Specifications Can Make Programs Run Faster

Mark T. Vandevoorde

Massachusetts Institute of Technology
Cambridge, MA 02139 USA
Email: mtv@lcs.mit.edu

Abstract. This paper describes a strategy for using the information contained in formal specifications to enhance a compiler's ability to perform optimizations. Because specifications are simpler than code and because they abstract away irrelevant implementation details, a compiler with access to specifications can determine that an optimization is safe more often than compilers that analyze only code. Furthermore, formal specifications can be used to allow programmers to define new optimizations.

Our strategy has been implemented in a prototype compiler that incorporates theorem proving technology. The compiler identifies opportunities to perform conventional and programmer-defined optimizations.

1 Introduction

Many approaches to programming emphasize the use of abstractions. The basic idea is to make it easier to understand programs by achieving a separation of concerns. The *client* of an abstraction looks at its specification and writes code that uses the abstraction. He need not concern himself with how the specified behavior is achieved. The *implementor*'s job is to provide an implementation that satisfies the specification. Often, the implementation is substantially more complex than the specification, e.g., for reasons of efficiency, so the specification allows the client to reason about the abstraction at a simpler level.

Although programmers benefit from abstraction and specification when reasoning about programs, existing compilers do not. Compilers should be able to make good use of the information in specifications when optimizing programs, since determining if an optimization is safe requires reasoning about programs.

In this paper, we investigate how to exploit formal specifications to enhance optimization. We use specifications in three ways:

1. to allow programmers to define new optimizations that make abstractions more efficient to use,
2. to relax the preconditions for performing conventional optimizations, and
3. to improve a compiler's ability to recognize that the precondition for performing an optimization is satisfied.

Ultimately, the aim of this research is to make programs built using abstraction and specification more efficient.

Our approach has been implemented in a prototype compiler that incorporates theorem-proving technology to optimize programs written in a strongly typed, imperative language. Early experience with the compiler is encouraging. Performance improvements were obtained in parts of mature programs, and the compiler is able to detect opportunities for optimization that cannot be detected without looking at specifications.

In Section 2 we discuss how specifications can enhance three kinds of optimizations and then briefly discuss related work. In Section 3 we describe the specifications used in Speckle, the source language for our compiler, and we give proof rules for Speckle programs. In Section 4 we describe how our prototype compiler uses the proof rules to identify safe optimizations. Finally, in Section 5 we discuss how our approach fits in the context of software development and report on a case-study of optimizing a program.

2 Optimizations

2.1 Specialized Procedures

Specialized procedures are one kind of programmer-defined optimization for making general procedures run faster in special cases. The basic idea is that a procedure presents one simple specification but has multiple implementations. One implementation is general enough to work in any context, while the others are more efficient but work in fewer contexts. The compiler substitutes one of the faster implementations for the general one when it can prove (using specifications) that a calling context satisfies the stronger pre-condition of the faster implementation.

For example, consider the specification in Fig. 1 for `Table_Enter`, which enters the value for a key into a table. Suppose that the specifications in Fig. 1 are implemented using an unsorted list of key-value pairs with the invariant that no key appears more than once. Thus, `Table_Enter` must in general check whether `k` appears in the list. However, we would like to specialize `Table_Enter` to avoid this check in contexts where `k` cannot appear in the list, e.g., in Fig. 2.

Our approach is to have the programmer write the dual implementation for `Table_Enter` shown in Fig. 3. The "*specialize when*" construct directs the compiler to use the second implementation of `Table_Enter` when it can discharge the pre-condition `not(defined(t^,k))`. Specifications are required both to express the pre-condition and to discharge it in contexts like that in Fig. 1.

