Generating Compiler Optimizations from Proofs [*]

Ross Tate Michael Stepp Sorin Lerner

University of California, San Diego
{rtate,mstepp,lerner}@cs.ucsd.edu

Abstract

We present an automated technique for generating compiler optimizations from examples of concrete programs before and after improvements have been made to them. The key technical insight of our technique is that a proof of equivalence between the original and transformed concrete programs informs us which aspects of the programs are important and which can be discarded. Our technique therefore uses these proofs, which can be produced by translation validation or a proof-carrying compiler, as a guide to generalize the original and transformed programs into broadly applicable optimization rules.

We present a category-theoretic formalization of our proof generalization technique. This abstraction makes our technique applicable to logics besides our own. In particular, we demonstrate how our technique can also be used to learn query optimizations for relational databases or to aid programmers in debugging type errors.

Finally, we show experimentally that our technique enables programmers to train a compiler with application-specific optimizations by providing concrete examples of original programs and the desired transformed programs. We also show how it enables a compiler to learn efficient-to-run optimizations from expensive-to-run super-optimizers.

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

General Terms Languages, Performance, Theory

1. Introduction

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

The broad focus of our recent work in this space has been to provide tools that help programmers with the difficulties of writing compiler optimizations. In this context, we have done work on automatically proving correctness of transformation rules, on generating provably correct dataflow analyses, and on mitigating the complexity of how transformation rules interact.

Despite all these advances, programmers who wish to implement optimizations often still have to write down the transformation rules that make up the optimizations in the first place. This task is error prone and tedious, often requiring multiple iterations to get the rules to be correct. It also often involves languages and interfaces that are not familiar to the programmer: either a language for rewrite rules that the programmer needs to become familiar with, or an interface in the compiler that the programmer needs to learn. These difficulties raise the barrier to entry for non-compiler-experts who wish to customize their compiler.

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

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

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

[*] This work was supported by NSF CAREER grant CCF-0644306.

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01. . . $5.00

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

Our contributions can therefore be summarized as follows:

- We present a technique for generating optimization rules from optimization instances by generalizing proofs of correctness and the objects that these proofs manipulate (Section 2).
- We formalize our technique as a category-theoretic framework that can be instantiated in various ways by defining a few key categories and operations on them (Section 3). The general nature of our formalism makes our technique broadly applicable.
- We illustrate the generality of our framework by instantiating it not only to compiler optimizations (Section 4), but to other domains (Section 5). In the database domain, we show that proof generalization can be used to learn efficient query optimizations. In the type-inference domain, we show that proof generalization can be used to improve type-error messages.
- We used an implementation of our approach in the Peggy compiler infrastructure [22] to validate the following three hypotheses about our approach: (1) our approach can learn complex optimizations not performed by `gcc -O3` from simple examples provided by the programmer (2) it can learn optimizations from Peggy's expensive super-optimization phase (3) it can learn optimizations that are useful on code that it was not trained on.

2. Overview

The goal that we are trying to achieve is to generalize optimization instances into optimization rules. The key to our approach is to use a proof of correctness of the optimization instance as a guide for generalization. The proof of correctness tells us precisely what parts of the program mattered and how, so that we can generalize them in a way that retains the validity of the proof structure.

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

In this section, we start by giving some examples of proof-based generalization (Section 2.1), explain some of the challenges behind generalization (Section 2.2), give an overview of our technique (Section 2.3), and finally describe a way of decomposing optimizations we generate into smaller independent ones (Section 2.4).

2.1 Generalization examples

Figure 1 shows an example of how our approach works. We describe the process at a high-level, and then describe the details of each step. At a high-level, we start with two concrete programs, presenting an example of what the desired transformation should do – parts (a) and (b); we convert these programs into our own intermediate representation – parts (c) and (d); we then prove that the two programs are equivalent, a process known as translation validation – part (e); from the proof of equivalence we then generalize into optimization rules – parts (f) and (g) show two possible generalizations. We now go through each of these steps in detail.

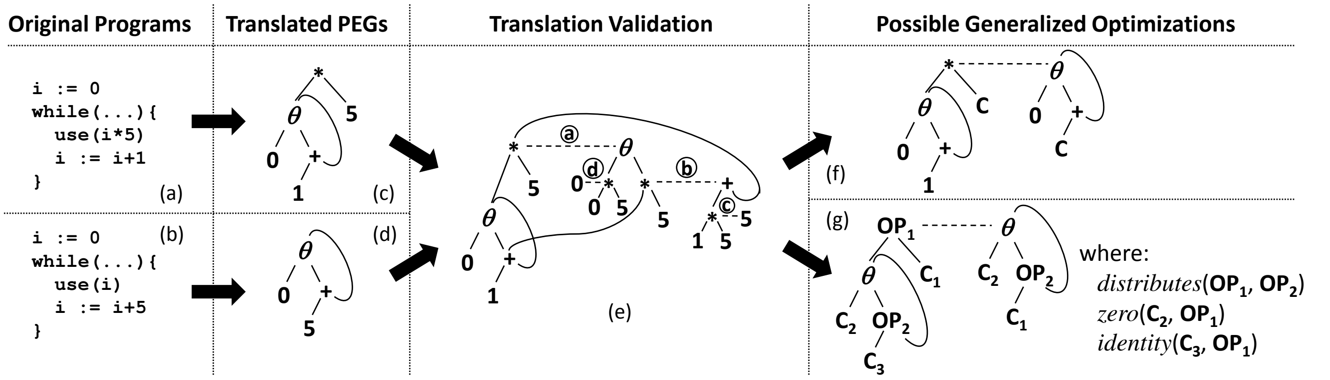
The optimization that is illustrated in parts (a) and (b) of Figure 1 is called loop-induction variable strength reduction (LIVSR). The optimization essentially replaces a multiplcation with an addition inside of a loop.

As we will show in Section 3, our approach is general and can be applied to many kinds of intermediate representations, and even to domains other than compiler optimizations. However, to make things concrete for our examples, we will use the PEG and E-PEG intermediate representations from our previous work on the Peggy compiler [22] (this is also the representation we use in our implementation and evaluation).

Part (c) of Figure 1 shows a Program Expression Graph (PEG) representing the use of `i*5` in the code of part (a). A PEG contains nodes representing operators (for example "+" and "$*$"), and edges representing which nodes are arguments to which other nodes. The arguments to a node are displayed below the node. The top of the PEG contains a multiply node representing the multiply from `i*5`. The θ node represents the value of `i` inside the loop. In particular, the θ node states that the initial value of `i` (the left child of θ) is 0, and that the next value of `i` (the right child of θ) is 1 plus the current value of `i`. Similarly, part (d) of the figure shows the PEG for the use of `i` in part (b).

