Equality Saturation: a New Approach to Optimization [*]

Ross Tate   Michael Stepp   Zachary Tatlock   Sorin Lerner

Department of Computer Science and Engineering
University of California, San Diego
{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.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Compilers; Optimization

General Terms Languages, Performance

Keywords Compiler Optimization, Equality Reasoning, Intermediate Representation

1. 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 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 8 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 two programs are found equal

after equality saturation, then they 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 2 and 3 present an overview of our approach and its capabilities, Section 4 makes our approach formal, and Section 5 describes the new IR that allows our approach to be effective.

- We have instantiated our approach in a new Java bytecode optimizer called Peggy (Section 6). Peggy uses our approach not only to optimize Java methods, but also to perform translation validation for existing compilers. Our experimental results (Section 7) 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 [36], and within the remaining 2% it uncovered a bug in Soot.

2. 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 [35, 19], 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 well-formed program, 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

```
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.

classes allows E-PEGs to efficiently represent exponentially many ways of expressing the input program, and it also allows the equality saturation engine to efficiently take into account previously discovered equalities. Among existing IRs, E-PEGs are unique in their ability to represent multiple optimized versions of the input program. A more detailed discussion of how PEGs and E-PEGs relate to previous IRs can be found in Section 8.

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).

2.1 Program Expression Graphs

A Program Expression Graph (PEG) is a graph containing: (1) operator nodes, for example "plus", "minus", or any of our built-in nodes for representing conditionals and loops (2) "dataflow" edges that specify where operator nodes get their arguments from. As an example, the PEG for the use of i*5 in Figure 1(a) 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 [26, 27, 13] 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 [11, 2]. 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. 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. As in gated SSA [35, 19], our ϕ nodes are executable: the first (left-most) argument to ϕ is a selector that is used to select between the second and the third argument. 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.

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.

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. Even though all PEG operators are pure, PEGs can still represent programs with state by using heap summary nodes. Section 6 explains how our Peggy compiler uses such heap summary nodes to represent the state of Java objects.

2.2 Encoding equalities using E-PEGs

A PEG by itself can only represent a single way of expressing the input program. To represent *multiple* optimized versions of the input program, we need to encode equalities in our representation. To this end, an E-PEG is a graph that groups together PEG nodes that are equal into equivalence classes. As an example, Figure 2(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:

$$(a + b) * m = a * m + b * m \tag{1}$$

$$\theta(a, b) * m = \theta(a * m, b * m) \tag{2}$$

$$\phi(a, b, c) * m = \phi(a, b * m, c * m) \tag{3}$$

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 7 will describe in more details, 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 (2) adds equality edge A, axiom (3) edge B, axiom (1) edge C, and axiom (1) edge D. Our engine also applies the constant folding equality analysis to show that $0 * 5 = 0$, $3 * 5 = 15$ and
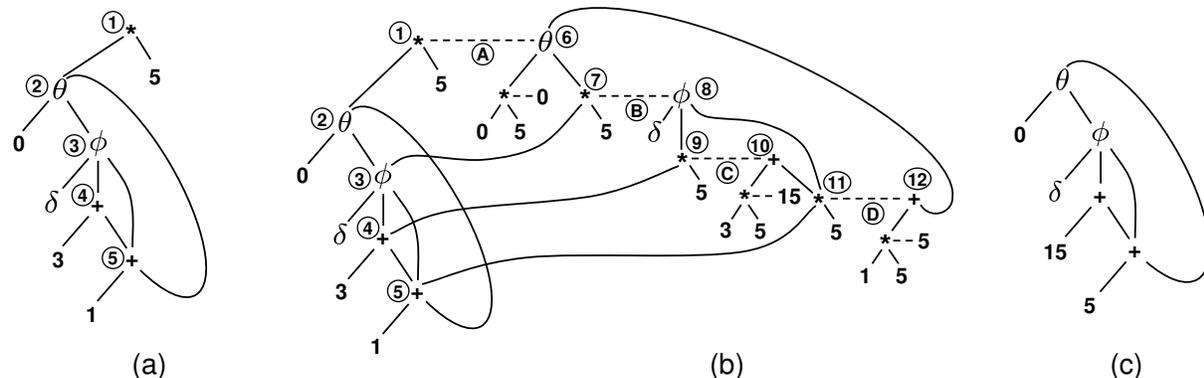
Figure 2. Loop-induction-variable Strength Reduction using PEGs: (a) shows the original PEG, (b) shows the E-PEG that our engine produces from the original PEG and (c) shows the optimized PEG, which results by choosing nodes 6, 8, 10, and 12 from (b).