2.2 Common Subexpression Elimination

Common subexpression elimination (CSE) is a standard optimization to avoid recomputing an expression when its value is already available. For example, in the code fragment

```
x := a[i]; ...; y := a[i]
```

```
TableEnter = proc (t: Table, k: Key, v: Value)
   modifies t
   ensures t' = bind(t^, k, v)

TableLookup = proc (t: Table, k: Key) returns (v: Value)
   modifies --
   ensures v = image(t^, k)
   except signals missing when not(defined(t^, k))
```

Fig. 1: Table Specification

```
v: Value := TableLookup(t, k)
  except when missing:
            v := ValueCreate()
            TableEnter(t, k, v)
         end
```

Fig. 2: A Context to Specialize `TableEnter`

there is no need to recompute `a[i]` because its value is available, unless the code between the two occurrences of `a[i]` changes `a` or `i`.

Although compilers are very good at eliminating expressions that use only primitive operations like `__+__` and `__[__]`, they are less effective at eliminating procedures calls. The problem is that in imperative programs, a procedure may perform a visible side effect, so eliminating a procedure call, even when its result is available, may alter the program's behavior.

Specifications make it easier to eliminate procedure calls by explicitly stating whether a procedure performs any visible side effects. For example, in Fig. 1 the specification of `TableLookup` states that `TableLookup` performs no visible side effect. Thus, it is safe to eliminate calls to `TableLookup`, but it isn't safe to eliminate calls to `TableEnter`, which may modify its first argument.

Specifications also make it easier to recognize equal expressions that are not syntactically identical. For example, in the code fragment

```
TableEnter(t, k, v1)
v2 := TableLookup(t, k)
```

the call to `TableLookup` can be replaced by `v1`. Although it is impractical to perform this optimization by analyzing the implementations of `TableLookup` and `TableEnter`, the optimization is easy to do given the specifications in Fig. 1 and the axiom about tables

```
∀ t: Table, k: Key, v: Value [ image(bind(t, k, v), k) == v ]
```

Currently, compilers only eliminate an expression when an *equal* value is available. With specifications, it is possible to relax this condition. Consider,

```
TableEnter = proc (t:Table,k:Key,v:Value)
        p:pair := FindPair(t,k)
          except when none:InsertPair(t,k,v)
                            return
                      end
        p.val:= v
    specialize when not(defined(t^,k)):
        InsertPair(t,k,v)
    end TableEnter
```

Fig. 3: A Dual Implementation for TableEnter

for example, the code

```
i1 := IntSetLeast(s)
i2 := IntSetAnyElement(s)
```

where **Least** is specified to return the smallest element of the set s while the specification of **AnyElement** allows it to return any element. With the specifications, it is possible to recognize that the call to **AnyElement** can be replaced by i1, even when **AnyElement** would have returned a different element. Without the specifications, this optimization appears unsafe because it might alter the value computed for i2.

2.3 Identifying Loop-Constant Expressions

When a compiler can determine that an expression is a loop constant, it can optimize the loop to compute the expression once rather than once per iteration.

Most of the enhancements to common subexpression elimination also apply to identifying loop-constant expressions. Expressions can be generalized to include procedure calls that perform no visible side effects. Furthermore, the called procedure may be non-deterministic: it need not compute the same value each iteration, but the value returned for the first iteration must be substitutable for the values returned on the other iterations. For example, it is safe to treat AnyElement(s) as a loop "constant" when the body of loop does not modify s.

2.4 Side Effect Analysis

All of the optimizations discussed above require the compiler to reason about side effects. For common subexpression elimination, it must determine that the code executed between the expressions does not change their value. To identify a loop constant expression, the compiler must determine if the body of the loop affects the value of the expression. To call a specialized implementation, the compiler must often determine that a pre-condition established at one point in the program is still true at a later point, e.g., that in Fig. 2, the pre-condition not(defined(t^,k)) established when TableLookup signals missing is not invalidated by a side effect of ValueCreate.

Specifications improve the analysis of side effects. Only specifications can distinguish between side effects that are visible to the client's code from those that are invisible. E.g., a procedure might cache previously computed results in a private data structure—changes to this data structure are invisible to clients.

