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 steps:

(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 7.3, 7.4 and 7.5, respectively. Finally, since the process described above is simplistic and results in large amounts of code duplication, we present in sections 7.6 through 7.9 several optimizations that improve the quality of the generated SIMPLE code.

7.3. **Translating Loops.** In our first pass, 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 22, 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 child (i.e. the

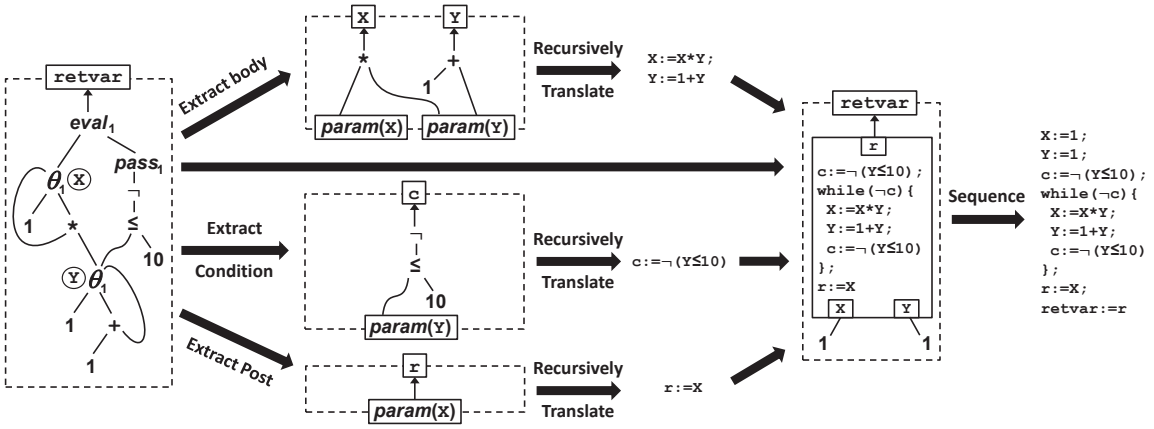

Figure 22: Example of converting *eval* nodes to while-loop statement nodes

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 22, 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 type-checking rules in Figure 19. For example, in Figure 22 the resulting type context Γ is $\{\texttt{x} : \texttt{int}, \texttt{y} : \texttt{int}\}$.

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 don't 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 (7.1)). 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 22 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 22 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 setup in Figure 19, 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 22 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 22 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 $\langle s_c; \texttt{while}\ (\neg x_c)\ \{s_i; s_c\}; s_r \rangle_\Gamma^{\{x_r:\tau\}}$. Figure 22 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 22 shows how in our running example, this amounts to connecting 1 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 22 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 7.5.

Note that in the computation of the break condition in Figure 22, there is a double negation, in that we have `c := ¬... ;` and `while(¬c)`. Our more advanced reversion algorithm, described in the accompanying technical report [50], takes advantage of more advanced control structures present in CFGs but not in SIMPLE, and does not introduce these double negations.

7.4. **Translating Branches.** After our first pass, 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 pass, 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

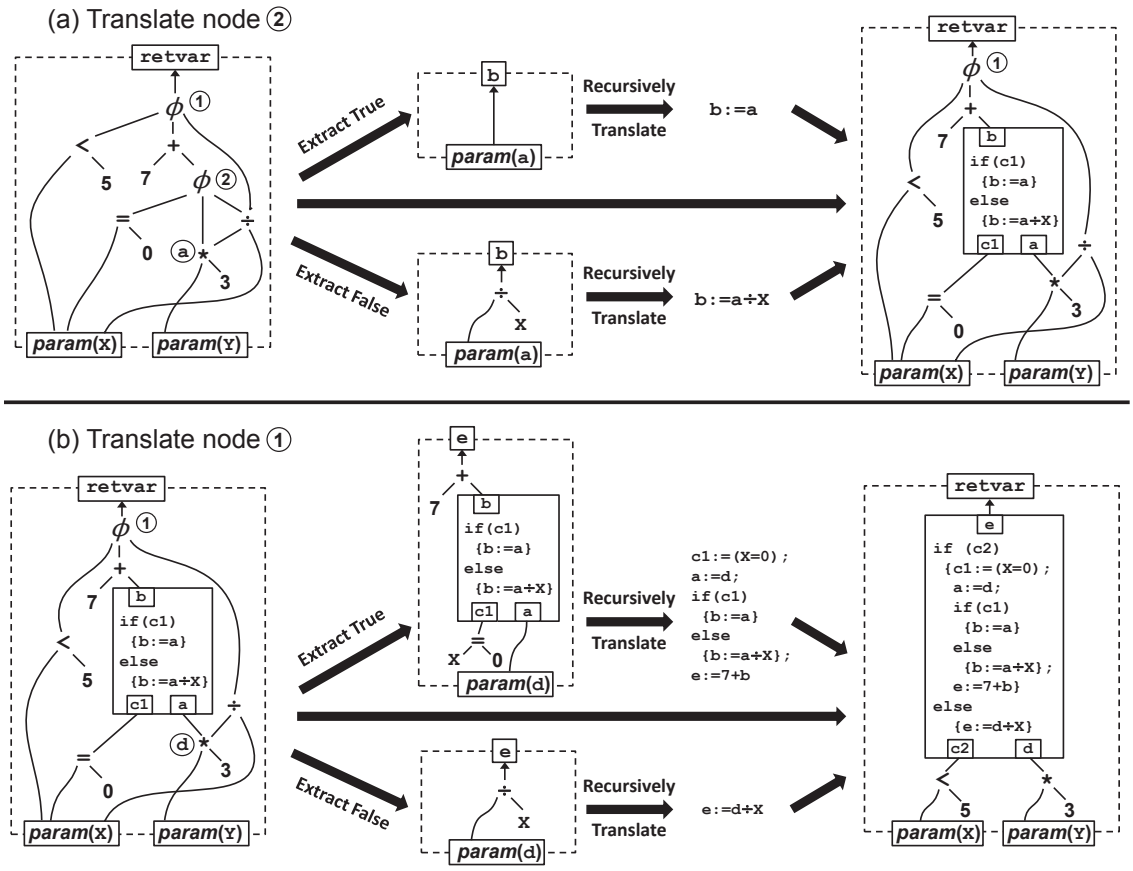

Figure 23: Example of converting ϕ nodes to if-then-else statement nodes

the ϕ node. To see why this would be the case, consider the example in Figure 23, 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 `x`$\neq$`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 23, 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've assigned to n. In Figure 23(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 23(a), t is $\overline{param}(\texttt{a})$ and f is $\overline{\div}(\overline{param}(\texttt{a}), \overline{param}(\texttt{x}))$. 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 $\{\texttt{a} : \texttt{int}\}$. 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 (7.1)). 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 23 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 23(b), but this poses no problems for our recursive algorithm. Finally, there is an important notational convention to note in Figure 23(a). Recall that Ψ_f satisfies $\Gamma_0; \Gamma \vdash \Psi_f : \{x_\phi : \tau\}$, and that in Figure 23(a) $\Gamma_0 = \{\texttt{x} : \texttt{int}\}$ and $\Gamma = \{\texttt{a} : \texttt{int}\}$. In the graphical representation of $\Gamma_0; \Gamma \vdash \Psi_f : \{x_\phi : \tau\}$ in Figure 23(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 $\langle \texttt{if}\ (x_c)\ \{s_t\}\ \texttt{else}\ \{s_f\}\rangle^{\{x_\phi:\tau\}}_{(\Gamma, x_c:\texttt{bool})}$ (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 23 shows the newly created if-then-else statement nodes and how they are connected when processing ϕ node 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.