$1 * 5 = 5$. Note that when axiom (2) adds edge A, it also adds node 7, which then enables axiom (3). Thus, equality analyses essentially communicate with each other by propagating equalities through the E-PEG. Furthermore, note that the instantiation of axiom (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 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 6.3 describes our technique for selecting nodes in more detail.

2.3 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: trans-

forming `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 [41] 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 3 gives such an example. Second, it makes it easier for end-user programmers to add domain-specific axioms to the compiler, because they
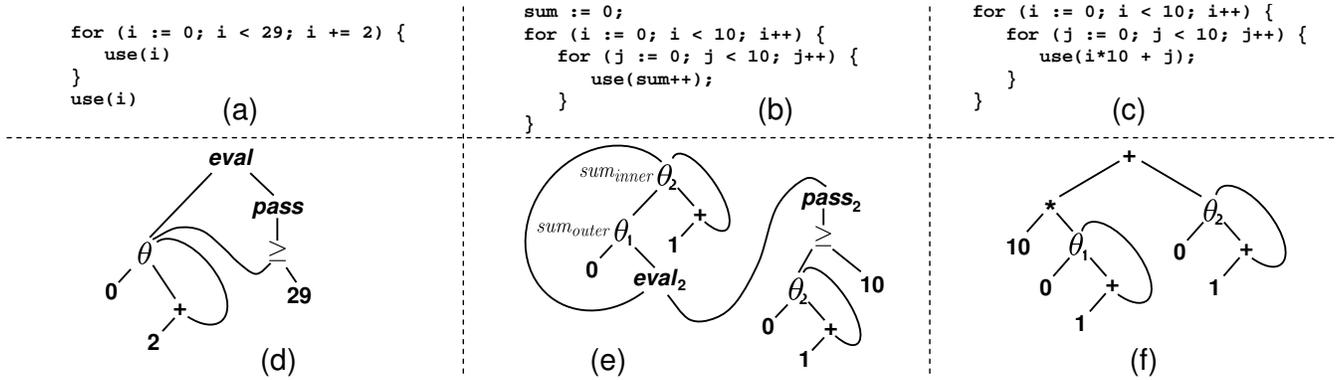
```
for (i := 0; i < 29; i += 2) {
    use(i)
}
use(i)                  (a)
```

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

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

Figure 3. Various loops and their PEG representations.

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* [12], 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 necessarily 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.

3. 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.

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

3.1 Single loop

Consider the simple loop from Figure 3(a). This loop iterates 15 times, incrementing the value of i each time by 2. Assume that the variable i is used inside the loop, and it is also used after the loop (as indicated by the two use annotations). The PEG for this code is shown in Figure 3(d). 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 nth 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 i $\geq$ 29 throughout the loop. This sequence is then sent to *pass*, which in this case produces the value 15, since the 15th value (counting from 0) of i in the loop (which is 30) is the first one to make i $\geq$ 29 true. The *eval* node then selects the 15th element of the sequence produced by the θ node, which is 30. In our previous example from Section 2, we omitted *eval/pass* from the PEG for clarity – because we were not

Figure 4. E-PEG that results from running the saturation engine on the PEG from Figure 3(f). By picking the nodes that are check-marked, we get the PEG from Figure 3(e).

interested in any of the values after the loop, the *eval*/*pass* nodes would not have been used in any reasoning.

3.2 Nested loops

We now illustrate, through an example, how nested loops can be encoded in our PEG representation. Consider the code snippet from Figure 3(b), which has two nested loops. We are interested in the value of sum inside the loop, as indicated by the use annotation. The PEG for this code snippet is shown in Figure 3(e). 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 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; 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(d).