Now that we have represented the two programs in our intermediate reprensentation, we must prove that they are equivalent. We do so using an E-PEG, which is a PEG augmented with equality information between nodes. Graphically, we represent two nodes being equal with a dotted edge between then, although in our implementation we represent equality by storing the equivalence classes of nodes. Part (e) of Figure 1 is an E-PEG, constructed by applying four equality axioms to the original PEG: edge ⓐ is added by applying the axiom $\theta(x, y) * z = \theta(x * z, y * z)$; edge ⓑ is added by applying the axiom $(x + y) * z = x * z + y * z$; edge ⓒ is added by applying the axiom $1 * x = x$; and edge ⓓ is added by applying the axiom $0 * x = 0$. This E-PEG represents many different versions of the original program, depending on how we choose to compute each equivalence class. By picking θ to compute the $\{*, \theta\}$ equivalence class, and $+$ to compute the $\{*, +\}$ equivalence class, we get the PEG from Figure 1(d), and as a result, the E-PEG therefore shows that the PEGs from parts (c) and (d) are equivalent.

In our Peggy compiler, there are two ways of arriving at the E-PEG in Figure 1(d). The first is as described above: we convert two programs into PEGs and then repeatedly add equalities until the result of the two programs become equal. The other approach is to start with just an original program and PEG, say the ones from Figure 1(a) and 1(c), and construct an E-PEG by repeatedly applying axioms to infer equalities. A profitability heuristic can then select which PEG is the most efficient way of computing the results in the E-PEG, which in our case would be the PEG from Figure 1(d).



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

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

Figure 1. Loop Induction Variable Strength Reduction (LIVSR)

Whichever approach is used, the starting point of generalization is the E-PEG from Figure 1(e), which represents a proof that the PEGs from Figure 1(c) and Figure 1(d) are equivalent. In particular, edges ⓐ through ⓓ in the E-PEG represent the steps of the equivalence proof.

Our goal is to take the conclusion of the proof – in this case edge ⓐ – and determine how one can generalize the E-PEG so that the proof encoded in the E-PEG is still valid. Figures 1(f) and 1(g) show two possible generalized optimizations that can result from this process. We represent a generalized optimization as an E-PEG which contains a single equality edge, representing the conclusion of the proof. There are two ways of interpreting such E-PEGs. One is that it represents a transformation rule, with the single equality edge representing the transformation to perform. The direction of the rule is determined by which of the two programs in the instance was the original, and which was the transformed.

Another way to interpret these rules is that they represent equality analyses to be used in our Peggy optimizer [22]. Optimizations in Peggy take the form of equality analyses that infer equality information in an E-PEG. Starting with some original program, Peggy converts the program to a PEG, and then repeatedly applies equality analyses to construct an E-PEG. It then uses a global profitability heuristic to select the best PEG represented in the computed E-PEG, and converts this PEG back to a program, which is the result of optimization. Section 7 will show that our generated optimizations, when used as equality analyses, make Peggy faster while still producing the same results. Furthermore, as equality analyses, our generated optimizations will just infer additional information from which the profitability heuristic can choose. We therefore do not have to worry about whether a generated optimization will always be just as profitable as the optimization instance.

Figure 1(f) shows a generalization where the constant 5 has been replaced with an arbitrary constant C. The key observation is that the particular choice of constant does not affect the proof – if we have a proof of LIVSR for 5, the same proof holds for an arbitrary constant. Note that PEGs abstract away the details of the control flow graph. As a result the generalizations of Figure 1(f) could be applicaple to a PEG even if there were many other nodes in the PEG representing various kinds of loops or statements not affecting the induction variable being optimized.

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

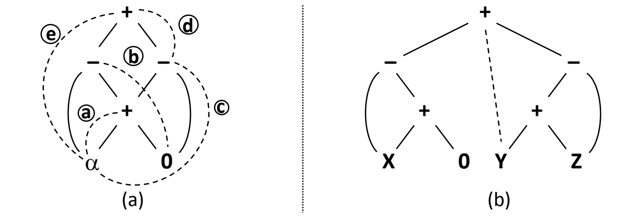


Figure 2. Example showing the need for splitting

The choice of axioms is what makes the difference between the above two generalizations. Figure 1(g) results from a proof expressed with more general axioms. Instead of the axiom $(x + y) * z = x * z + y * z$, the proof uses:

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

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

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

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

2.2 Challenge with obtaining the most general form

Looking at the LIVSR example, one may think that generalization is as simple as replacing all nodes and operators in an E-PEG with meta-variables, and then constraining the meta-variables

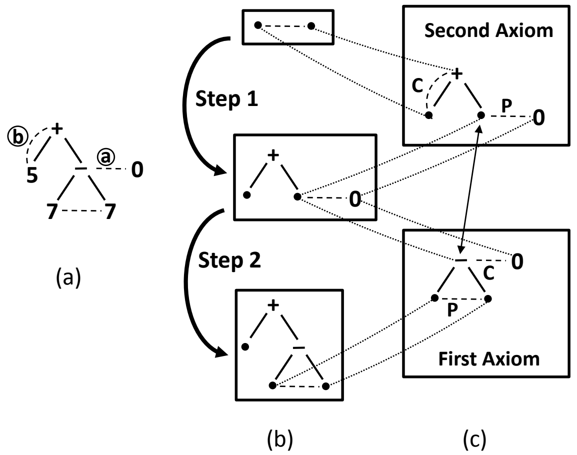
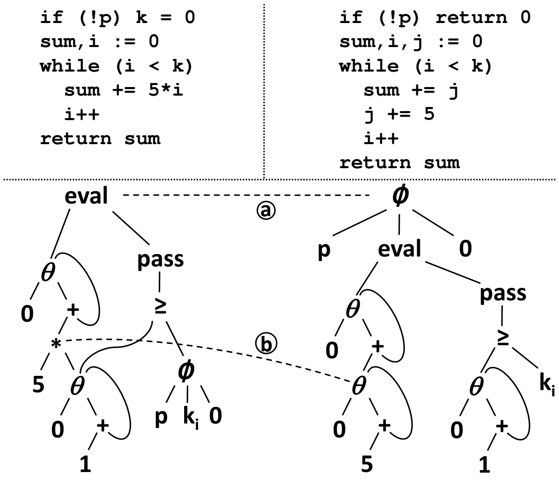


Figure 3. Example showing how generalization works



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

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

Figure 4. One E-PEG with two conceptual optimizations

based on the axioms that were applied. Although this approach is very simple, it does not always produce the most general optimization rule for a given proof. Consider for example the E-PEG from Figure 2(a), where α is some PEG expression. Edge ⓐ is produced by axiom $x+0 = x$; edge ⓑ by $x-x = 0$; edge ⓒ by $(x+y)-y = x$; edge ⓓ by $0 + x = x$; and edge ⓔ by transitivity of edges ⓒ and ⓓ. This E-PEG therefore represents a proof that the plus expression at the top is equivalent to α. If we replace nodes with meta-variables and constrain the meta-variables based on axiom applications, one would simply generalize α to a meta-variable. However, the most general optimization rule from the proof encoded in Figure 2(a) is shown in Figure 2(b). The key difference is that by duplicating the shared "+" node, one can constrain the arguments of the two new plus nodes differently. However, because PEGs can contain cycles, one cannot simply duplicate every node that is shared, as this would lead to an infinite expansion. The main challenge then with getting the most general form is determining precisely how much to split.