Specifications of data abstractions also make some side effects invisible because they abstract away parts of data structures used to implement values of a data type. E.g., if tables are represented as lists of key-value pairs, an operation of the table type might sort the list. This side effect is invisible to clients.

Data abstractions are also useful because they introduce new types that, in a strongly typed language, can be used to bound side effects. For example, suppose a user defines the types `IntSet`, `IntQueue`, `IntStack` and implements each one using integer arrays. Because a user-defined type must guarantee that the representation of a value is never directly accessible outside of the implementation of the type, the compiler can assume that a procedure that can only access an `IntSet` cannot modify an `IntQueue` even though both are represented using the same type.

Finally, a data abstraction may specify that the data type is *immutable*, i.e., that no visible side effect is possible on instances of the data type.[1] This eliminates the need to analyze side effects for instances of the type.

2.5 Related Work

The idea of allowing programmers to define new optimizations has been suggested before. In [15], Scherlis allows programmers to enhance performance by writing "expression procedures" for a functional language of recursive equations. In [8], Hisgen presents an unimplemented design of a language where the author of a module provides transformation rules used to restructure the program for efficiency. However, his approach is not modular since, in general, the user must consider how procedures from different modules interact.

Our extensions to standard optimizations are not intended to replace traditional compilation techniques that deal with register allocation and other machine-level issues. The extensions are complementary to code analysis [3], which can perform standard optimizations that span procedure boundaries, e.g., identifying redundant code in different procedures. However, even interprocedural code analysis techniques like [2, 10] can be foiled by invisible side effects that are concealed by specifications in our approach. Furthermore, many code analysis techniques simplify the problem of estimating side effects by either not supporting procedures, not supporting pointers, or restricting pointers to one level of indirection [1, 9, 13, 14, 16]. Because specifications contain information that is useful for bounding side effects, we do not need to make such restrictions. Our handling of side effects is more like the FX language, which augments an imperative dialect of LISP with specifications describing side effects [12]. However, FX specifications cannot express user-defined optimizations.

[1] For example, integers and bignums are immutable in Common LISP, but cons cells are mutable.

3 Speckle

Speckle is a strongly typed, imperative programming language based on CLU [11] and designed to experiment with specification-based optimization. We chose CLU as a starting point because it has many of the features of modern programming languages, including data abstraction, exceptions, and (implicit) pointers. Speckle programs are specified using Larch [7], which uses first order predicate logic. To simplify reasoning about programs, some features of CLU are omitted: polymorphism and non-local variables.[2] We also omit first class procedures (procedures as data) because they are difficult to specify in first-order logic.

First, we describe Speckle specifications. Next, we give inference rules, based on specifications, that will be used to prove that optimizations are safe.

3.1 Larch/Speckle Specifications

Larch is a two-tiered specification language. The Larch Shared Language (LSL) is used to define useful mathematical functions in a fragment of multisorted, first-order logic. Functions and sorts defined in LSL are independent of any programming language.

The semantics of LSL defines a first-order *theory*—an infinite set of formulae—for LSL specifications. The theory is the consequence closure of the specification's axioms and inference rules, which include the normal inference rules of predicate logic. For our purposes, it suffices that LSL specifications provide useful axioms in the form of equations.

The Larch/Speckle interface language is the glue between a Speckle program and LSL. Larch/Speckle formalizes the notion of a program state and provides a language for specifying data type and procedure interfaces. These interfaces refer to LSL sorts and functions, e.g., Fig. 1 refers to the LSL functions `image` and `bind`.

A Speckle program state consists of an environment and a store:

$$\begin{aligned} \mathit{Prog\ State} &= \mathit{Env}\ X\ \mathit{Store} \\ \mathit{Env} &= \mathit{Ident} \rightarrow (\ \mathit{ImmValue} + \mathit{Loc}\ ) \\ \mathit{Store} &= \mathit{Loc} \rightarrow \mathit{MutValue} \end{aligned}$$

σ^{Env} denotes the environment of program state σ, and σ^{Str} denotes its store.