7.5. **Sequencing Operations and Statements.** When we reach our final pass, 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

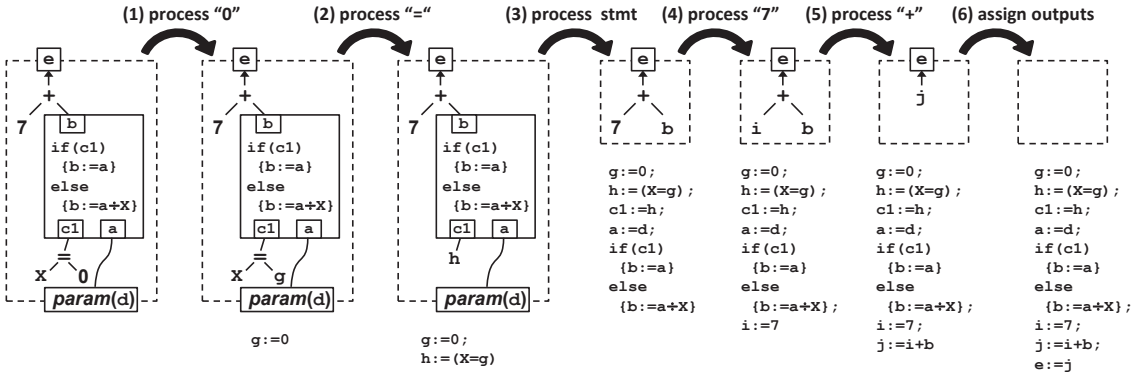

Figure 24: Example of sequencing a PEG context (with statement nodes) into a statement

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 24 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 24. Note that in the first and fourth steps, the constants 0 and 7 are a null-ary domain operators.

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 24.

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 24 (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.