3.3 Inter-loop strength reduction

Our approach allows an optimizing compiler to perform intricate optimizations of looping structures. We present such an example here, with a kind of inter-loop strength reduction. Consider the code snippet from Figure 3(c), which is equivalent to one we've already seen in Figure 3(b). However, the code in 3(b) is faster because sum++ is cheaper than i * 10 + j. We show how our approach can transform the code in Figure 3(c) to the code in Figure 3(b).

Figure 3(f) shows the PEG for i*10 + j, which will be the focus of our optimization. We omit *eval* and *pass* in this PEG because they are not used in this example, except in one step that we will make explicit.

Figure 4 shows the saturated E-PEG that results from running the saturation engine on the PEG from Figure 3(f). The checkmarks indicate which nodes will eventually be selected – they can be ignored for now. 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.

In drawing Figure 4, we have already performed loop-induction variable strength reduction on the left child of the topmost + from

Figure 3(f). In particular, this left child has been replaced with a new node i, where $i = \theta_1(0, 10 + i)$. We skip the steps in doing this because they are similar to the ones described in Section 2.2.

Figure 4 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.e.: $i + 0 = i$.

- 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 4 to highlight the nodes that Peggy would select using its Pseudo-Boolean profitability heuristic. These nodes constitute exactly the PEG from Figure 3(e), meaning that Peggy optimizes the code in Figure 3(c) to the one in Figure 3(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.

4. Formalization of our Approach

Having given an intuition of how our approach works through examples, we now move to a formal description. Figure 5 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.

Function Optimize($cfg : CFG$) : CFG

1: **let** $ir = \text{ConvertToIR}(cfg)$
2: **let** $saturated\_ir = \text{Saturate}(ir, A)$
3: **let** $best = \text{SelectBest}(saturated\_ir)$
4: **return** $\text{ConvertToCFG}(best)$

Figure 5. 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 $\mathrm{ConvertToIR}$ and $\mathrm{ConvertToCFG}$, (2) a saturation engine $\mathrm{Saturate}$, and (3) a global profitability heuristic $\mathrm{SelectBest}$. Future sections will show how we instantiate these three components in our Peggy compiler.

Saturation Engine. The saturation engine $\mathrm{Saturate}$ 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.

We define a partial order $\sqsubseteq$ on IRs, based on the equalities they encode: $ir_1 \sqsubseteq ir_2$ iff the equalities in ir_1 are a subset of the equalities in ir_2. Immediately from this definition, we get:

$$(ir_1 \xrightarrow{a} ir_2) \Rightarrow ir_1 \sqsubseteq ir_2 \qquad (4)$$

We define an equality analysis a to be monotonic iff:

$$(ir_1 \sqsubseteq ir_2) \wedge (ir_1 \xrightarrow{a} ir_1') \wedge (ir_2 \xrightarrow{a} ir_2') \Rightarrow (ir_1' \sqsubseteq ir_2')$$

Intuitively, our approach addresses the phase ordering problem because applying an equality analysis a before b cannot make b less effective, as stated in the following non-interference theorem.

THEOREM 1. *If a and b are monotonic then:*

$$(ir_1 \xrightarrow{a} ir_2) \wedge (ir_2 \xrightarrow{b} ir_3) \wedge (ir_1 \xrightarrow{b} ir_4) \Rightarrow (ir_4 \sqsubseteq ir_3)$$

The above follows immediately from monotonicity and Property 4.
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.

Given a set A of monotonic equality analyses, if the saturation engine terminates, then it is canonizing, where canonizing means that for any ir_1, there is a unique ir_2 with the following properties: (1) $ir_1 \rightarrow^* ir_2$ and (2) there is no ir_3 such that $ir_2 \rightarrow ir_3$. In this case the saturation engine computes this canonical saturated IR, which means that $\mathrm{Optimize}$ returns the same result no matter what order optimizations run in.

Because in general saturation may not terminate, we bound the number of times that analyses can run. In this case we cannot provide the same canonizing property, but the non-interference theorem (Theorem 1) 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 5.2), and finally we give a high-level description of how to translate from CFGs to our IR and back (Section 5.3).