Values are divided into three domains. ImmValues are used to represent values of immutable types, and MutValues are used for values of mutable types. A Loc represents the location (address) of a mutable data structure. A procedure *modifies* a Loc l if $\sigma^{\mathrm{Str}}_{\mathrm{pre}}(l) \neq \sigma^{\mathrm{Str}}_{\mathrm{post}}(l)$.

[2] CLU does not have fully global variables, but it has "own" variables. A module's "own" variables are accessible by any procedure in the module but are inaccessible outside the module. In the absence of procedure variables, techniques based on interprocedural code analysis could effectively bound the set of global variables read or written by a Speckle procedure, e.g., [1]. These variables would be treated like additional arguments to the procedure. Alternatively, Speckle could require the user to list any global variable that a procedure could read or write, e.g., in a fashion similar to Euclid and Modula.

$\forall\ \sigma_{\text{pre}}, \sigma_{\text{post}}$: *Prog State, t: TableLoc, k: Key, v: Value*

`Table_Enter`:

$$Norm(\sigma^{\text{Str}}_{\text{pre}}, \sigma^{\text{Str}}_{\text{post}}, t, k, v) == \sigma^{\text{Str}}_{\text{post}}(t) = bind(\sigma^{\text{Str}}_{\text{pre}}(t), k, v) \land \forall\ l{:}\ TableLoc\ [l \neq t \Longrightarrow \sigma^{\text{Str}}_{\text{post}}(l) = \sigma^{\text{Str}}_{\text{pre}}(l)]$$

`Table_Lookup`:

$$Norm(\sigma^{\text{Str}}_{\text{pre}}, \sigma^{\text{Str}}_{\text{post}}, t, k, v) == v = image(\sigma^{\text{Str}}_{\text{pre}}(t), k, v) \land \sigma^{\text{Str}}_{\text{post}} = \sigma^{\text{Str}}_{\text{pre}}$$

$$Excpt(\sigma^{\text{Str}}_{\text{pre}}, \sigma^{\text{Str}}_{\text{post}}, t, k) == \sigma^{\text{Str}}_{\text{post}} = \sigma^{\text{Str}}_{\text{pre}}$$

$$Guard(\sigma^{\text{Str}}_{\text{pre}}, t, k) == not(defined(\sigma^{\text{Str}}_{\text{pre}}(t), k))$$

Fig. 4: Procedure Predicates

There may be several aliases for a given Loc l. For example, the environment may map any number of Idents to l. Also, any MutValue in the range of the store or any ImmValue in the range of the environment may contain l. For example, a value for an array of mutable data contains the Loc of each (mutable) element. Similarly, a record value contains the Loc of each mutable field.

Aliasing is not possible for Idents because ImmValues and MutValues may not contain Idents.

Data Type Interfaces. A data type interface names an LSL sort used to represent values of the type and indicates whether the type is mutable—this determines whether the sort is an ImmValue or a MutValue. When a type is specified as mutable, a *location sort* is implicitly defined for the type, e.g., *TableLoc* for locations of `Tables`. In procedure specifications, a term l denoting a location may be suffixed by ^ to denote $\sigma^{\text{Str}}_{\text{pre}}(l)$ or by ' to denote $\sigma^{\text{Str}}_{\text{post}}(l)$. A formal parameter whose type is mutable denotes a location; a formal parameter whose type is immutable denotes an ImmValue.

Procedure Interfaces. Specifications of procedures, which may signal exceptions, are written as pre- and post-conditions in a stylized fashion. The `requires` clause defines the pre-condition, *Req* (if any). The **`modifies`** clause restricts the side-effects of the procedure by defining the set of locations that the procedure is allowed to modify; this restriction is part of every post-condition. The post-condition for a normal return, *Norm*, is further specified by the **`ensures`** clause, which typically defines the results in terms of the arguments. All procedures are assumed to terminate.