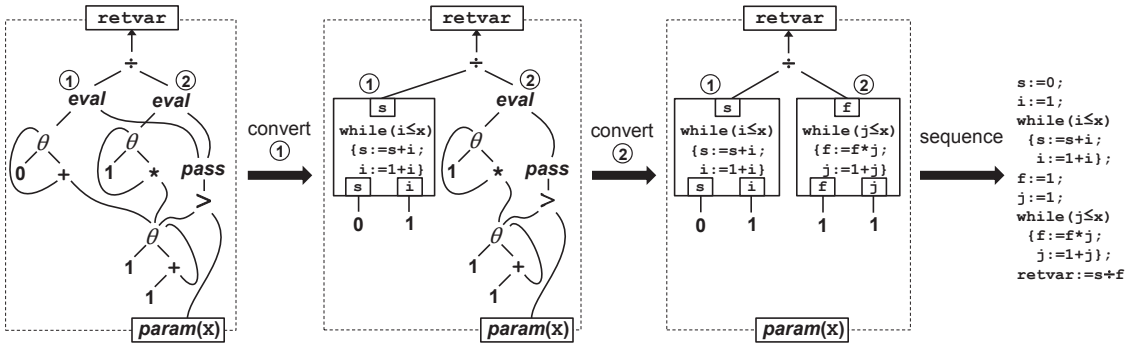

Figure 25: Reversion of a PEG without applying loop fusion

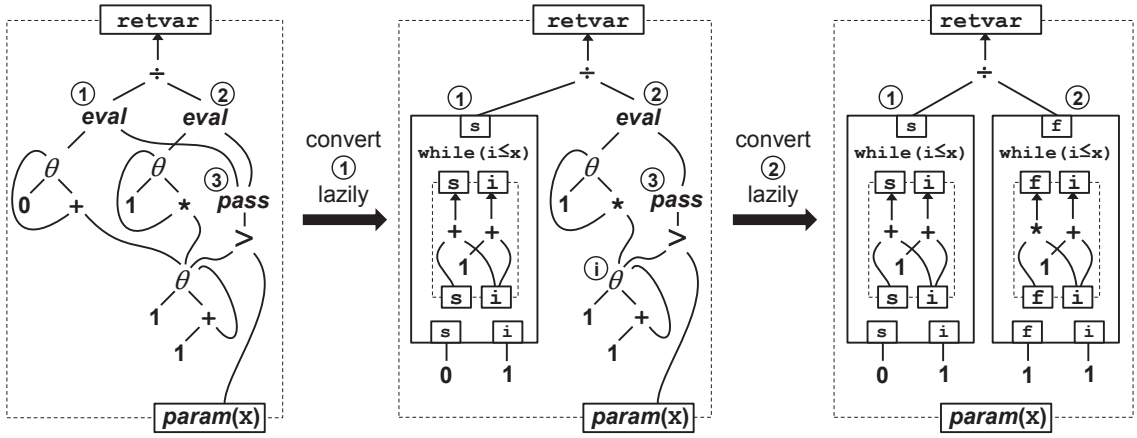

Figure 26: Conversion of *eval* nodes to revised loop nodes

7.6. **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 25. The original SIMPLE program for this PEG was as follows:

```
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 step 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 26 using the 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 26 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 26, we reuse the variable `i` unlike in Figure 25 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 26 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 22). As an example, nodes ① and ② in the right most part of Figure 26 are loop nodes. Furthermore, because we are leaving PEGs inside the loop nodes to be converted for later, we use the term "convert lazily" in Figure 26. 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 26, 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 26 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, and 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 27 demonstrates this process on the result of Figure 26. 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 27. 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 and (4) sequencing step. It is important to perform the fusion of loop nodes *after* converting ϕ nodes, rather than 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,

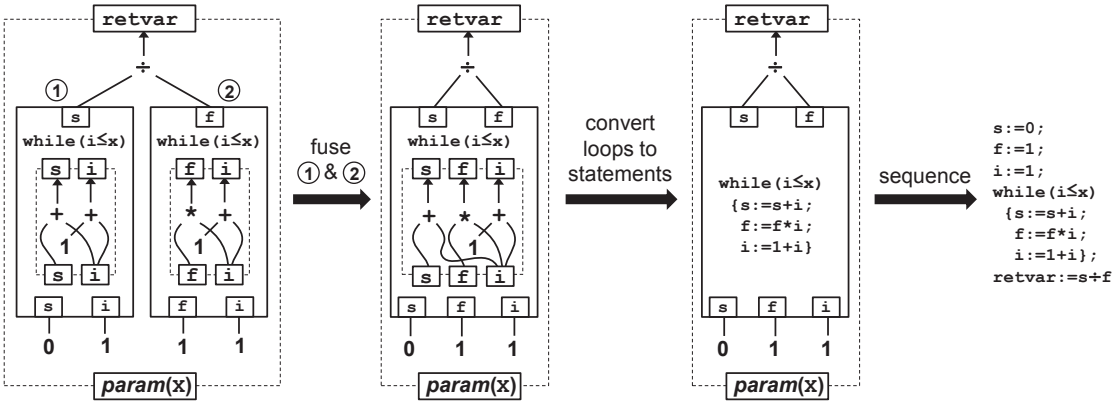

Figure 27: Fusion of loop nodes

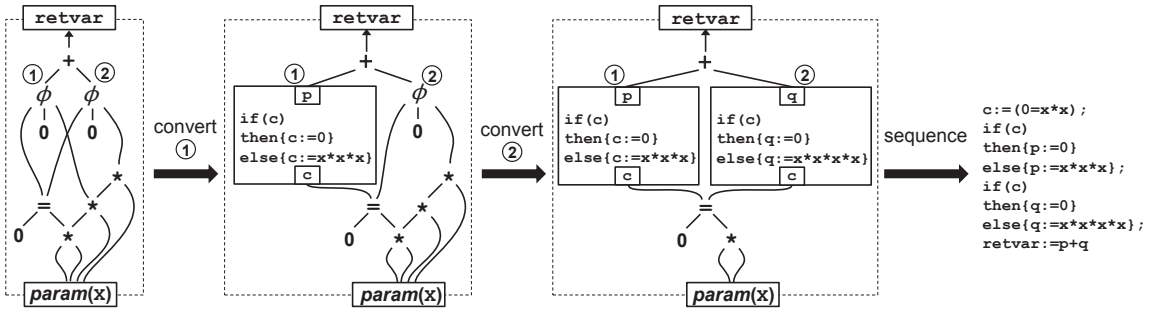

Figure 28: Reversion of a PEG without applying branch fusion

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, and so the semantics is preserved.

7.7. **Branch Fusion.** In the same way that our previously described reversion process duplicated loops, so does it duplicate branches, as demonstrated in Figure 28. 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 23, we instead create a new kind of node called a *branch node* and

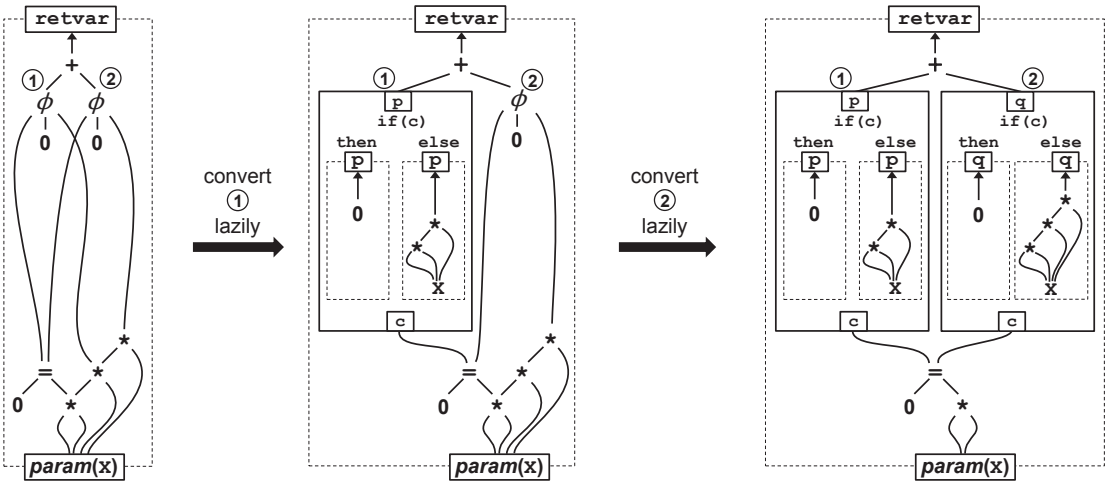

Figure 29: Conversion of ϕ nodes to revised branch nodes

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 29 shows this lazy conversion of ϕ nodes on the same example as Figure 28. The nodes labeled ① and ② in the right most part of Figure 29 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 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 30 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 30: 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 `c*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 30. 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 (4) perform fusion of branch nodes (5 sequencing step. 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,

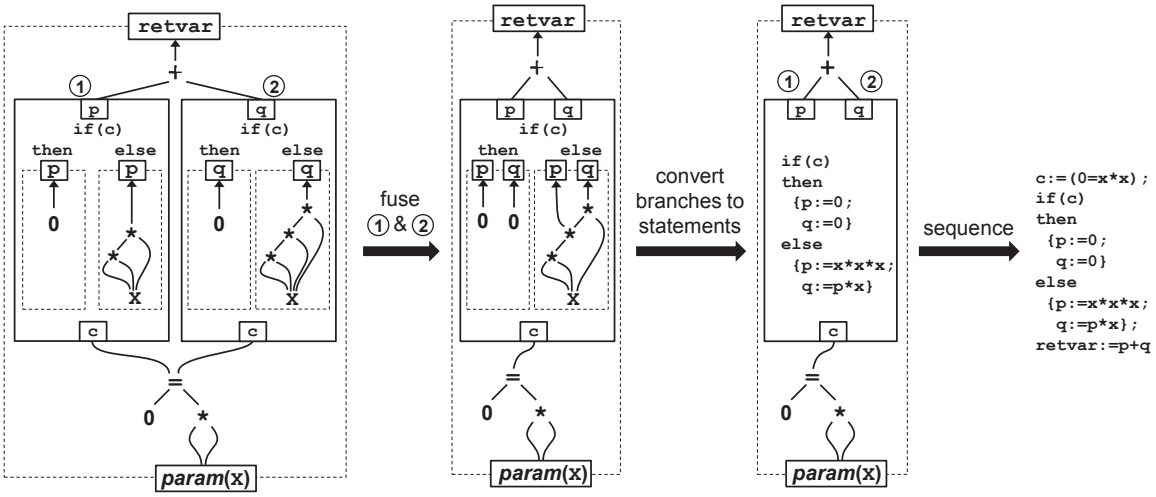

Figure 30: Fusion of branch nodes

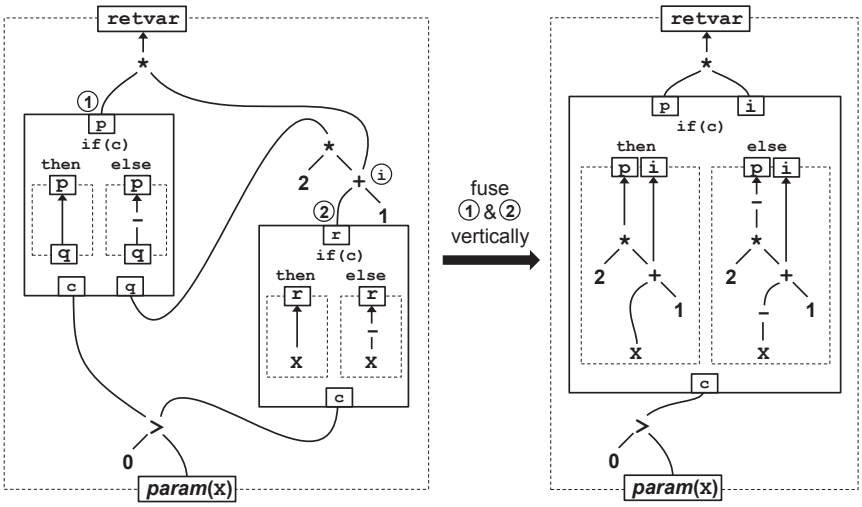

Figure 31: Vertical fusion of branch nodes

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 31. 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.

7.8. **Hoisting Redundancies from Branches.** Looking back at the branch fusion example from Figures 29 and 30, 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 7.4, 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 23 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 branch. This is precisely what happened in Figure 29: 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 `a` was in Figure 23). 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 23. 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.

Figure 32 shows the example from Figure 29 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 7.4, where ϕ nodes were processed starting from the lower ones (such as node ② in Figure 23) to the higher ones (such as node ① in Figure 23). Our updated process, when running on the example from Figure 23, would process node ① first, and in

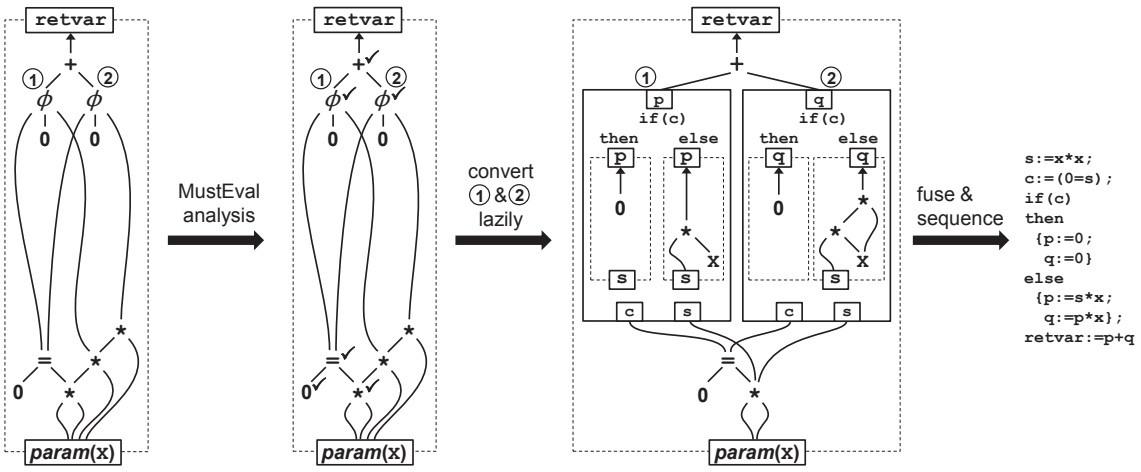

Figure 32: Same example as in Figure 29, but this time hoisting redundancies from branches

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 32, 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 32 we use the fresh variable Ⓢ for the node `x*x`. We then produce a t and f node for each ϕ node as before, replacing nodes which will always be evaluated with parameter nodes of appropriate variables. Figure 29 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 29 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 doesn't mark any nodes as being always evaluated, then we would not process any ϕ nodes, which is a problem before after the ϕ-processing stage, 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 7. *We say that a MustEval analysis is minimally precise if for any PEG context* Ψ, $S = \mathit{MustEval}(\Psi)$ *implies the following properties:*

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

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

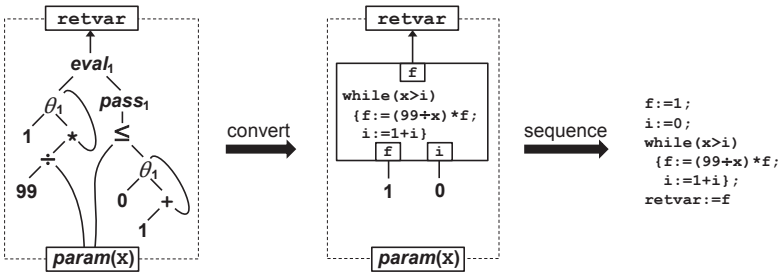

Figure 33: Reversion of a PEG without applying loop-invariant code motion

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.

7.9. **Loop-Invariant Code Motion.** The process described so far for converting *eval* nodes in Section 7.3 duplicates code unnecessarily. In particular, every node used by the loop body is copied into the loop body, including loop-invariant nodes. Figure 33 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 7.7 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 5.3 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 33. In this PEG the ÷ 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 ÷ 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.

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

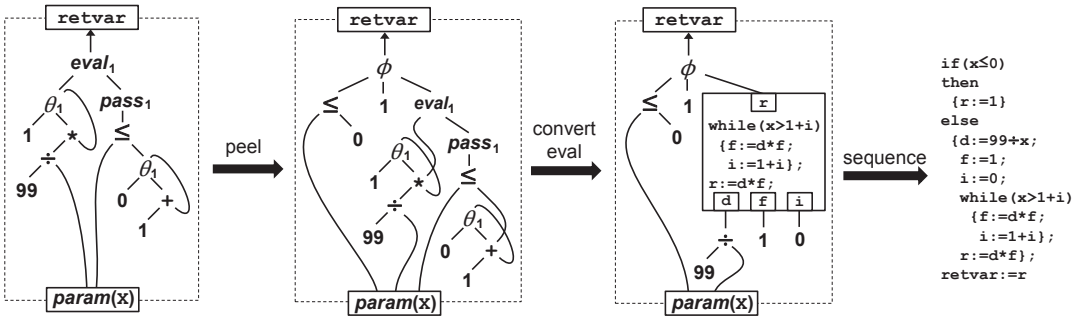

Figure 34: Reversion of a PEG after peeling the loop once

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 7.8, 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 a loop-invariant node if its evaluation condition is implied by the evaluation condition of the *eval* node being processed.

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 [7], 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 3.3), except that instead of using equality analyses, it is performed destructively on the PEG representation. Using the same starting example as before, Figure 34 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 34, 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 34. Figure 34 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 the guard 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 do not show the details here – instead we refer the interested reader to our technical report [50].

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 stage. This rearrangement enables more advanced redundancy elimination optimizations.

8. The Peggy Instantiation