5.1 Formalization of PEGs and E-PEGs

A PEG is a triple (N, L, C), 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 $C : N \rightarrow list[N]$ is a function that maps each node to its children (i.e. arguments).

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 $\bot$-lifted, meaning that we add the special value $\bot$ to each type domain. The $\bot$ value indicates that the computation fails or does not terminate. Formally, for each type τ, we define $\widehat{\tau} = \tau \cup \{\bot\}$.

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 5th iteration and loop ℓ_2 is on the 3rd iteration. We let $\mathbb{I} = \mathcal{L} \rightarrow \mathbb{N}$ 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 $\widetilde{\tau} = \mathbb{I} \rightarrow \widehat{\tau}$. PEG nodes operate on these loop-lifted types.

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

Figure 6 gives the definition of the primitive functions $Prims = \{\phi, \theta_\ell, eval_\ell, pass_\ell\}$. 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 $eval_\ell$ and $pass_\ell$ make use of the function $monotonize_\ell$, whose definition is given in Figure 6. The $monotonize_\ell$ function transforms a sequence so that, once an indexed value is undefined, all following indexed values are undefined. The $monotonize_\ell$ function formalizes the fact that once a value is undefined at a given loop iteration, the value remains undefined at subsequent iterations.

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

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

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

$$[\![n]\!] = L(n)([\![C(n)]\!]) \qquad (5)$$

Equation 5 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 as a set of recursive equations to be solved. To guarantee that a unique solution exists, we impose some well-formedness constraints on PEGs.

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

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

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

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

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

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

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

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

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

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 6. 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, $\mathbb{I} = \mathcal{L} \to \mathbb{N}$, $\widehat{\tau} = \tau \cup \{\bot\}$, and $\widetilde{\tau} = \mathbb{I} \to \widehat{\tau}$.

DEFINITION 1. *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.

THEOREM 2. *If a PEG is well-formed, then for each node n in the PEG there is a unique semantic value $[\![n]\!]$ satisfying Equation 5.*

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 the traditional operational semantics for loops.

E-PEG Semantics. An E-PEG is a PEG with a set of equalities E between nodes. An equality between n and n' denotes value equality: $[\![n]\!] = [\![n']\!]$. The set E forms an equivalence relation, which in turn partitions the PEG nodes into equivalence classes.

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 • denotes "don't care":

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

One of the benefits of having a well-defined semantics for primitive PEG functions is that we can reason formally about these functions. In particular, using our semantics, we have proved all the axioms presented in this paper.

5.2 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 [5]. In our PEG representation, referential transparency can be formalized as follows:

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

This property follows from the definition in Equation (5), 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, 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.

5.3 Translating between CFGs and PEGs

To incorporate PEGs and E-PEGs into our approach, we have developed the $\mathrm{ConvertToIR}$ and $\mathrm{ConvertToCFG}$ functions from Figure 5. We give only a brief overview of these algorithms, with more details in a technical report [34].

Transforming a CFG into a PEG. The key challenge in constructing a PEG from a CFG is determining the branching structure of the CFG. We perform this task with a function that, given paths from one CFG basic block to another, produces a PEG expression with ϕ, *eval*, and *pass* operations, specifying which path is taken under which conditions. We use this to determine the break conditions of loops and the general branching structure of the CFG. We also identify nesting depth, entry points, and back-edges of loops to construct θ nodes. We piece these components together with the instructions of each basic block to produce the PEG. Lastly, we apply some basic simplifications to remove conversion artifacts. Our conversion algorithm from CFG to PEG handles arbitrary control flow, including irreducible CFGs (by first converting them so they become reducible).

Transforming a PEG into a CFG. Intuitively, our algorithm first groups parts of the PEG into sub-PEGs; then it recursively converts these sub-PEGs into CFGs; and finally it combines these sub-CFGs into a CFG for the original PEG. The grouping is done as follows: ϕ nodes are grouped together whose conditions are equal, thus performing branch fusion; θ nodes are grouped together that have equal *pass* conditions, thus performing loop fusion. The pure mathematical nature of PEGs makes it easy to identify when two conditions are equal, which makes branch/loop fusion simple to implement.