2.3 Our Approach

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

We illustrate our approach on a very simple example so that we can show all the steps. Consider the E-PEG of Figure 3(a), where two axioms have been applied to determine equality edges ⓐ and ⓑ. The axioms are shown in Figure 3(c), with each axiom being an E-PEG where one edge has been labeled "P" for "Premise", and one edge has been labeled "C" for "Conclusion". The • in the axioms represent meta-variables to be instantiated. The first axiom states $x - x = 0$, and the second axiom states $x + 0 = x$.

Our process is shown in Figure 3(b). We start at the top with a single equality edge representing the conclusion of the proof, and then work our way downward by applying the proof in reverse order: in step 1 we apply the second axiom backward, and then in step 2 we apply the first axiom backward. Each time we apply an axiom backward, we create and/or constrain E-PEG nodes in order to allow that axiom to be applied. Figure 3 shows using fine-dotted edges how the "Premise" and "Conclusion" edges of the axioms map onto the E-PEGs being constructed. For example, note that in step 1, when the second axiom is applied backward, we remove the final conclusion edge, and instead replace it with an E-PEG that essentially represents • + 0.

There is an alternate way of viewing our approach. In this alternate view, we instantiate all the axioms that have been applied in the proof with fresh meta-variables, and then use unification to stitch these freshly instantiated axioms together so that they connect in the right way to make the proof work. With this view in mind, we show in Figure 3 how the first and second axioms would be stitched together using a bi-directional arrow.

This section has only given an overview of how our approach works. Sections 3 and 4 will formalize our approach using category theory and revisit the above example in much more detail.

2.4 Decomposition

Even though our algorithm finds the most general transformation rule for a given proof, the produced rule may still be too specific to be reused. This can happen if the input-output example has several conceptually different optimizations happening at the same time. Consider for example the optimization instance shown in Figure 4. The top of the Figure shows the original and transformed code. There are two independent high-level optimizations. The first is LIVSR, which replaces the `i*5` with a variable `j` that is incremented by `5` each time around the loop; the second is specialization for the true case of the `if(!p)` branch, so that it immediately returns.

The corresponding E-PEG is shown at the bottom of the Figure. The E-PEG does not show all the steps – instead it just displays the final equality edge ⓐ, and an additional edge ⓑ, which we will discuss shortly. This E-PEG uses three new kinds of nodes (all of which were introduced in [22]): $\phi(p, e_t, e_f)$ evaluates to e_t if p is true and e_f otherwise; $\mathsf{eval}(s, i)$ returns the i^{th} element from the sequence s; and $\mathsf{pass}(s)$ returns the index of the first element in the boolean sequence s that is true. The eval/pass operators are used to extract the value of a variable *after* a loop. Consider for example the top-leftmost θ in Figure 4, which represents the sequence of values that the variable `sum` takes throughout the loop. The pass node computes the index of the last loop iteration, and the result of pass is used to index into the sequence of values of `sum`.

In the E-PEG, the two optimizations manifest themselves as follows: LIVSR happens using steps similar to those from Figure 1 on the PEG rooted at the "$*$" node (producing edge ⓑ); the specialization optimization happens by pulling the ϕ node up through the $\geq$, pass and eval nodes (producing edge ⓐ). Each of these optimizations takes several axiom applications to perform, introducing various temporary nodes that are not shown in Figure 4.

If we simply apply the generalization algorithm outlined in Section 2.3, we will get a single rule (although generalized) that applies the two optimizations together. However, these two optimizations are really independent of each other in the sense that each can be applied fruitfully in cases where the other does not apply. Thus, in order to learn optimizations that are more broadly applicable, we further *decompose* optimizations that have been generalized into smaller optimizations. One has to be careful, however, because decomposing too much could just produce the axioms we started with.

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

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

In the example of Figure 4, we would find one such equality edge, namely edge ⓑ. As a result, our decomposition algorithm would perform two generalizations. The first one starts at the conclusion ⓐ, going backwards from there, but stops when edge ⓑ is reached (i.e. edge ⓑ is treated as an assumption). This would produce a branch-lifting optimization. Separately, our decomposition algorithm would perform a generalization starting with ⓑ as the conclusion, which would essentially produce the LIVSR optimization.

3. Formalization

Having seen an overview of how our approach works, we now give a formal description of our framework for generalizing proofs using category theory. The generality of our framework not only gives us flexibility in applying our algorithm to the setting of compiler optimizations by allowing us to choose the domain of axioms and proofs, but it also makes our framework applicable to settings beyond compilers optimizations. After a quick overview of category theory (Section 3.1), we show how axioms (Section 3.2) and inference (Section 3.3) can be encoded in category theory. We then define what a generalization is (Section 3.4), and finally show how to construct the most general one (Section 3.5).

3.1 Overview of category theory

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

Given two objects $\mathcal{A}$ and $\mathcal{B}$ in a category, the notation $f : \mathcal{A} \to \mathcal{B}$ indicates that f is a morphism from $\mathcal{A}$ to $\mathcal{B}$. This same information is displayed graphically as follows:

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

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

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

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

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

Although there are many kinds of categories, we will be focusing on structured sets. In such categories, objects are sets with some additional structure and the morphisms are structure-preserving functions. The **Set** category mentioned previously is the simplest example of such a category, since there is no structure imposed on the sets. A more structured example is the **Rel** category of binary relations. An object in this category is a binary relation (represented, say, as a set of pairs), and a morphism is a relation-preserving function. In particular, the morphism $f : R_1 \to R_2$ is a function from the domain of R_1 to the domain of R_2 satisfying: $\forall x, y \;.\; x \; R_1 \; y \implies f(x) \; R_2 \; f(y)$. Informally, there are also categories of expressions, even recursive expressions, and the morphisms are substitutions of variables. As shown in more detail in Section 4, in the setting of compiler optimizations, we will use a category in which objects are E-PEGs and morphisms are substitutions.

3.2 Encoding axioms in category theory

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

$$\boxed{\begin{array}{cc} x & y \\ y & z \end{array}} \xrightarrow{trans} \boxed{\begin{array}{cc} x & y \\ y & z \\ x & z \end{array}} \tag{1}$$

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

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

$$\begin{tikzcd}
\boxed{\begin{matrix} x & y \\ y & z \end{matrix}} \arrow[r, "\mathit{trans}"] \arrow[d, "f"'] & \boxed{\begin{matrix} x & y \\ y & z \\ x & z \end{matrix}} \arrow[dl, "f'"] \\
\mathcal{A} &
\end{tikzcd} \tag{2}$$

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

Similarly, our E-PEG axioms can be encoded as identity-carried morphisms which simply add an equality. The axiom $x * 0 = 0$ is encoded as the identity-carried morphism from the E-PEG $x * y$, with y equivalent to 0, to the E-PEG $x * y$, with y equivalent to 0 *and* $x * y$ equivalent to 0. Thus, an E-PEG satisfies this axiom if for every $x * y$ node where y is equivalent to 0, the $x * y$ node is also equivalent to 0. More details on our E-PEG category can be found in Section 4.