In this section we discuss details of our concrete implementation of equality saturation as the core of an optimizer for Java bytecode programs. We call our system Peggy, named after our PEG intermediate representation. As opposed to the previous discussion of the SIMPLE language, Peggy operates on the entire Java bytecode instruction set, complete with side effects, method calls, heaps, and exceptions. Recall from Figure 10 that an instantiation 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.

8.1. Intermediate Representation.

Peggy uses the PEG and E-PEG representations which, as explained in Section 5, are well suited for our approach. Because Peggy is a Java bytecode optimizer, an additional challenge is to encode Java-specific concepts like the heap and exceptions in PEGs.

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/or return additional σ values. Because Java stack variables cannot be modified except by direct assignments, operations on stack variables 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.

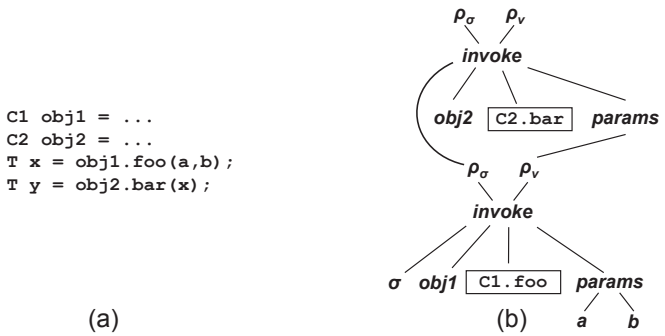

Figure 35: Representation of Java method calls in a PEG; (a) the original Java source code, (b) the corresponding PEG.

Method calls. Figure 35 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 heap summary, the receiver object of the method call, a method identifier, and a list of actual parameters. A static method call simply elides the receiver object. Logically, our `invoke` nodes return a tuple (σ, v), where σ is the resulting heap summary and the v is the return value of the method. The operator ρ_σ is used to project out the σ value 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 σ value from `foo` as its input σ value. This is how we encode the control dependency between the two `invoke` operators; by representing it as a data dependency on the heap. Many other Java operators have side effects that can be modeled as modifications to our σ heap summary, including array accesses, field accesses, object creation, and synchronization. They similarly take a σ value as input and/or return one as output, which encodes the data dependencies between the different heap operations.

Exceptions. In order to maintain the program state at points where exceptions are thrown, we bundle the exception state into our abstraction of the heap, namely the σ summary nodes. As a result, operations like division which may throw an exception, but do not otherwise modify the heap, now take and return a σ node (in addition to their regular parameters and return values).

The control flow of exceptions in a PEG is computed from a CFG where exceptional control flow has been made explicit. To build such an exception-aware CFG, we introduce a new boolean function called *isException*, which returns true if the current state of the program is in an exceptional state. After every statement in the CFG that could possibly cause an exception, we insert a conditional branch on *isException*, which jumps to the exception handler if one exists in the current method, or to the end of the CFG otherwise. Once exceptions have been made explicit in the CFG, we simply use our conversion algorithm to create a PEG from the exception-aware CFG. The *isException* tester in the CFG gets translated into a new PEG operator *isException*, which reads the output σ node of an operation, and returns a boolean indicating whether an exception occurred. The normal translation from CFG to PEG introduces the appropriate ϕ nodes to produce the correct values in the exception vs. non-exception cases.

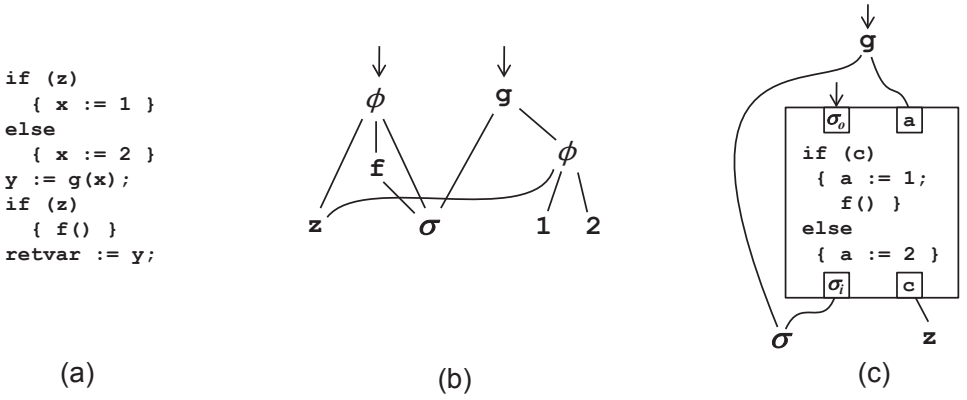

Figure 36: Example demonstrating challenges of linearizing heap values.

By bundling the exceptional state inside the heap summary node σ, and using an explicit tester function *isException*, the above encoding makes exceptions explicit. Since our optimization process preserves heap behavior, it therefore forces the observable state at the point where an exception is thrown to be preserved.

Reverting Uses of the Heap. One issue that affects our Peggy instantiation is that the σ nodes representing the heap cannot be duplicated or copied when running the program. More technically, heap values must be used linearly in the CFG. Linear values complicate the reversion presented in Section 7, which assumes any value can be duplicated freely. In order to adapt the algorithm from Section 7 to handle linear values, one has to "linearize" heap uses, which consists of finding a valid order for executing operations so that the heap does not need to be duplicated. Since PEGs come from actual programs which use the heap linearly, we decided to linearize the uses of the heap within each branch nodes and loop nodes 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 36(a), and its PEG representation in Figure 36(b). Suppose that `g` is a side-effect free operation of one parameter, which reads the heap and returns a value (but does not make any heap modifications). Thus, in the PEG representation, `g` takes two inputs: a heap value σ (which is the original heap at the entry), and its regular input parameter; `g` also returns a single value – the computed return value of `g` – but it does *not* return a new heap value since it does not modify the heap. We also assume that `f` is an operation that reads and writes to the heap, but returns `void`. For example, `f` could just increment a global on the heap. Thus, in the PEG representation, `f` takes an input heap parameter, and produces an output heap (but does not produce a real return value since it returns `void`). We see from the code in part (a) that the return value is produced by `g`. However, since `f` may have produced a new heap, we must also encode

the new heap as a return value. Thus, the code in part (a) returns two values: its regular return value, and a new heap, as shown with the two arrows in part (b).

Looking at the PEG in Figure 36(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 heap values) 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 heap. 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 first creates and fuses branch nodes, and then it tries to linearize heap values in each branch nodes. For example, Figure 36(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 of the branch block, the linearization constraints are not solvable anymore: `f` can no longer be executed after `g` since `g` relies on the result of the branch block which 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 heap nodes that takes a global view of the PEG is non-trivial. Heap nodes 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 isn't available to the reversion algorithm, though, so it cannot determine that the write can be safely moved after the read.

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 paper, we 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, then 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 [27]. This approach makes the memory overhead of an E-PEG proportional to the number of nodes it contains, and requires no additional memory to represent the equivalences between nodes.

8.2. **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