For the sake of brevity, here we allow at most one exception. The **`when`** clause defines a second condition on the pre-state, *Guard*. The procedure must signal the exception exactly when *Guard* is true. The post-condition for an exceptional return, *Excpt*, is defined by the **`modifies`** clause and an optional **`ensuring`** clause.

Fig. 4 lists the result of translating the interface specifications of Fig. 1 into predicates when `Table` is the only mutable type used by the program. For every other mutable type T, the assertion "$\forall$ *l*: *TLoc* [$\sigma^{\text{Str}}_{\text{post}}(l) = \sigma^{\text{Str}}_{\text{pre}}(l)$]" would be added as a conjunct to the post-conditions of `Table_Enter`. Note that the post-conditions constrain only the store; the environment is defined by the semantics of the programming language. Pre- and post-conditions may not refer to environments because Speckle does not allow global variables.

3.2 Proof Rules for Speckle Programs

Hoare rules are a standard way of defining proof rules for programs in the form of a parse tree. However, in language with exceptions, it is awkward to use parse trees for the same reason that it is hard to give Hoare rules for statements like `break` and `continue`. Therefore, we use Floyd's approach [4] and define proof rules for Speckle programs that are in the form of a control flow graph (CFG). The proof rules have not been checked against a language semantics.

Each program has a unique entering edge labeled *enter*, one or more exiting edges, and zero or more internal edges. A program is the body of a single procedure, and each exiting edge corresponds to either a normal return or signaling an exception. There are four kinds of nodes: assignment, procedure call, merge, and loop.

We assume that the pre-condition of the program ensures that the program terminates, only calls procedures whose pre-conditions are satisfied, and uses no uninitialized variables. We also assume that all specifications are correct, i.e., they accurately describe their implementations.

Associated with each CFG edge e is an LSL theory, $\mathcal{T}_{\text{e}}$. To prove that some predicate P holds at the program point denoted by an edge e, one must prove that the formula $P(\sigma_{\text{e}}) = true$ is in $\mathcal{T}_{\text{e}}$. The relation $\in$ is used to define the theories at each edge: $F \in \mathcal{T}_{\text{e}}$ means that formula F is in theory $\mathcal{T}_{\text{e}}$. Proof rules define $\in$ inductively, i.e., the theory of each internal and exiting edge is determined by the structure of the CFG and the theory of the entering edge.

The theory $\mathcal{T}_{\text{enter}}$ of the entering edge comes from the specifications of of procedures and data types used in the CFG. $\mathcal{T}_{\text{enter}}$ is the consequence closure of the union of: the theories of all LSL specifications used; the theory of the program state, and procedure predicates defined by Larch/Speckle; and the precondition for entering the CFG specified by the user, if any.

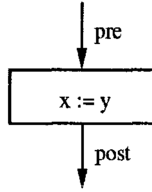
Fig. 5 and Fig. 6 list the proof rules for Speckle. We extend the notation for the hypothesis of a proof rule to include a template of a subgraph appearing in the program. The first proof rule is that each theory is closed under the usual inference rules of predicate logic.

The second proof rule is that the theory of an edge j is an extension of the theories of each edge i that dominates j.[3] This rule propagates the formulae defining σ_{i} in $\mathcal{T}_{\text{i}}$ to $\mathcal{T}_{\text{j}}$, so it allows $\mathcal{T}_{\text{j}}$ to define σ_{j} in terms of σ_{i}. For example, if

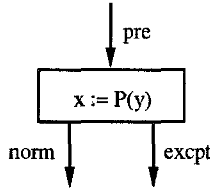
[3]Edge i *dominates* edge j if every path from the entering edge to j must pass through i. Every edge dominates itself. Edge i *strictly dominates* edge j if i dominates j and $i \neq j$.

$$\frac{F \in ConsequenceClosure(\mathcal{T}_e)}{F \in \mathcal{T}_e}$$

$$\frac{\begin{array}{c} F \in \mathcal{T}_i \\ \text{Edge } i \text{ dominates edge } j \end{array}}{F \in \mathcal{T}_j}$$