PEGs follow the insight from Click of separating code placement issues from the IR [9]. In particular, PEGs do not represent code placement explicitly. Instead, placement is performed during the translation back to a CFG. As a result, the translation from a CFG to a PEG and back (without any saturation) ends up performing a variety of optimizations: Constant Propagation, Copy Propagation, Common Subexpression Elimination, Partial Redundancy Elimination, Unused Assignment Elimination, Unreachable Code Elimination, Branch Fusion, Loop Fusion, Loop Invariant Branch Hoisting/Sinking, Loop Invariant Code Hoisting/Sinking, and Loop Unswitching.

6. The Peggy Instantiation

We have instantiated our approach in a Java bytecode optimizer called Peggy. Recall from Figure 5 that an instantiation of our approach consists of three components: (1) an IR where equality reasoning is effective, along with the translation functions $\mathrm{ConvertToIR}$ and $\mathrm{ConvertToCFG}$, (2) a saturation engine $\mathrm{Saturate}$, and (3) a global profitability heuristic $\mathrm{SelectBest}$. We now describe how each of these three components work in Peggy.

6.1 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

---

Function $\mathrm{Saturate}(peg : PEG, A : \text{set of analyses}) : EPEG$

1: **let** $epeg = \mathrm{CreateInitialEPEG}(peg)$
2: **while** $\exists (p, f) \in A, subst \in S \ . \ subst = \mathrm{Match}(p, epeg)$ **do**
3:     $epeg := \mathrm{AddEqualities}(epeg, f(subst, epeg))$
4: **return** $epeg$

Figure 7. Peggy's Saturation Engine. We use S to denote the set of all substitutions from pattern nodes to E-PEG nodes.

---

we have simply chosen one where all objects are put into a single summarization object σ.

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). This forces the observable state at the point where an exception is thrown to be preserved by our optimization process. Furthermore, to preserve Java semantics, Peggy does not perform any optimizations across `try`/`catch` boundaries or synchronization boundaries.

6.2 Saturation Engine

The saturation engine's purpose is to repeatedly dispatch equality analyses. In our implementation an equality analysis is a pair (p, f) where p is a trigger, which is an E-PEG pattern with free variables, and f is a callback function that should be run when the pattern p is found in the E-PEG. While running, the engine continuously monitors the E-PEG for the presence of the pattern p, 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 7 shows the pseudo-code for Peggy's saturation engine. The call to $\mathrm{CreateInitialEPEG}$ on the first line 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 $\mathrm{AddEqualities}$ to add the equalities computed by the analysis into the E-PEG.

A remaining concern in Figure 7 is how to efficiently implement the existential check on line 2. To address this challenge, we adapt techniques from the AI community. In particular, the task of finding the matches on line 2 is similar to the task of determining when rules fire in rule-based expert or planning systems. These systems make use of an efficient pattern matching algorithm called the Rete algorithm [17]. Intuitively, the Rete algorithm stores the state of partially completed matches in a set of FSMs, and when new information is added to the system, it transitions the appropriate FSM. Our saturation engine uses an adaptation of the Rete algorithm for the E-PEG domain to efficiently implement the check on line 2.

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 4 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.

6.3 Global Profitability Heuristic

Peggy's SelectBest function uses a Pseudo-Boolean solver called Pueblo [32] 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 well-formed computation. 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 state that: (1) we must select the equivalence class of the return value; (2) if an equivalence class is selected, we must select one of its nodes; (3) if a node is selected, we must select its children's equivalence classes; (4) the chosen PEG is well-formed.

The cost model that we use assigns a constant cost C_n to each node n. In particular, $C_n = basic\_cost(n) \cdot k^{depth(n)}$, where $basic\_cost(n)$ accounts for how expensive n is by itself, and $k^{depth(n)}$ accounts for how often n is executed. We use $depth(n)$ to denote the loop nesting depth of n, and k is simply a constant, which we have chosen as 20. Using C_n, the objective function we want to minimize is $\sum_{n \in N} B_n \cdot C_n$, where N is the set of nodes in the E-PEG, and B_n is the pseudo-boolean variable for node n. Peggy asks Pueblo to minimize this objective function subject to the well-formedness constraints described above. The nodes assigned 1 in the solution that Pueblo returns are selected to form the PEG that SelectBest returns. This PEG is the lowest-cost PEG that is represented in the E-PEG, according to our cost model.