3.3 Encoding inference in category theory

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

$$\begin{tikzcd}
\boxed{\begin{matrix} x & y \\ y & z \end{matrix}} \arrow[r, "\mathit{trans}"] \arrow[d, "{\begin{matrix} x \mapsto a, \\ y \mapsto b, \\ z \mapsto c \end{matrix}}"'] & \boxed{\begin{matrix} x & y \\ y & z \\ x & z \end{matrix}} \\
\boxed{\begin{matrix} a & b \\ b & c \\ c & d \end{matrix}} &
\end{tikzcd} \tag{3}$$

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

$$\begin{tikzcd}
\boxed{\begin{matrix} x & y \\ y & z \end{matrix}} \arrow[r, "\mathit{trans}"] \arrow[d, "{\begin{matrix} x \mapsto a, \\ y \mapsto b, \\ z \mapsto c \end{matrix}}"'] & \boxed{\begin{matrix} x & y \\ y & z \\ x & z \end{matrix}} \arrow[d, "{\begin{matrix} x \mapsto a, \\ y \mapsto b, \\ z \mapsto c \end{matrix}}"] \\
\boxed{\begin{matrix} a & b \\ b & c \\ c & d \end{matrix}} \arrow[r] & \boxed{\begin{matrix} a & b \\ b & c \\ c & d \\ a & c \end{matrix}}
\end{tikzcd} \tag{4}$$

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

Definition 1 (Pushout). *A commuting square* $[\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}]$ *is said to be a* pushout square *if for any object* $\mathcal{E}$ *that makes* $[\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{E}]$ *a commuting square, there exists a unique morphism from* $\mathcal{D}$ *to* $\mathcal{E}$ *such that the following diagram commutes:*

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

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

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

Going back to our example, to encode the application of the transitivity axiom, we require that the commuting square in diagram (4) be a pushout square. The pushout square property applied to diagram (4) ensures that, for any relation $\mathcal{E}$ such that $a\mathcal{E}c$, there will be a morphism from the bottom right object in the diagram (call it $\mathcal{D}$) to $\mathcal{E}$, meaning that $\mathcal{E}$ contains as much or more information than $\mathcal{D}$, which in turn means that $\mathcal{D}$ encodes the least relation that includes (a, c). This is exactly the result we want from applying transitivity on our example.

Furthermore, we can obtain the bottom right coner of diagram (4) by taking the pushout of diagram (3). Thus, inference is the process of repeatedly identifying points where axioms can apply and pushing out to add the learned information. This produces a sequence of pushout squares whose bottom edges all chain together. For example, in the diagram below, $\mathit{app_1}$ states where to apply $\mathit{axiom_1}$ in $\mathcal{E}_0$, and the pushout $\mathcal{E}_0 +_{\mathcal{A}_1} \mathcal{C}_1$ produces the result $\mathcal{E}_1$; in the second step, $\mathit{app_2}$ states where to apply $\mathit{axiom_2}$ in $\mathcal{E}_1$, and the pushout $\mathcal{E}_1 +_{\mathcal{A}_2} \mathcal{C}_2$ produces $\mathcal{E}_2$; this process can continue to produce an entire sequence $\mathcal{E}_i$, where each $\mathcal{E}_i$ encodes more information than the previous one.

$$\begin{tikzcd}
\mathcal{A}_1 \arrow[r, "\mathit{axiom}_1"] \arrow[d, "\mathit{app}_1"] & \mathcal{C}_1 \arrow[dr] & \mathcal{A}_2 \arrow[r, "\mathit{axiom}_2"] \arrow[d, "\mathit{app}_2"] & \mathcal{C}_2 \arrow[dr] & & \cdots \\
\mathcal{E}_0 \arrow[rr] & & \mathcal{E}_1 \arrow[rr] & & \mathcal{E}_2 &
\end{tikzcd} \tag{5}$$

In the E-PEG setting, each $\mathcal{E}_i$ will be an E-PEG, and each axiom application will add an equality edge. The entire sequence above constitutes a proof in our formalism: it encodes both the axioms being applied (axiom_1, axiom_2, etc.), how they are applied (app_1, app_2, etc.), and the sequence of conclusions that are made ($\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{E}_2$, etc.). Traditional tree-style proofs (such as derivations) can be linearized into our categorical encoding of proofs (see Section 6 for more details on how this can be done).

3.4 Defining generalization in category theory

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

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

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

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

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

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

Definition 2 (Pullback). *A commuting square* $[\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}]$ *is said to be a* pullback square *if for any object* $\mathcal{E}$ *that makes* $[\mathcal{E}, \mathcal{B}, \mathcal{C}, \mathcal{D}]$ *a commuting square, there exists a unique morphism from* $\mathcal{E}$ *to* $\mathcal{A}$ *such that the following diagram commutes:*

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

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

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

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

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

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

A generalization of $\mathcal{E}$ is an object $\mathcal{G}$ with a morphism *gen* : $\mathcal{G} \to \mathcal{E}$ (see diagram (6) below). A generalization of *app* is a morphism $app_{\mathcal{G}} : \mathcal{A} \to \mathcal{G}$ with $app_{\mathcal{G}}; gen = app$. We apply *axiom* to the generalized application $app_{\mathcal{G}}$ by taking the pushout to produce $\mathcal{G}'$. Lastly, we want our property to hold in $\mathcal{G}'$ for the same reason that it holds in $\mathcal{E}'$; that is, any information added by *axiom* to make the property *prop* hold in $\mathcal{E}'$ should also make the property hold in $\mathcal{G}'$. We enforce this by requiring an additional morphism $prop_{\mathcal{G}} : \mathcal{P} \to \mathcal{G}'$. To summarize, then, a generalization of applying *axiom* via *app* to produce *prop* is an object $\mathcal{G}$ with morphisms *gen*, $app_{\mathcal{G}}$, and $prop_{\mathcal{G}}$ making the following diagram commute:

$$\begin{tikzcd}
& & & \mathcal{O} \arrow[dll] \arrow[dd] \\
\mathcal{A} \arrow[r, "axiom"] \arrow[dr, "app_{\mathcal{G}}"] \arrow[dd, "app"'] & \mathcal{C} \arrow[dr] \arrow[dd] & & \\
& \mathcal{G} \arrow[r] \arrow[dl, "gen"] & \mathcal{G}' \arrow[dl, dashed] & \mathcal{P} \arrow[l, "prop_{\mathcal{G}}"'] \arrow[dll, "prop"] \\
\mathcal{E} \arrow[r] & \mathcal{E}' & &
\end{tikzcd} \tag{6}$$

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

The above diagram defines a generalization for the *last* step of the inference process. A generalization for an entire sequence of steps – such as diagram (5) – is an initial $\mathcal{G}_0$ and a morphism *gen* from $\mathcal{G}_0$ to $\mathcal{E}_0$ with a sequence of generalized axiom applications such that the property holds in the final $\mathcal{G}_n$.