$\sigma^{\mathtt{Env}}_{\text{post}}(\text{'}\mathbf{x}\text{'}) = \sigma^{\mathtt{Env}}_{\text{pre}}(\text{'}\mathbf{y}\text{'})$	$\in \mathcal{T}_{\text{post}}$
$\forall var : Ident[var \neq \text{'}\mathbf{x}\text{'} \Longrightarrow \sigma^{\mathtt{Env}}_{\text{post}}(var) = \sigma^{\mathtt{Env}}_{\text{pre}}(var)]$	$\in \mathcal{T}_{\text{post}}$
$\sigma^{\mathtt{Str}}_{\text{post}} = \sigma^{\mathtt{Str}}_{\text{pre}}$	$\in \mathcal{T}_{\text{post}}$



$Norm(\sigma^{\mathtt{Str}}_{\text{pre}}, \sigma^{\mathtt{Str}}_{\text{norm}}, \sigma^{\mathtt{Env}}_{\text{pre}}(\text{'}\mathbf{y}\text{'}), \sigma^{\mathtt{Env}}_{\text{norm}}(\text{'}\mathbf{x}\text{'}))$	$\in \mathcal{T}_{\text{norm}}$
$\neg\, Guard(\sigma^{\mathtt{Str}}_{\text{pre}}, \sigma^{\mathtt{Env}}_{\text{pre}}(\text{'}\mathbf{y}\text{'}))$	$\in \mathcal{T}_{\text{norm}}$
$\forall var : Ident[var \neq \text{'}\mathbf{x}\text{'} \Longrightarrow \sigma^{\mathtt{Env}}_{\text{norm}}(var) = \sigma^{\mathtt{Env}}_{\text{pre}}(var)]$	$\in \mathcal{T}_{\text{norm}}$
$Excpt(\sigma^{\mathtt{Str}}_{\text{pre}}, \sigma^{\mathtt{Str}}_{\text{excpt}}, \sigma^{\mathtt{Env}}_{\text{pre}}(\text{'}\mathbf{y}\text{'}))$	$\in \mathcal{T}_{\text{excpt}}$
$Guard(\sigma^{\mathtt{Str}}_{\text{pre}}, \sigma^{\mathtt{Env}}_{\text{pre}}(\text{'}\mathbf{y}\text{'}))$	$\in \mathcal{T}_{\text{excpt}}$
$\sigma^{\mathtt{Env}}_{\text{excpt}} = \sigma^{\mathtt{Env}}_{\text{pre}}$	$\in \mathcal{T}_{\text{excpt}}$

Fig. 5: Proof Rules for Speckle

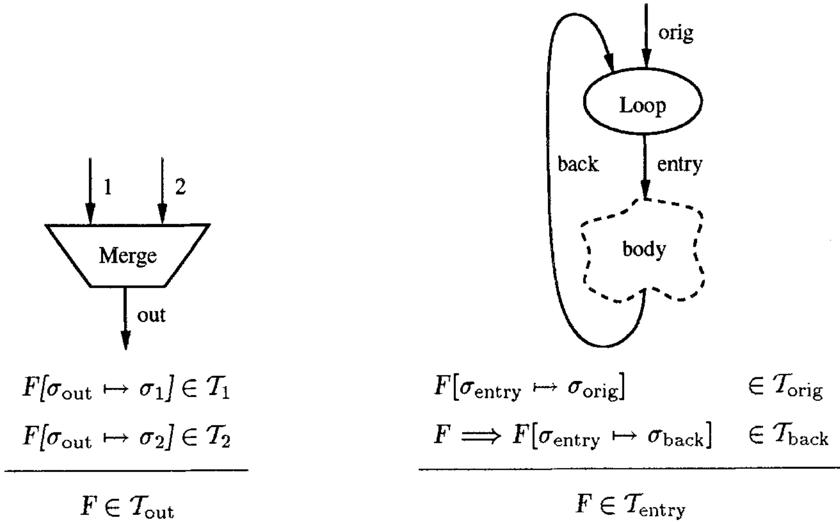


Fig. 6: Proof Rules for Speckle, continued

i and j are the edges before and after the statement `x := x+1`, σ_j^{Env}('**x**') can be defined in terms of σ_i^{Env}('**x**').