7. 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 and time (Section 7.1), it is effective at discovering both simple and intricate optimization opportunities (Section 7.2), and it is effective at performing translation validation (Section 7.3).

7.1 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).

7.2 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 8(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 8(b) shows the optimizations that we observed fall out from our approach using equality analyses 1 through 6, and Figure 8(c) shows optimizations that we observed fall out from our approach using equality analyses 1 through 7.

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 19) 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.

In terms of generated-code quality, Peggy generates code whose performance is comparable to the code generated by Soot's intraprocedural optimizations, 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. However, intraprocedural optimizations on Java bytecode generally produce only small gains (on the order of a few percent). In the rare cases where significant gains are to be had from intraprocedural optimizations, Peggy excelled. 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. We

(a) Equality Analyses	Description
1. Built-in E-PEG axioms	Axioms from Section 5.1 stating properties of primitive PEG nodes (ϕ, θ, $eval$, $pass$)
2. Basic Arithmetic Axioms	Axioms stating properties of arithmetic operators like $+$, $-$, $*$, $/$, `<<`, `>>`
3. Constant Folding	Equates a constant expression with its constant value
4. Java-specific Axioms	Axioms stating properties of Java-specific operators like field and array accesses
5. Tail-recursion Elimination	Replaces the body of a tail-recursive procedure with a loop
6. Method Inlining	Inlining based on intraprocedural class analysis to disambiguate dynamic dispatch
7. Domain-specific Axioms	User-provided axioms about certain application domains (optional)

(b) Optimizations	Description
8. Constant Propagation and Folding	Traditional Constant Propagation and Folding
9. Algebraic Simplification	Various forms of traditional algebraic simplifications
10. Peephole Strength Reduction	Various forms of traditional peephole optimizations
11. Array Copy Propagation	Replace read of an array element by the expression it was previously assigned
12. CSE for Array Elements	Remove redundant array accesses
13. Loop Peeling	Pulls the first iteration of a loop outside of the loop
14. LIVSR	Optimization described in Section 2, namely Loop-induction-variable Strength Reduction
15. Interloop Strength Reduction	Optimization described in Section 3
16. Entire-loop Strength Reduction	Transforms entire loop into one operation, e.g. loop doing i incrs becomes "plus i"
17. Loop-operation Factoring	Factors expensive operations out of a loop, for example multiplication
18. Loop-operation Distributing	Distributes an expensive operation into a loop, where it cancels out with another operation
19. Partial Inlining	Inlines part of a method in the caller but keeps the call for the rest of the computation

(c) Domain-specific Optimizations	Description
20. Domain-specific LIVSR	LIVSR, but with domain operations like matrix/vector addition and multiply
21. Domain-specific code hoisting	Code hoisting based on domain-specific invariance axioms
22. Domain-specific Redundancy Removal	Removes redundant computations based on domain axioms
23. Temporary Object Removal	Removes temporary objects created by calls to matrix/vector libraries
24. Specialization of Math Libraries	Specializes vector/matrix algorithms based on, for example, the size of the vector/matrix
25. Design-pattern Optimizations	Removes overhead of certain design patterns, like the indirection or interpreter pattern
26. Method Outlining	Replaces code by call to a method performing the same computation, but more efficiently
27. Specialized Redirection	Replaces method call with call to a more efficient version based on the calling context

Figure 8. Optimizations performed by Peggy

conjecture that interprocedural optimizations can be expressed as equality analyses, and leave this exploration to future work.

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 23 in Figure 8).

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 `contains`(e) and a call to `indexOf`(e) on the true side of the branch. Unfortunately, `contains` and `indexOf` would perform the same linear search, which makes this clean way of writing the code inefficient. Using the equality axiom $l.\texttt{contains}(e) = (l.\texttt{indexOf}(e) \neq -1)$, Peggy can convert the clean code into the hand-optimized code that programmers typically write, which stores `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.