3.5 Constructing generalizations using category theory

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

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

Definition 3 (Pushout completion). *Given a diagram*

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

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

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

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

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

Our requirement for constructing generalizations is that, for every axiom $\mathcal{A} \xrightarrow{axiom} \mathcal{C}$, there is a pushout completion for any morphism f from $\mathcal{C}$ to any object. There is encouraging evidence that axioms with pushout completions are quite common. In particular, all of our PEG axioms satisfy this condition. More generally, all identity-carried morphisms in **Rel** or the category of expressions satisfy this condition. Furthermore, an accompanying technical report [21] shows how to loosen this condition in a way that allows *all* morphisms in **Set** and **Rel** to qualify as generalizable axioms.

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

$$\begin{tikzcd}
 & & & & \mathcal{O} \arrow[dll] \arrow[dd] \\
\mathcal{A} \arrow[rr, "\mathit{axiom}"] \arrow[dr] \arrow[dd, "\mathit{app}"'] & & \mathcal{C} \arrow[dr] \arrow[dd] & & \\
 & \mathcal{P}' \arrow[rr] \arrow[dl, dashed] & & \bar{\mathcal{P}} \arrow[dl, dashed] & \mathcal{P} \arrow[l] \arrow[dll, "\mathit{prop}"] \\
\mathcal{E} \arrow[rr] & & \mathcal{E}' & &
\end{tikzcd} \tag{7}$$

1. $\mathcal{O}$ is constructed by taking the pullback $\mathcal{C} \times_{\mathcal{E}'} \mathcal{P}$.
2. $\bar{\mathcal{P}}$ is constructed by taking the pushout $\mathcal{C} +_{\mathcal{O}} \mathcal{P}$. The pushout $\bar{\mathcal{P}}$ intuitively represents the unification of property $\mathcal{P}$ with the assumptions and conclusions of *axiom* (which are represented by $\mathcal{C}$). The dashed morphism from $\bar{\mathcal{P}}$ to $\mathcal{E}'$ is induced by the pushout property of $\bar{\mathcal{P}}$; it identifies how this unified structure fits in $\mathcal{E}'$.
3. $\mathcal{P}'$ is constructed by taking the pushout completion $\bar{\mathcal{P}} -_{\mathcal{A}} \mathcal{C}$. The pushout completion $\mathcal{P}'$ intuitively represents the information in $\bar{\mathcal{P}}$ but with the inferred conclusion of *axiom* removed. The dashed morphism from $\mathcal{P}'$ to $\mathcal{E}$ is induced by the pushout property of $\bar{\mathcal{P}}$; it identifies the minimal information necessary in $\mathcal{E}$ so that applying *axiom* produces property $\mathcal{P}$.

Let us return to the larger context of a chain of axioms rather than applying just one axiom. In diagram (7) above, $\mathcal{E}$ would be the result of an earlier axiom application. $\mathcal{P}'$ then identifies the property of $\mathcal{E}$ that needs to be generalized in that axiom application. This process of generalization can be repeated backwards through the chain of axioms until we arrive at the original $\mathcal{E}_0$ that the inference process started with. The generalized property of $\mathcal{E}_0$ at that point is then the best generalization of $\mathcal{E}_0$ so that the proof infers the property $\mathcal{P}$ we started with in diagram (7). The fact that this solution is the most general falls immediately from the pushout and pushout-completion properties of the construction. In particular, suppose that in diagram (7) there is another generalization $\mathcal{G}$, which essentially means that we merge diagrams (6) and (7) together. We need to show that there exists a morphism from $\mathcal{P}'$ to $\mathcal{G}$. First, the pushout property on $\bar{\mathcal{P}}$ induces a morphism from $\bar{\mathcal{P}}$ to $\mathcal{G}'$. Second, the pushout-completion property on $\mathcal{P}'$ induces the desired morphism from $\mathcal{P}'$ to $\mathcal{G}$.

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

4. E-PEG instantiation

We now show how to instantiate the parameterized algorithm from Section 3 with E-PEGs. The category of E-PEGs can be formalized in a straghtforward way using well established categories such as partial algebras **PAlg** and relations **Rel** [1]. The full formal description, however, is lengthy to expose, so we provide here a semi-formal description.

An object in the E-PEG category is simply an E-PEG, where an E-PEG is a set of possibly recursive expressions (think recursive abstract syntax trees), with equalities between some of the nodes in these expressions. These E-PEG expressions can have free variables, and a morphism f from one E-PEG $\mathcal{A}$ to another E-PEG $\mathcal{B}$ is a map from the free variables of $\mathcal{A}$ to nodes of $\mathcal{B}$ such that, when the substitution f is applied to $\mathcal{A}$, the resulting E-PEG is a subgraph of $\mathcal{B}$. The substitution is also required to map expressions that are known to be equal in $\mathcal{A}$ to expressions that are equal in $\mathcal{B}$.

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

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

We now revisit the example from Figure 3, and this time explain it using our category theoretic algorithm. Figure 5 shows the generalization process for this example using E-PEGs. The objects (boxes) in this figure are E-PEGs, and the thin arrows between them are morphisms. The example has two axiom applications, and so Figure 5 consists essentially of two side-by-side instantiations of diagram (7). The thick arrows identify the steps taken by inference and generalization. The equality edges in each E-PEG are labeled with either a circle or a triangle to show how these equality edges map from one E-PEG to another through the morphisms.

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

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

Having singled out which equality we want to focus on, we generalize the second axiom application using our three step approach from Section 3: a pullback, a pushout, and then a pushout completion. The pullback identifies how the axiom is contributing to the equalities we are interested in – in this case it contributes through the triangle-labeled equality. The pushout then unifies the equality we are interested in generalizing with a freshly instantiated version of the axiom's conclusion (with a and b being fresh). Finally, the pushout completion essentially runs the axiom in reverse, removing the axiom's conclusion. In particular, it takes the substitution from the unification and applies it to the premise of the axiom to produce the E-PEG $[a + b \cdots 0]$.