The rule for an assignment states that the only variable affected by the assignment is the one named on the left of "`:=`", so assignment to '**x**' never affects '**z**'. Furthermore, the store is unchanged.

The rule for a procedure call node[4] uses the predicates from the procedure's specification to define the post-state in terms of the pre-state. When the procedure returns normally, the only variable affected by a call is the one that is assigned the result of the call. If the procedure signals an exception, the environment is unchanged. Note that the theories of the exiting edges contain control-dependent information derived from the exception guard.

Because procedures can signal exceptions, a branch can be treated as a call to a procedure that takes a boolean argument and signals an exception exactly when the argument is true.

The merge and loop rule rules correspond to the familiar rules for proof-by-cases and proof-by-induction. The notation $F[\sigma_i \mapsto \sigma]$ denotes F with σ substituted for σ_i and with bound variables renamed to avoid capture.

Note that the theory of edge e is inconsistent ($true = false \in \mathcal{T}_e$) precisely when the edge is unreachable at run-time. E.g., if a procedure is called in a context where it cannot signal an exception ($\neg Guard \in \mathcal{T}_{pre}$), $\mathcal{T}_{excpt}$ is inconsistent because it contains both *Guard* and $\neg$*Guard*. The second proof rule propagates the inconsistency to each edge dominated by *excpt*, and this is correct: *excpt* is unreachable, so any edge dominated by *excpt* is also unreachable.

[4]The rule shown is for a procedure with a single argument and result. We rely on the reader's intuition to extend the rule for different numbers of arguments and results.

4 Prototype Compiler

We constructed a prototype compiler for Speckle that identifies optimizations using the LP theorem-prover [5] and the proof rules from the previous section. The compiler does not generate any code. LP is particularly well-suited for our purpose because it was designed to work with LSL and because it fails quickly when trying to prove a difficult conjecture rather than attempting expensive proof strategies. This is particularly important because many conjectures a compiler wants to prove are false, e.g., most procedure calls cannot be eliminated.

In Section 2, we gave informal pre-conditions for performing various kinds of optimizations. Each of these can be stated formally using the framework of the previous section. For example, to replace a procedure call `x := P(y)` by an assignment `x := z`, the compiler must prove that the available value `z` satisfies the post-condition of `P`, i.e.,

$$Norm(\sigma^{\mathtt{Str}}_{\mathrm{pre}}, \sigma^{\mathtt{Str}}_{\mathrm{pre}}, \sigma^{\mathtt{Env}}_{\mathrm{pre}}(\text{`}\mathbf{y}\text{'}), \sigma^{\mathtt{Env}}_{\mathrm{e}}(\text{`}\mathbf{z}\text{'})) \in \mathcal{T}_{\mathrm{pre}}$$

where *Norm* is the post-condition of `P`, pre is the edge entering the call node, and post is the exiting edge for the normal return, and e is an edge that dominates pre.[5] The compiler must also prove that the call does not signal an exception, i.e.,

$$\neg Guard(\sigma_{\mathrm{pre}}, \sigma^{\mathtt{Env}}_{\mathrm{pre}}(\text{`}\mathbf{y}\text{'})) \in \mathcal{T}_{\mathrm{pre}}$$

The Speckle compiler uses LP to discharge the proof obligations for optimizations. LP is primarily based on conditional term rewriting, which it uses to simplify terms to normal forms. To prove a conjecture by rewriting, one must simplify it to the term *true*.