7.3 Translation Validation

We used Peggy to perform translation validation for the Soot optimizer [36]. In particular, we used Soot to optimize a set of benchmarks with all of its intraprocedural optimizations turned on. The benchmarks included SpecJVM, along with other programs, comprising a total of 3,416 methods. After Soot finished compiling, for each method we asked Peggy's saturation engine to show that the original method was equivalent to the corresponding method that Soot produced. The engine was able to show that 98% of methods were compiled correctly.

Among the cases that Peggy was unable to validate, we found three methods that Soot optimized *incorrectly*. In particular, Soot

incorrectly pulled statements outside of an intricate loop, transforming a terminating loop into an infinite loop. It is a testament to the power of 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.

8. 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 [24, 18, 4, 16]. 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 3.

Our approach was inspired by Denali [21], 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 [6], ASF+SDF [37], the ML compilation system of Visser *et al.* [38], and Stratego [7]. 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 [40, 41], automated techniques for generating good sequences [10, 1, 22], manual techniques for combining analyses and optimizations [8], and automated techniques for the same purpose [23]. 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 [30, 25, 43, 31], 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 [11, 2, 35, 19, 15, 39, 9, 33, 29]. 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 [11], gated SSA [35] and thinned-gated SSA [19]. The μ function from gated SSA is similar to our θ function, and the η function is similar to our $eval/pass$ pair. However, unlike PEGs, all these variants of SSA are tried to an underlying CFG representation.

Program Dependence Graphs [15] and the Program Dependence Web [28] represent control information by grouping together operations that execute in the same control region. However, these IRs are still statement based, and maintain explicit control edges.

Like PEGs, the Value Dependence Graph [39] (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. These specialized nodes, combined with simple axioms about them, allow us to perform intricate optimizations across loops, such as the optimization from Section 3.

Dependence Flow Graphs [29] (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.

Dataflow Languages. Our PEG intermediate representation is related to the broad area of dataflow languages [20]. The most closely related is the Lucid programming language [3], 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 [13], with its E-graph data structure for representing equalities [27]. 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 [26].

Execution Indices. Execution indices identify the state of progress of an execution [14, 42]. 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.

9. 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.

References

[1] 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.

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

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

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

[5] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[6] 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.

[7] 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.

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

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

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

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

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

[13] 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.

[14] E. Dijkstra. Go to statement considered harmful. pages 27–33, 1979.

[15] 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.

[16] 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.

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

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

[19] P. Havlak. Construction of thinned gated single-assignment form.

[20] In *Workshop on Languages and Compilers for Parallel Computing*, 1993.

[20] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[21] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *PLDI*, June 2002.

[22] L. Torczon K. D. Cooper, D. Subramanian. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, pages 7–22, 2002.

[23] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *POPL*, January 2002.

[24] Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS*, 1987.

[25] G. Necula. Translation validation for an optimizing compiler. In *PLDI*, June 2000.

[26] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[27] 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.

[28] 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.

[29] K. Pengali, M. Beck, and R. Johson. Dependence flow graphs: an algebraic approach to program dependencies. In *POPL*, January 1991.

[30] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.

[31] H. Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, July 1978.

[32] H. Sheini and K. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:61–96, 2006.

[33] B. Steffen, J. Knoop, and O. Ruthing. The value flow graph: A program representation for optimal program transformations. In *European Symposium on Programming*, 1990.

[34] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Translating between PEGs and CFGs. Technical Report CS2008-0931, University of California, San Diego, November 2008.

[35] P. Tu and D. Padua. Efficient building and placing of gating functions. In *PLDI*, June 1995.

[36] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON*, 1999.

[37] 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.

[38] E. Visser, Z. Benaissa, and A Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, 1998.

[39] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *POPL*, 1994.

[40] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *PPOPP*, 1990.

[41] 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.

[42] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI*, June 2008.

[43] 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.