This E-PEG, which is the result of generalizing the second axiom application, is then used as a starting point for generalizing the first axiom application, again using our three steps. The pullback identifies that the first axiom establishes the circle-labeled equality edge. The pushout unifies $[a + b \cdots 0]$ with a freshly instantiated version of the axiom's conclusion (with c being fresh). Note that

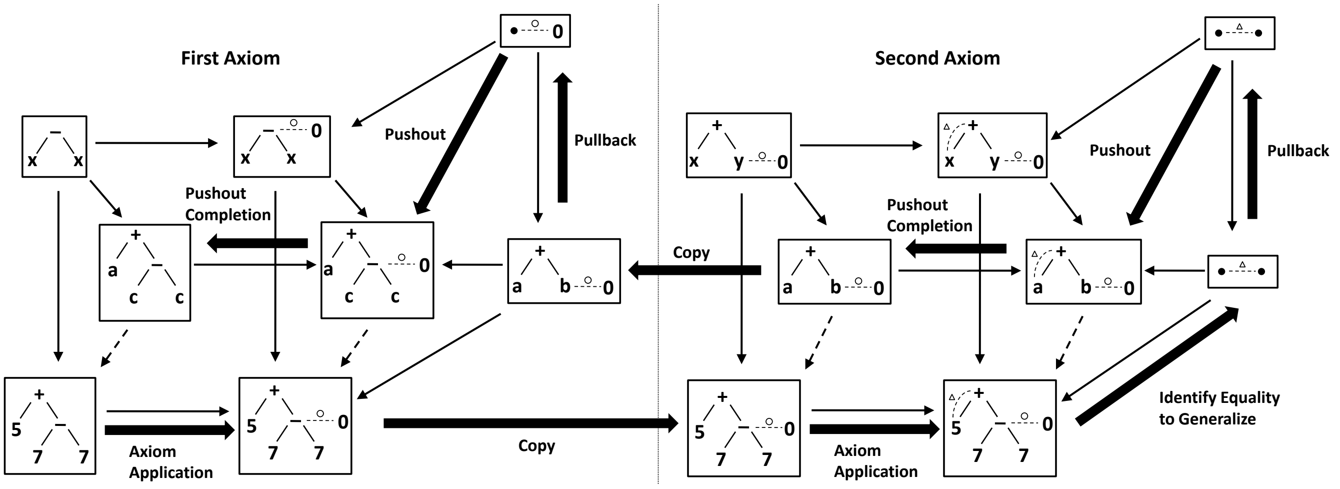


Figure 5. Example of generalization using E-PEGs

the circle-labeled equality edge in $[a + b \cdots 0]$ must unify with the corresponding equality edge in the axiom's conclusion, and so b gets unified with the minus node. Finally, the pushout completion runs the first axiom in reverse, essentially removing the axiom's conclusion. The result is our generalized starting E-PEG for that proof. We then generate a rule stating that whenever this starting E-PEG is found, the final conclusion of the proof is added, in this case the triangle-labeled equality.

The details of how our E-PEG category is designed affects the optimizations that our approach can learn. For example, the category described above has free variables, but they only range over E-PEG nodes. For additional flexiblity, we can also introduce free variables that range over node operators, such as variables OP_1 and OP_2 in Figure 1. This would allow us to generate optimizations that are valid for any operator, for example pulling an operation out of a loop if its arguments are invariant in the loop. For even more flexibility, we can augment our E-PEG catgory with domain-specific relationships on operator variables, which could be used to indicate that one operator distributes over another. With this additional flexiblity, we can learn the more general version of LIVSR show in Figure 1. In all these cases, to learn the more general optimizations, one has to not only add flexiblity to the category, but also re-express the axioms so that they take advantage of the more general category (as was shown in Section 2.1). The E-PEG category can also be augmented with new structure in order to accommodate analyses not based on equalities. For example, an alias analysis could add a distinctness relation to identify when two references point to different locations. This would allow our generalization technique to apply beyond the kinds of equality-based optimizations that our Peggy compiler currently performs [22].

5. Other applications of generalization

The main advantage of having an abstract framework for proof generalization is that it separates the domain-independent components of proof generalization — how to combine pullbacks, pushouts, and pushout completions — from the domain-specific components of the algorithm — how to compute pullbacks, pushouts, and pushout completions. As a result, not only does this abstraction provides us with a significant degree of flexibility within our own domain of E-PEGs, as described in Section 4, but it also enables applications of proof generalization to problems unrelated to E-PEGs. We illustrate this point by showing how our generalization framework from Section 3 can be used to learn efficient query optimizations in relational databases (Section 5.1) and also assist programmers with debugging static type errors (Section 5.2). Additional applications of our proof generalization framework, such as type generalization and contract debugging, can be found in the technical report [21].

5.1 Database query optimization

In relational databases, a small optimization in a query can produce massive savings. However, these optimizations become more expensive to find as the query size grows and as the database schema grows. We focus here on the setting of conjunctive queries, which are existentially quantified conjunctions of the predicates defined in the database schema. For example, the query $\exists y.\ R(x, y) \wedge R(y, z)$ returns all elements x and z for which there exists a y such that (x, y) and (y, z) are in the R table (relation). For sake of brevity, we discuss only conjunctive queries without existential quantification.

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

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

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

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

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

Adding tuples to a query in general is harmful because each added tuple represents an additional join. However, combined with equality-generating dependencies, these additional tuples can be used to infer useful equalities, which can then simplify the query. Let us apply an algorithm known as "the chase" [6] to optimize our example query using $A \rightarrow C$ and $B \twoheadrightarrow A$:

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

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

Our framework from Section 3, instantiated to the database setting, can use instances of optimized queries to identify general rules for when adding tuples to a query is helpful. In particular, in the above example, it could identify exactly what properties of the original query led to the inferred equality. The category we will use in this example is like **Rel** but with quaternary relations. The "axiom" $A \rightarrow C$ can be expressed categorically by the morphism

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

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

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

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

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

or simply $B \rightarrow C$. Thus, our framework can be used to learn equality-generating dependencies, removing the need for the intermediate generated tuples. This was possible because the dependencies involved, namely $A \rightarrow C$ and $B \twoheadrightarrow A$, could be expressed categorically as morphisms. We have proven that our learning technique can be used so long as all the dependencies can be expressed in this manner. Although the primary purpose of applying our framework to database optimizations was to demonstrate the flexibility of our framework, discussions with an expert in the database community [5] have revealed that our technique is in fact a promising approach that would merit further investigation.

5.2 Type debugging

As type systems grow more complex, it also becomes more difficult to understand why a program does not type check. Type systems relying on Hindly-Milner type inference [18] are well known for producing obscure error messages since a type error can be caused by an expression far removed from where the error was finally noticed by the compiler. Below we show how to apply our framework as a type-debugging assistant that is similar to [12], but is also easily adaptable to additional language features such as type classes [21].

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

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

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

Since `readSTRef` is a stateful operation, the lifting functions `liftM2` and `liftM` allow `max` and `Just` to handle this state. Unfortunately, this program does not type check. The Glasgow Haskell Compiler, when run on the above program using `do` notation for the recursive call, produces the error "`readSTRef ref` has inferred type `ST` $s\ a$ but is expected to have type `Maybe` a". This error message does not point directly to the problem, so the programmer has to examine the program, possibly even callers of `maxInRefList`, to understand why the compiler expects `readSTRef ref` to have a different type. Within `maxInRefList` alone there are many possiblities, such as the lifting operations, dealing with `Maybe` correctly, and the recursive call. Here we can apply proof generalization to limit the scope of where the programmer has to search, thereby helping identify the cause of the type error.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6. Manipulating proofs

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

6.1 Sequencing axiom applications

Our generalization technique requires proofs to be represented as a sequence of linear steps. However, proofs are often expressed as trees, in which case one needs to linearize the tree before our technique is applicable. The most faithful encoding of a proof tree is to use "parallel" axiom applications (which are formalized in the technical report [21]) to directly encode the tree: each step in the linearized proof corresponds to the parallel application of all axioms in one layer of the proof tree. This encoding is the most faithful linearization of a proof tree because the tree can be reconstructed from the linearization.