```
1: function Saturate(peg : PEG, A : set of analyses) : EPEG
2: let   epeg = CreateInitialEPEG(peg)
3: while ∃(p, f) ∈ A, subst ∈ S . subst = Match(p, epeg) do
4:     epeg := AddEqualities(epeg, f(subst, epeg))
5: return   epeg
```

Figure 37: Peggy's Saturation Engine. We use S to denote the set of all substitutions from pattern nodes to E-PEG nodes.

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, and when it is discovered, the engine constructs a *matching substitution*, which is a map from each node in the pattern to the corresponding 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.

Figure 37 shows the pseudo-code 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 AddEqualities to add the equalities computed by the analysis into the E-PEG.

A remaining concern in Figure 37 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 naive 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 [28]. 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 axioms 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 5 also checks that there is an appropriate "pass" expression. In this way, it does not add a loop to the E-PEG if there wasn't some kind of loop to begin with. Using our current axioms and triggers, our engine completely saturates 84% of the methods in our 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.

8.3. **Global Profitability Heuristic.** Peggy's SelectBest function uses a Pseudo-Boolean solver called Pueblo [47] to select which nodes from an E-PEG to include in the optimized program. A Pseudo-Boolean problem is an integer linear programming 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 which induces a set of equivalence classes N/E. Also recall that for $n \in N$, $params(n)$ is the list 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 (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:

$$\textbf{min} \sum_{n \in N} B_n \cdot C_n \ \textbf{s.t.}\ \mathsf{Constraints}(\langle N, L, C, E \rangle)$$

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, $C_n = \mathit{basic\_cost}(n) \cdot k^{\mathit{depth}(n)}$, where $\mathit{basic\_cost}(n)$ accounts for how expensive n is by itself, and $k^{\mathit{depth}(n)}$ accounts for how often n is executed. k is simply a constant, which we have chosen as 20. We use $\mathit{depth}(n)$ to denote the loop nesting depth of n, computed as follows (recalling definition 5.2 of $\mathit{invariant}_\ell$ from Section 5.3): $\mathit{depth}(n) = \max\{\ell \mid \neg \mathit{invariant}_\ell(n)\}$. Using this cost model, Peggy asks Pueblo to minimize the objective function subject to the constraints described above. Hence, the PEG that Pueblo returns is CFG-like and 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, which leads to more efficient code. One of the main contributors to the accuracy of our cost model is that $\mathit{depth}(n)$ is defined in terms of $\mathit{invariant}_\ell$, and $\mathit{invariant}_\ell$ is what the reversion process uses for pulling code outside of loops (see Section 7.9). 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 parameters. 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 $B_{i \rightsquigarrow j}$ 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.

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 [23] and SAT4J [1]. 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.

8.4. **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 3.3 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.

9. Evaluation

In this section we use our Peggy implementation to validate three hypotheses about our approach for structuring optimizers: our approach is practical both in terms of space

(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. TRE	Replaces the body of a tail-recursive procedure with a loop
6. Method Inlining	Inlining based on intraprocedural class analysis
7. Domain-specific	User-provided axioms about application domains
(b) Optimizations	**Description**
8. Constant Prop/Fold	Traditional Constant Propagation and Folding
9. Simplify Algebraic	Various forms of traditional algebraic simplifications
10. Peephole SR	Various forms of traditional peephole optimizations
11. Array Copy Prop	Replace read of array element by last expression written
12. CSE for Arrays	Remove redundant array accesses
13. Loop Peeling	Pulls the first iteration of a loop outside of the loop
14. LIVSR	Loop-induction-variable Strength Reduction
15. Interloop SR	Optimization described in Section 2
16. Entire-loop SR	Entire loop becomes one op, *e.g.* n incrs becomes "plus n"
17. Loop-op Factoring	Factor op out of a loop, *e.g.* multiplication
18. Loop-op Distributing	Distribute op into loop, where it cancels out with another
19. Partial Inlining	Inlines part of method in caller, but keeps the call
20. Polynomial Factoring	Evaluates a polynomial in a more efficient manner
(c) DS Opts	**Description**
21. DS LIVSR	LIVSR on domain ops like matrix addition and multiply
22. DS Code Hoisting	Code hoisting based on domain-specific invariance axioms
23. DS Remove Redundant	Removes redundant computations based on domain axioms
24. Temp. Object Removal	Remove temp objects made by calls to, *e.g.*, matrix libraries
25. Math Lib Specializing	Specialize matrix algs based on, *e.g.*, the size of the matrix
26. Design-pattern Opts	Remove overhead of common design patterns
27. Method Outlining	Replace code by method call performing same computation
28. Specialized Redirect	Replace call with more efficient call based on calling context

Figure 38: Optimizations performed by Peggy. Throughout this table we use the following abbreviations: EQ means "equality", DS means "domain-specific", TRE means "tail-recursion elimination", SR means "strength reduction"

and time (Section 9.1), it is effective at discovering both simple and intricate optimization opportunities (Section 9.2), and it is effective at performing translation validation (Section 9.3).

9.1. **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 4 main Peggy phases (for Pueblo, a timeout counts as 60 seconds).

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

All phases combined take slightly over 1.5 seconds. An end-to-end run of Peggy is on average 6 times slower than Soot with all of its intraprocedural optimizations turned on. Nearly all of our time is spent in the Pseudo-Boolean solver. We have not focused our efforts on compile-time, and we conjecture there is significant room for improvement, such as better pseudo-boolean encodings, or other kinds of profitability heuristics that run faster.

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

In 84% of compiled methods, the engine ran to complete saturation, without imposing bounds. For the remaining cases, the engine limit of 500 was reached, meaning that the engine ran until fully processing 500 expressions in the 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 200 MB of heap, our E-PEGs represented more than 2^{103} versions of the input program (using geometric average).

9.2. **Implementing optimizations.** The main goal of our evaluation is to demonstrate that common, as well as unanticipated, optimizations result in a natural way from our approach. To achieve this, we implemented a set of basic equality analyses, listed in Figure 38(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 38(b) shows the optimizations that we observed fall out from our approach using equality analyses 1 through 6, and Figure 38(c) shows optimizations that we observed fall out from our approach using equality analyses 1 through 7. Based on the optimizations we observed, we designed some micro-benchmarks that exemplify these optimizations. We then ran Peggy on each of these micro-benchmarks to show how much these optimizations improve the code when isolated from the rest of the program.

Figure 39 shows our experimental results for the runtimes of the micro-benchmarks listed in Figure 38(b) and (c). The y-axis shows run-time normalized to the runtime of the unoptimized code. Each number along the x-axis is a micro-benchmark exemplifying the optimization from the corresponding row number in Figure 38. The “rt” and “sp” columns correspond to our larger raytracer benchmark and SpecJVM, respectively. The value reported for SpecJVM is the average ratio over all benchmarks within SpecJVM. Our experiments with Soot involve running it with all intra-procedural optimizations turned on, which include: common sub-expression elimination, lazy code motion, copy propagation, constant propagation, constant folding, conditional branch folding, dead assignment elimination, and unreachable code elimination. Soot can also perform interprocedural optimizations, such as class-hierarchy-analysis, pointer-analysis, and method-specialization. We did not enable these optimizations when performing our comparison against Soot, because we have not yet attempted to express any interprocedural optimizations in Peggy. In terms of runtime improvement, Peggy performed very well on the micro-benchmarks, optimizing

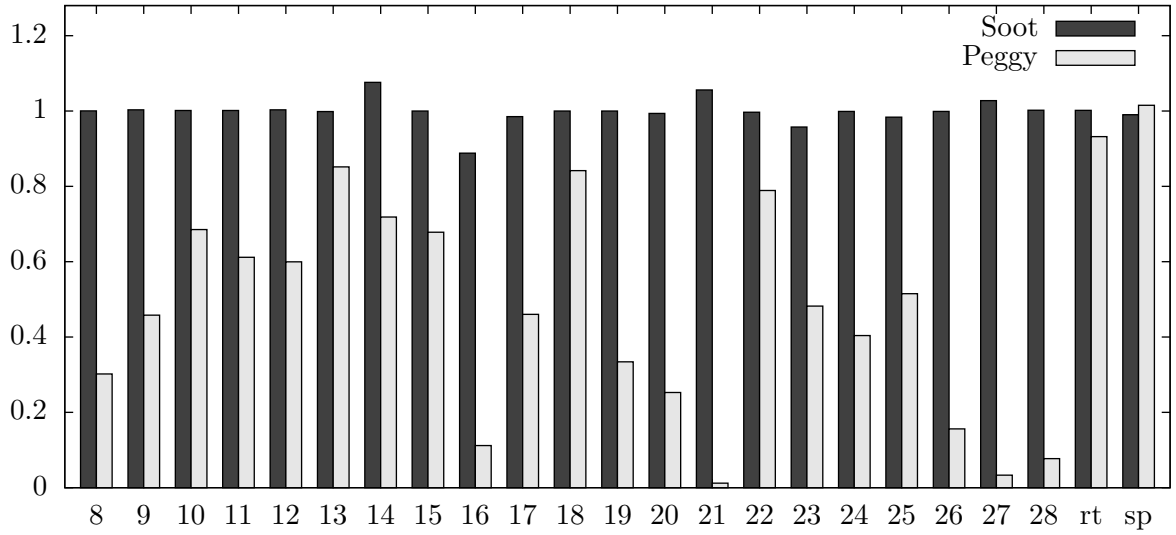

Figure 39: Runtimes of generated code from Soot and Peggy, normalized to the runtime of the unoptimized code. The x-axis denotes the optimization number from Figure 38, where "rt" is our raytracer benchmark and "sp" is the average over the SpecJVM benchmarks.

all of them by at least 10%, and in many cases much more. Conversely, Soot gives almost no runtime improvements, and in some cases makes the program run slower. For the larger raytracer benchmark, Peggy is able to achieve a 7% speedup, while Soot does not improve performance. On the SpecJVM benchmarks both Peggy and Soot had no positive effect, and Peggy on average made the code run slightly slower. This leads us to believe that traditional intraprocedural optimizations on Java bytecode generally produce only small gains, and in this case there were few or no opportunities for improvement.

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

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

The first is a ray tracer application (5 KLOCs) that one of the authors had previously developed. To make the implementation clean and easy to understand, the author used immutable vector objects in a functional programming style. This approach however introduces many intermediate objects. With a few simple vector axioms, Peggy is able to remove the overhead of these temporary objects, thus performing a kind of deforestation

optimization. This makes the application 7% faster, and reduces the number of allocated objects by 40%. Soot is not able to recover any of the overhead, even with interprocedural optimizations turned on. This is an instance of a more general technique where user-defined axioms allow Peggy to remove temporary objects (optimization 24 in Figure 38).

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 $l.\texttt{contains}(e) = (l.\texttt{indexOf}(e) \neq -1)$, 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 $(l.\texttt{indexOf}(e) \neq -1)$ 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 38 is presented in Appendix A.

9.3. **Translation Validation.** We used Peggy to perform translation validation for the Soot optimizer [52]. In particular, we used Soot to optimize a set of benchmarks with all of its intraprocedural optimizations turned on, which include: common sub-expression elimination, lazy code motion, copy propagation, constant propagation, constant folding, conditional branch folding, dead assignment elimination, and unreachable code elimination. 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 our approach that it is able not only to perform optimizations, but also to validate a large fraction of Soot runs, and that in doing so it exposed a bug in Soot. Furthermore, because most false positives are a consequence of our coarse heap model (single σ node), a finer-grained model can increase the effectiveness of translation validation, and it would also enable more optimizations.

Our equality saturation engine can easily be extended so that, after each translation validation, it generates a machine-checkable proof of equivalence. With this in place, the trusted computing base for our translation validator would only be: (1) the proof checker, (2) the built-in axioms used in translation validation, most of which we have proved by hand, and (3) the conversion from Java bytecode to PEG.

10. Related Work

Superoptimizers. Our approach of computing a set of programs and then choosing from this set is related to the approach taken by super-optimizers [39, 29, 11, 26]. Superoptimizers strive to produce optimal code, rather than simply improve programs. Although super-optimizers can generate (near) optimal code, they have so far scaled only to small code sizes, mostly straight line code. Our approach, on the other hand, is meant as a general purpose paradigm that can optimize branches and loops, as shown by the inter-loop optimization from Section 2.

Our approach was inspired by Denali [33], a super-optimizer for finding near-optimal ways of computing a given basic block. Denali represents the computations performed in the basic block as an expression graph, and applies axioms to create an E-graph data structure representing the various ways of computing the values in the basic block. It then uses repeated calls to a SAT solver to find the best way of computing the basic block given the equalities stored in the E-graph. The biggest difference between our work and Denali is that our approach can perform intricate optimizations involving branches and loops. On the other hand, the Denali cost model is more precise than ours because it assigns costs to entire sequences of operations, and so it can take into account the effects of scheduling and register allocation.

Rewrite-Based Optimizers. Axioms or rewrite-rules have been used in many compilation systems, for example TAMPR [13], ASF+SDF [53], the ML compilation system of Visser *et al.* [54], 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. Furthermore, 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 [59, 60], automated techniques for generating good sequences [17, 4, 34], manual techniques for combining analyses and optimizations [15], and automated techniques for the same purpose [38]. 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 Section 4, our approach can do well even if every part of the input program requires a different ordering.

Translation Validation. Although previous approaches to translation validation have been explored [45, 40, 62], our approach has the advantage that it can perform translation validation by using the same technique as for program optimization.

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 [19, 5, 51, 31, 24, 58, 16, 48, 44]. 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 [19], gated SSA [51] and thinned-gated SSA [31]. 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 Section 3. Furthermore, the conversion from any of the SSA representations back to imperative code is extremely simple since the CFG is already there. It suffices for each assignment $x := \phi(a, b)$ to simply insert the assignments $x := a$ and $x := b$ at the end of the two predecessors CFG basic blocks. The fact that our PEG representation is not tied to a CFG makes the conversion from PEGs back to a CFG-like representation much more challenging, since it requires reconstructing explicit control information.

The Program Dependence Graph [24] (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 [43] (PDW) combines the PDG with gated SSA. Our conversion algorithms have some similarities with the ones from the PDW. The PDW however still maintains explicit PDG control edges, whereas we do not have such explicit control edges, making converting back to a CFG-like structure more complex.

Dependence Flow Graphs [44] (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 [58] (VDG) is a complete functional representation. VDGs use λ nodes (i.e. regular function abstraction) to represent loops, whereas we use specialized θ, *eval* and *pass* nodes. Using λs as a key component in an IR is problematic for the equality saturation process. In order to effectively reason about λs one must particularly be able to reason about substitution. While this is possible to do during equality saturation, it is not efficient. The reason is that equality saturation is also being done to the body of the λ expression (essentially optimizing the body of the loop in the case of VDGs), so when the substitution needs to be applied, it needs to be applied to all versions of the body and even all future versions of the body as more axioms are applied. Furthermore, one has to determine when to perform λ abstraction on an expression, that is to say, turn

e into $(\lambda x.e_{body})(e_{arg})$, which essentially amounts to pulling e_{arg} out of e. Not only can it be challenging to determine when to perform this transformation, but one also has to take particular care to perform the transformation in a way that applies to *all* equivalent forms of e and e_{arg}.

The problem with λ expressions stems in fact from a more fundamental problem: λ expressions use *intermediate variables* (the parameters of the λs), and the level of indirection introduced by these intermediate variables adds reasoning overhead. In particular, as was explained above for VDGs, the added level of indirection requires reasoning about substitution, which in the face of equality saturation is cumbersome and inefficient. An important property of PEGs is that they have no intermediate variables. The overhead of using intermediate variables is also why we chose to represent effects with an effect token rather than using the techniques from the functional languages community such as monads [55, 56, 57] or continuation-passing style [6, 35, 30, 25, 8], 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 [32]. The most closely related is the Lucid programming language [9], 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 [21], with its E-graph data structure for representing equalities [42]. 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 [41].

Execution Indices. Execution indices identify the state of progress of an execution [22, 61]. 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.

11. Conclusion and future work

We have presented a new approach to structuring optimizers that is based on equality saturation. Our approach has a variety of benefits over previous compilation models: it addresses the phase ordering problem, it enables global profitability heuristics, and it performs translation validation.

There are a variety of directions for future work. One direction is to extend Peggy so that it generates a proof of correctness for the optimizations it performs. Each step in this proof would be the application of an equality analysis. Since the majority of our analyses are axiom applications, these proofs would be similar to standard mathematical proofs. We would then like to use these proofs as a way of automatically generating optimizations. In particular, by determining which nodes of the original PEG the proof depends on, and what properties of these nodes are important, we will explore how one can generalize not only the proof but also the transformation. Using such an approach, we hope to develop a compiler that can learn optimizations as it compiles.

Another direction involves addressing our heap linearizing issues when reverting a PEG to a CFG. One promising solution to this problem involves adapting our PEG representation to use *string diagrams* [10, 18]. Expressions are an excellent theory for non-linear values; string diagrams are a similar theory, but for linear values. A string diagram is comprised of nodes with many inputs and many outputs along with strings which connect outputs of nodes to inputs of other nodes. By default these strings cannot be forked, capturing the linear quality of the values carried by the strings; however, strings for non-linear types are privileged with the ability to fork. In addition to using string diagrams to encode linearity in our PEGs, we could also re-express all of our axioms in terms of string diagrams, thus preserving the linear qualities of any strings involved. This prevents the saturation engine from producing PEGs which cannot be linearized without additional information. Also, string diagrams can be used to preserve well-formedness of PEGs. Well-formedness constraints are the only quadratic component of our Pseudo-Boolean profitability heuristic formulation, so removing these constraints could drastically improve the speed of our Pseudo-Boolean solver.

Appendix A. Axioms

In this section we describe the axioms used to produce the optimizations listed in Figure 38. 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 38.

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 ops): This group of axioms relates to the special PEG operators θ, *eval*, and ϕ. Many of these axioms describe properties that hold for any operation OP.

- if $T = \theta_i(\mathbf{A}, T)$ exists, then $T = \mathbf{A}$
  [*If a loop-varying value always equals its previous value, then it equals its initial value*]
- if $\mathbf{A}$ is invariant w.r.t. i, then $eval_i(\mathbf{A}, \mathbf{P}) = \mathbf{A}$
  [*Loop-invariant operations have the same value regardless of the current loop iteration*]

- $\text{OP}(\mathbf{A_1}, \ldots, \theta_j(\mathbf{B_i}, \mathbf{C_i}), \ldots, \mathbf{A_k}) = \theta_j(\text{OP}(eval_j(\mathbf{A_1}, 0), \ldots, \mathbf{B_i}, \ldots, eval_j(\mathbf{A_k}, 0)), \text{OP}(peel_j(\mathbf{A_1}), \ldots, \mathbf{C_i}, \ldots, peel_j(\mathbf{A_k})))$
  [*Any operator can distribute through* θ_j]
- $\phi(\mathbf{C}, \mathbf{A}, \mathbf{A}) = \mathbf{A}$
  [*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*]
- $\text{OP}(\mathbf{A_1}, \ldots, eval_j(\mathbf{A_i}, \mathbf{P}), \ldots, \mathbf{A_k}) = eval_j(\text{OP}(\mathbf{A_1}, \ldots, \mathbf{A_i}, \ldots, \mathbf{A_k}), \mathbf{P})$, when $\mathbf{A_1}, \ldots, \mathbf{A_{i-1}}, \mathbf{A_{i+1}}, \ldots, \mathbf{A_k}$ are invariant w.r.t. j
  [*Any operator can distribute through* $eval_j$]

(Code patterns): These axioms are more elaborate and describe some complicated (yet still non-domain-specific) code patterns. These axioms are awkward to depict using our expression notation, so instead we present them in terms of before-and-after source code snippets.

- Unroll loop entirely:

```
x = B;                  ==      x = B;
for (i=0;i<D;i++)               if (D>=0) x += C*D;
   x += C;
```

  [*Adding C to a variable D times is the same as adding C*D (assuming* $D \geq 0$*)*]
- Loop peeling:

```
A;                      ==      if (N>0) {
for (i=0;i<N;i++)                  B[i -> 0];
   B;                              for (i=1;i<N;i++)
                                      B;
                                } else {
                                   A;
                                }
```

  [*This axiom describes one specific type of loop peeling, where* $B[i \to 0]$ *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 38.

- $(\mathbf{A} * \mathbf{B}) + (\mathbf{A} * \mathbf{C}) = \mathbf{A} * (\mathbf{B} + \mathbf{C})$
- if $\mathbf{C} \neq 0$, then $(\mathbf{A}/\mathbf{C}) * \mathbf{C} = \mathbf{A}$
- $\mathbf{A} * \mathbf{B} = \mathbf{B} * \mathbf{A}$
- $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$
- $\mathbf{A} * 1 = \mathbf{A}$
- $\mathbf{A} + 0 = \mathbf{A}$
- $\mathbf{A} * 0 = 0$

- $\mathbf{A} - \mathbf{A} = 0$
- $\mathbf{A} \mod 8 = \mathbf{A}\&7$
- $\mathbf{A} + (-\mathbf{B}) = \mathbf{A} - \mathbf{B}$
- $-(-\mathbf{A}) = \mathbf{A}$
- $\mathbf{A} * 2 = \mathbf{A} << 1$
- $(\mathbf{A} + \mathbf{B}) - \mathbf{C} = \mathbf{A} + (\mathbf{B} - \mathbf{C})$
- $(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C})$
- if $\mathbf{A} \geq \mathbf{B}$ then $(\mathbf{A} + 1) > \mathbf{B}$
- if $\mathbf{A} \leq \mathbf{B}$ then $(\mathbf{A} - 1) < \mathbf{B}$
- $(\mathbf{A} > \mathbf{A}) = \mathbf{false}$
- $(\mathbf{A} \geq \mathbf{A}) = \mathbf{true}$
- $\neg(\mathbf{A} > \mathbf{B}) = (\mathbf{A} \leq \mathbf{B})$
- $\neg(\mathbf{A} \leq \mathbf{B}) = (\mathbf{A} > \mathbf{B})$
- $(\mathbf{A} < \mathbf{B}) = (\mathbf{B} > \mathbf{A})$
- $(\mathbf{A} \leq \mathbf{B}) = (\mathbf{B} \geq \mathbf{A})$
- 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.** Each of these axioms provides useful information about a particular programming domain. These could be considered "application-specific" or "program-specific" axioms, and are only expected to apply to that particular application/program.

(Inlining): Inlining in Peggy acts like one giant axiom application, equating the inputs of the inlined PEG with the actual parameters, and the outputs of the PEG with the outputs of the INVOKE operator.

- Inlining axiom:

```
x = pow(A,B);        ==      result = 1;
                             for (e = 0;e < B;e++)
                                result *= A;
                             x = result;
```

[*A method call to pow is equal to its inlined body*]

(Sigma-invariance):— It is very common for certain Java methods to have no effect on the heap. This fact is often useful, and can easily be encoded with axioms like the following.

- $\rho_\sigma(\text{INVOKE}(\mathbf{S}, \mathbf{L}, [\texttt{Object List.get()}], \mathbf{P})) = \mathbf{S}$
  [`List.get` *is* σ*-invariant*]
- $\rho_\sigma(\text{INVOKE}(\mathbf{S}, \mathbf{L}, [\texttt{int List.size()}], \mathbf{P})) = \mathbf{S}$
  [`List.size` *is* σ*-invariant*]
- $\rho_\sigma(\text{INVOKESTATIC}(\mathbf{S}, [\texttt{double Math.sqrt(double)}], \mathbf{P})) = \mathbf{S}$
  [`Math.sqrt` *is* σ*-invariant*]

(Vector axioms): In our raytracer benchmark, there are many methods that deal with immutable 3D vectors. The following are some axioms that pertain to methods of the Vector class. These axioms when expressed in terms of PEG nodes are large and awkward, so we present them here in terms of before-and-after source code snippets.

- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.scaled}(\mathbf{D}) = \texttt{construct}(\mathbf{A} * \mathbf{D}, \mathbf{B} * \mathbf{D}, \mathbf{C} * \mathbf{D})$
  [*Vector* (A, B, C) *scaled by* D *equals vector* $(A * D, B * D, C * D)$]
- $\mathbf{A}\texttt{.distance2}(\mathbf{B}) = \mathbf{A}\texttt{.difference}(\mathbf{B})\texttt{.length2()}$
  [*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*]
- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.difference}(\texttt{construct}(\mathbf{D}, \mathbf{E}, \mathbf{F})) =$
  $\texttt{construct}(\mathbf{A} - \mathbf{D}, \mathbf{B} - \mathbf{E}, \mathbf{C} - \mathbf{F})$
  [*The difference of vectors* (A, B, C) *and* (D, E, F) *equals* $(A - D, B - E, C - F)$]
- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.dot}(\texttt{construct}(\mathbf{D}, \mathbf{E}, \mathbf{F})) = \mathbf{A} * \mathbf{D} + \mathbf{B} * \mathbf{E} + \mathbf{C} * \mathbf{F}$
  [*The dot product of vectors* (A, B, C) *and* (D, E, F) *equals* $A * D + B * E + C * F$]
- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.length2()} = \mathbf{A} * \mathbf{A} + \mathbf{B} * \mathbf{B} + \mathbf{C} * \mathbf{C}$
  [*The squared length of vector* (A, B, C) *equals* $A^2 + B^2 + C^2$]
- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.negative()} = \texttt{construct}(-\mathbf{A}, -\mathbf{B}, -\mathbf{C})$
  [*The negation of vector* (A, B, C) *equals* $(-A, -B, -C)$]
- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.scaled}(\mathbf{D}) = \texttt{construct}(\mathbf{A} * \mathbf{D}, \mathbf{B} * \mathbf{D}, \mathbf{C} * \mathbf{D})$
  [*Scaling vector* (A, B, C) *by* D *equals* $(A * D, B * D, C * D)$]
- $\texttt{construct}(\mathbf{A}, \mathbf{B}, \mathbf{C})\texttt{.sum}(\texttt{construct}(\mathbf{D}, \mathbf{E}, \mathbf{F})) =$
  $\texttt{construct}(\mathbf{A} + \mathbf{D}, \mathbf{B} + \mathbf{E}, \mathbf{C} + \mathbf{F})$
  [*The sum of vectors* (A, B, C) *and* (D, E, F) *equals* $(A + D, B + E, C + F)$]
- $\texttt{getZero().mX} = \texttt{getZero().mY} = \texttt{getZero().mZ} = 0.0$
  [*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 wrapper object:
  $\mathbf{A}$`.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`:

```
Object o = ...                    ==   Object o = ...
List l = ...                           List l = ...
if (l.contains(o)) {                   int index = l.indexOf(o);
   int index = l.indexOf(o);           if (index >= 0) {
   ...                                    ...
}                                      }
```

  [*Checking if a list contains an item then asking for its index is redundant*]

(Method Outlining): Method "outlining" is the opposite of method inlining; it is an attempt to replace a snippet of code with a procedure call that performs the same task. This type of optimization is useful when refactoring code to remove a common yet inefficient snippet of code, by replacing it with a more efficient library implementation.

- Body of selection sort replaced with `Arrays.sort(int[])`:

```
length = A.length;                ==       Arrays.sort(A);
for (i=0;i<length;i++) {
   for (j=i+1;j<length;j++) {
      if (A[i] > A[j]) {
         temp = A[i];
         A[i] = A[j];
         A[j] = temp;
      }
   }
}
```

(Specialized Redirect): This optimization is similar to Method Outlining, but instead of replacing a snippet of code with a procedure call, it replaces one procedure call with an equivalent yet more efficient one. This is usually in response to some learned contextual information that allows the program to use a special-case implementation.

- if $I = \text{INVOKESTATIC}(\mathbf{S}, [\texttt{void sort(int[])}], \text{PARAMS}(\mathbf{A}))$ exists,
  then add equality $\mathit{isSorted}(\rho_\sigma(I), \mathbf{A}) = \mathbf{true}$
  [*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

We would like to thank Jeanne Ferrante, Todd Millstein, Christopher Gautier, members of the UCSD Programming Systems group, and the anonymous reviewers for giving us invaluable feedback on earlier drafts of this paper.

References

[1] SAT4J. http://www.sat4j.org/.

[2] C++0x draft standard, 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf, page 130.

[3] Embedded in Academia, 2010. http://blog.regehr.org/archives/140.

[4] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *LCTES*, 2004.

[5] B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *POPL*, January 1988.

[6] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1991.

[7] A. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[8] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, 1997.

[9] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[10] John C. Baez and Mike Stay. Physics, topology, logic and computation: A rosetta stone. http://arxiv.org/abs/0903.0340. Mar 2009.

[11] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[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] K. D. Cooper C. Click. Combining analyses, combining optimizations. *Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.

[16] C. Click. Global code motion/global value numbering. In *PLDI*, June 1995.

[17] K. D. Cooper, P. J. Schielke, and Subramanian D. Optimizing for reduced code space using genetic algorithms. In *LCTES*, 1999.

[18] Pierre-Louis Curien. The joy of string diagrams. In *CSL '08: Proceedings of the 22nd international workshop on Computer Science Logic*, pages 15–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method for computing static single assignment form. In *POPL*, January 1989.

[20] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Conference on LISP and Functional Programming*, 1994.

[21] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. *Journal of the Association for Computing Machinery*, 52(3):365–473, May 2005.

[22] E. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147 – 148, 1968.

[23] 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.

[24] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[25] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, New York, NY, USA, 1993. ACM.

[26] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG – fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[27] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 246–251, New York, NY, USA, 1983. ACM.

[28] J. Giarratano and G. Riley. *Expert Systems – Principles and Programming*. PWS Publishing Company, 1993.

[29] Torbjorn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *PLDI*, 1992.

[30] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471, New York, NY, USA, 1994. ACM.
[31] P. Havlak. Construction of thinned gated single-assignment form. In *Workshop on Languages and Compilers for Parallel Computing*, 1993.
[32] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.
[33] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *PLDI*, June 2002.
[34] L. Torczon K. D. Cooper, D. Subramanian. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, pages 7–22, 2002.
[35] Andrew Kennedy. Compiling with continuations, continued. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 177–190, New York, NY, USA, 2007. ACM.
[36] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.
[37] Daan Leijen. A type directed translation of MLF to System F. In *The International Conference on Functional Programming (ICFP'07)*. ACM Press, October 2007.
[38] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *POPL*, January 2002.
[39] Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS*, 1987.
[40] G. Necula. Translation validation for an optimizing compiler. In *PLDI*, June 2000.
[41] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
[42] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.
[43] K. Ottenstein, R. Ballance, and A. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, June 1990.
[44] K. Pengali, M. Beck, and R. Johson. Dependence flow graphs: an algebraic approach to program dependencies. In *POPL*, January 1991.
[45] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.
[46] W. Quine. *Word and Object*. Simon and Schuster, New York, 1964.
[47] H. Sheini and K. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:61–96, 2006.
[48] B. Steffen, J. Knoop, and O. Ruthing. The value flow graph: A program representation for optimal program transformations. In *European Symposium on Programming*, 1990.
[49] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, 2000.
[50] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Translating between PEGs and CFGs. Technical report, University of California, San Diego, October 2010.
[51] P. Tu and D. Padua. Efficient building and placing of gating functions. In *PLDI*, June 1995.
[52] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON*, 1999.
[53] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *Transactions on Programming Languages and Systems*, 24(4), 2002.
[54] E. Visser, Z. Benaissa, and A Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, 1998.
[55] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM.
[56] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.
[57] Philip Wadler. The marriage of effects and monads. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 63–74, New York, NY, USA, 1998. ACM.
[58] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *POPL*, 1994.

[59] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *PPOPP*, 1990.
[60] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.
[61] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI*, June 2008.
[62] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.