A simpler linearization is to flatten the tree so that each axiom application in the linearized proof corresponds to an axiom application in the proof tree. In this setting, axiom applications that are unordered in the tree must somehow be ordered. Unfortunately, when two axiom applications have overlapping conclusions, different orders in the linearized proof can lead to different and incomparable generalizations. Nonetheless, we have shown that no matter what order is selected, the generalized result will be equal to or possibly better than the result of using the "parallel" encoding which keeps the tree structure intact. As a result, since sequencing can only help, our implementation sequences axiom applications before generalizing, rather than retaining the parallel encoding.

6.2 Removing irrelevant axiom applications

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

6.3 Decomposition

As mentioned in Section 2.4, we decompose generated optimizations into smaller optimizations that are more broadly applicable. We can view decomposition as taking the original proof and cutting it up into smaller lemmas before applying generalization. In the context of E-PEGs, performing decomposition requires us to determine the set of inferred equalities along which we want to cut the proof (the first step mentioned in Section 2.4). Formally, we represent the set of cut-points as an object $\mathcal{S}$ and a morphism $\mathit{sub} : \mathcal{S} \to \mathcal{E}_n$, where $\mathcal{E}_n$ is the final inferred result of the proof. Then, in each step of generalization, we check whether the current property prop_m being generalized is contained within sub by determining whether there exists a morphism from $\mathcal{P}_m$ to $\mathcal{S}$ such that the following diagram commutes:

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

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

Optimization	Description
LIVSR	Loop-induction variable SR
Inter-loop-SR	SR across two loops
LIVSR-bounds	Optimizes loop bounds after LIVSR
ILSR-bounds	Optimizes loop bounds after Inter-loop-SR
Fun-specific-opts	Function-specific optimizations
Spec-inlining	Inline only for special parameter values
Partial-inlining	Inline only part of the callee
Tmp-obj-removal	Remove temporary objects
Loop-op-Factor	Factor op out of loop
Loop-op-Distr	Distribute op into loop to cancel other ops
Entire-loop-SR	Replace entire loop with one op
Array-copy-prop	Copy prop through array elements
Design-pats-opts	Remove overhead of design patterns

Figure 6. Learned optimizations (SR = Strength Reduction)

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

$\Longmapsto$

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

(a) Concrete example

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

$\Longmapsto$

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

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

(b) Generated optimization rule

Figure 7. Generalized inter-loop strength reduction

7. Experimental evaluation

We used the Peggy infrastructure [22] and E-PEGs to implement our technique for generating optimizations (as was illustrated in Sections 2 and 4). In this section, we experimentally validate three hypotheses about our technique: (1) our technique allows a programmer to easily extend the compiler by sketching what an optimization looks like with before-and-after examples (2) our technique can amortize the cost of expensive-to-run super-optimizers by generating fast-to-run optimizations and (3) our technique can even learn optimizations that are significantly profitable on code that the compiler was *not* trained on.

7.1 Extending the compiler through examples

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

The optimizations that our system learned from examples are listed in Figure 6. It took on average 3.5 seconds to learn each example (including translation validation). The only optimizations in this list that are performed by `gcc -O3` are LIVSR and Array-copy-prop. This demonstrates the benefit of our system: it allows an end-user programmer to easily implement non-standard optimizations targeting application domains with high-performance needs, such as computer graphics, image processing, and scientific computing.

The LIVSR optimization was already shown in Section 2. Optimizations LIVSR-bounds through Loop-op-Factor will be covered throughout the remainder of Section 7. We start with Inter-loop-SR and ILSR-bounds. These optimizations apply to a common programming idiom in image processing, which is shown in the left part of Figure 7(a). The `img` variable is a two-dimensional image represented as a one-dimensional array in row-major form. Programmers commonly use this kind of representation to efficiently store dynamically-sized two-dimensional images. The original code in Figure 7(a) uses a convenient way of iterating over such arrays, whereas the transformed code uses the more efficient, yet harder to program, formulation that programmers typically use (it removes the multiplication and addition from the inner loop). From the concrete instance, our generalizer determines that the `img` array is actually insignificant (the only part that matters is the `i*w + j` computation); it determines that the starting point of the outer loop is insignificant; and that the bounds on both loops can be generalized into loop invariant expressions. Furthermore, although we show the learned optimization as one rule, our decomposition algorithm would split this into two optimizations: one which optimizes the body of the loop (Inter-loop-SR), and one which optimizes the bounds of the loop (ILSR-bounds). A similar decomposition when learning LIVSR would also produce LIVSR-bounds. Also, all our generated optimization rules — including the one in Figure 7(b) — can apply to programs containing other statements in the loops that do not affect the significant program fragments being optimized.

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

$\Longmapsto$

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

(a) Concrete example

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

$\Longmapsto$

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

where E_3 is a loop-invariant expression

(b) Generated optimization rule

Figure 8. Loop-operation factoring

When traditional compilers like `gcc -O3` do not perform the optimization described above, an image-processing programmer would be forced to use the more efficient but error-prone version of the code. With our system, the programmer can use the simpler version and rely on the compiler to optimize it into the more efficient one. Furthermore, if the programmer encounters a new instance of the programming pattern that the generated rule does not cover, the programmer can train the compiler with another example. This scenario emphasizes how easy it is for non-compiler-experts to benefit from our system. Extending the compiler is as simple as providing a single concrete example, without having to worry about the side conditions required for correctness or having to learn a new language — like the language in Figure 7(b), including expression variables like E_1 and side conditions like "E_1 is loop-invariant".

We next show another example of an optimization not performed by `gcc -O3`, loop-operation factoring (Loop-op-Factor in Figure 6). The concrete example that we used to sketch the optimization is shown in Figure 8(a). The multiplication inside the loop gets factored out of the loop through the additions. The generalization that our system generates is shown in Figure 8(b). Our generalizer determined that the use of the array is insignificant and the format of the loop is insignificant, as is the constant 7, which can in fact be any loop-invariant expression.

Finally, we show how our system can learn several optimizations for the power function from Figure 9. Starting with the concrete optimization instance `pow(a, p) * pow(b, p)` ⇛ `pow(a * b, p)`, our generalizer shows that the two concrete programs are equivalent, and then generalizes the example into the optimiza-

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

Figure 9. The power function for integers

tion $\forall x, y, z \in \texttt{int}.\ \texttt{pow}(x, z)\ \texttt{*}\ \texttt{pow}(y, z) \Mapsto \texttt{pow}(x\ \texttt{*}\ y, z)$. This is an example of Fun-specific-opts from Figure 6. Similarly, we were able to get our generalizer to learn the non-trivial optimization: $\forall x \in \texttt{uint}.\ \texttt{pow}(2, x) \Mapsto 1 \ll x$ (which is an example of Spec-inlining in Figure 6). Here again, neither of these optimizations are performed by `gcc -O3`, whereas our approach allows the programmer to easily specify these optimizations by example.

7.2 Learning from super-optimizers

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

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

Several of the optimizations learned using before-and-after examples in Section 7.1 cannot be learned by using the super-optimizer – they really do require a programmer to give an input-output example. For instance, if the Peggy super-optimizer were given `pow(a,p)*pow(b,p)` to optimize using basic axioms, it would not be able to find the desired transformed program `pow(a*b,p)` because, even though it can decompose the original expression into smaller pieces, it cannot guess how to reassemble them into `pow(a*b,p)`. However, Peggy can prove the original and transformed programs equivalent because it sees the `pow(a*b,p)` term to which it can apply axioms. In essence it is easier to apply axioms on the original and transformed programs, and meet in the middle, rather than derive the transformed program from the original. With the proof, our approach can learn a new optimization that the Peggy super-optimizer would not have performed.

Our super-optimizer experiments also show how inlining combined with generalization produces useful and unconventional optimizations. Inlining in Peggy simply adds the information that a call node is equivalent to the body of the called function. Adding this equality does not force Peggy to choose the inlined version; it just provides more options in the caller's E-PEG for the profitability heuristic to choose from. When running on code that uses the `pow` function from Figure 9, the super-optimizer applied the inlining axiom on `pow`, thus pulling the body of `pow` into the E-PEG. Peggy then exploited the fact that `pow` does not affect the heap to optimize the *surrounding context*, but chose not to inline `pow` in the final result. From this our generalizer learned the optimization that `pow` is heap-invariant; in future compilations, Peggy can immediately use the fact that `pow` does not affect the heap to optimize the surrounding context without the expensive process of inlining `pow`. This is an example of what we call *partial inlining* (Partial-inlining in Figure 6), where the optimizer exploits the information learned from inlining without opting to inline the function. Another example of partial inlining we observed involved a large function that modifies the heap but always returns 0. By applying the inlining axiom, Peggy was able to optimize the surrounding context with the information that the return value was 0, but then chose not to inline the called function because it was too large. In this case the generalizer would learn that the function always returns 0, which can be used in future compilations without having to apply the inlining axiom.

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

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

7.3 Cross-training

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

We conducted a preliminary evaluation of this hypothesis on a Java ray tracer that uses a functional vector library. Peggy's optimization phase improves this benchmark's performance by 7%, compared to only using Sun's Java 6 JIT, by removing the short-lived temporary objects (Tmp-obj-removal in Figure 6).

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

8. Related work

There has been a long line of work on making optimizers extensible or easier to develop, including Sharlit [23], Whitfield and Soffa's Gospel system [24], the Broadway extensible compiler [11], and the Rhodium system for expressing optimizations [14, 15]. In all these systems, however, the programmer has to learn a new language or compiler interface to express optimizations. In contrast, our approach learns an optimization from a single example provided by the programmer in a language already familiar to them.

In the context of machine learning, our approach is an instance of Explanation-Based Learning (EBL) [9]. EBL refers to learning from a single example using an explanation of that example. EBL has been applied to a wide variety of domains, such as Prolog optimization [10], logic circuit designs [8], and software reuse [3]. Many of these applications use algorithms based on unification or Prolog [7, 8, 10]. The declarative components of Prolog can be encoded in our framework by combining categories of expressions with categories of relations. Furthermore, most EBL implementations provide no guarantees on what is learned, while we can prove that our technique learns the most general lesson for a given explanation. Within EBL, our work is closely related that of Dietzen and Pfenning [7]. They use λProlog to extend EBL to higher-order and modal logic, and apply this framework to various settings including program transformations. However, they do not investigate ways to automatically train the optimizer, relying instead on the user to prove the transformation correct using tacticals. As a consequence, they do not attempt to decompose an optimization into suboptimizations, since a user manually proving an optimization would already do this. Furthermore, we experimentally demonstrate that generalization can be useful for extending compilers and amortizing the cost of super-optimizers.

Aside from EBL, there have been other uses of machine learning in the context of compiler optimizations. Techniques like genetic algorithms, reinforcement learning, and supervised learning, have been used to generate effective heuristics for instruction scheduling [17, 20], register allocation [20], prefetching [20], loop-unroll factors [19], and for optimization ordering [4]. In all these cases, the parts being learned are not the transformation rules themselves, but profitability heuristics, which are functions that decide when or where to apply certain transformations. As a result, these techniques are complementary to our technique: we generate the optimization rules themselves, but not the profitability heuristics (we use a single global profitability heuristic for all optimizations). Also, while modern machine learning techniques use statistical methods over large data sets, our EBL-based approach can learn from a very small dataset, even from just one example.

The idea of discovering optimizations has also been been explored in the setting of super-optimizers [2, 13, 16]. Super-optimizers try to discover optimizations by a brute-force exploration of possible transformed programs for a given input program. Traditional super-optimizers find concrete optimization *instances*, whereas our approach *starts* with optimization instances, and tries to generalize the instances into reusable optimization rules. As such, our work is complementary to super-optimization techniques. However, Bansal and Aiken's recent super-optimizer [2] does achieve a simple form of generalization, namely abstraction of register names and constants. In contrast, we perform a more sophisticated kind of generalization based on the reasons why the original and transformed programs are equivalent.

Acknowledgments We thank Suresh Jagannathan, Todd Millstein, Zachary Tatlock and the anonymous reviewers for their invaluable feedback on earlier drafts of this paper, as well as Alin Deutsch, Daan Leijen, and Rustan Leino for their feedback on alternative applications of proof generalization.

References

[1] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. John Wiley & Sons, 1990.

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

[3] R. Bergmann. Explanation-based learning for the automated reuse of programs. In *CompEuro*, 1992.

[4] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, Aug. 2002.

[5] A. Deutsch. Author of [6]. Personal communication, July 2009.

[6] A. Deutsch, A. Nash, and J. Remmel. The chase revisited. In *PODS*, 2008.

[7] S. Dietzen and F. Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. *Machine Learning*, 9(1):23–55, June 1992.

[8] T. Ellman. Generalizing logic circuit designs by analyzing proofs of correctness. In *IJCAI*, volume 1, pages 643–646, 1985.

[9] T. Ellman. Explanation-based learning: a survey of programs and perspectives. *ACM Computing Surveys*, 21(2):163–221, June 1989.

[10] M. Gandhe and G. Venkatesh. Improving prolog performance by inductive proof generalizations. In *Knowledge Based Computer Systems*, 1990.

[11] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of IEEE*, 93(2), 2005.

[12] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

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

[14] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.

[15] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

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

[17] A. McGovern, J. E. B. Moss, and A. G. Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine Learning, Special Issue on Reinforcement Learning*, 49(2/3):141–160, May 2002.

[18] R. Millner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[19] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, Mar. 2005.

[20] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *PLDI*, June 2003.

[21] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. Technical report, University of California, San Diego, Nov. 2009.

[22] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL*, Jan. 2009.

[23] S. W. K. Tjiang and J. L. Hennessy. Sharlit – A tool for building optimizers. In *PLDI*, 1992.

[24] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, Nov. 1997.