LP can automatically convert a set of assertions like those found in LSL and Larch/Speckle specifications into a conditional term rewriting system. In general, a rewriting system only approximates the consequence closure of a set of assertions because a rewriting system is usually incomplete: it may not simplify some formulae in $\mathcal{T}_{\mathrm{e}}$ to *true*.

The compiler uses LP to approximate each theory $\mathcal{T}_{\mathrm{e}}$ by a conditional term rewriting system $\mathcal{R}_{\mathrm{e}}$. The strategy is to simplify a term containing a program state symbol σ_{e} into either a term that contains no program state symbols or one that contains only program state symbols of edges that dominate e. The net effect is to try to simplify a term into one expressed using at most σ_{enter}, the program state symbol for the entering edge.

The first step in building the term rewriting systems is to construct $\mathcal{R}_{\mathrm{enter}}$, the rewriting system for the entering edge. $\mathcal{R}_{\mathrm{enter}}$ is derived mechanically from the LSL specifications referenced by the program using the LSL Checker, which automatically translates LSL specifications into LP input, which LP then converts into a rewriting system. The compiler also adds assertions to axiomatize program states, location sorts, and predicates derived from Larch/Speckle specifications of procedure interfaces.

[5] If z is assigned between edge e and pre, the compiler must introduce a temporary variable to store the available value.

The next step is to construct rewriting systems for the other edges in the program in depth-first order. When edge e exiting node n is visited, $\mathcal{R}_e$ is constructed by extending the rewriting system of the edge entering n. The extensions are determined by n: if n is an assignment node, the conclusions from the assignment rule in Fig. 3 are added to $\mathcal{R}_e$; if n is a procedure call node, the procedure call rule is used. Nothing is added for merge or loop nodes.

Merge and loop nodes must be handled differently from the others because the hypotheses of these rules include subgoals that must be discharged using the theories of other edges. The basic strategy for proving $F \in \mathcal{T}_e$ is to first simplify F using $\mathcal{R}_e$; let t be the simplified form of F. If t contains a program state symbol for an edge that exits a merge or loop node, the compiler automatically attempts proof-by-cases or proof-by-induction.[6]

4.1 An Example

Fig. 7 is an example that illustrates our strategy. Procedure `RemoveDuplicates` uses two user-defined data types: `IntSet`, a type for mutable integer sets, and `IntArray`, a type for integer arrays that can grow and shrink dynamically. The syntactic expressions `a.low`, `a[i]`, and `a[j] :=...` are shorthands for calls to the procedures `IntArray_GetLow`, `IntArray_Fetch`, and `IntArray_Store`. `IntArray_Trim` takes an array, a starting index, and an element count and discards all other elements.

Using formal specifications and specializations of `IntSet` and `IntArray`, the compiler identifies the following optimizations automatically:[7]

1. The expressions `a[i]` on lines 5 and 6 can be replaced by the value computed for `a[i]` on line 4.

   This optimization relies on the `modifies` clauses of `Member` and `Insert` to show that `a` is unchanged since the call to `Fetch` on line 4.

2. The call to `Insert` need not check whether `a[i]` is in `s`. This is a specialization of `Insert`.

   This optimization relies on the semantics of `if`, the `modifies` clause of `Fetch` to show that `s` is unchanged, and the specification of `Member`.

3. The bounds checks for `a[i]` on line 4 and for `a[j]` on line 6 are unnecessary. These are specializations defined by `Fetch` and `Store`.

4. The two expressions `a.low` on line 10 can be replaced by the value computed for `a.low` on line 1.

Optimizations 3 and 4 require proof-by-cases, proof-by-induction, and array axioms to determine that the bounds of the array are invariant over the loop.

[6] If t contains more than one such symbol, the one for the edge closest to e is handled first.

[7] Many of the array optimizations are similar to those in [6]. However, there the semantics of arrays is fixed by the compiler. Here, the semantics of dynamic arrays is defined by the specifier.

```
RemoveDuplicates = proc(a: IntArray)
  1   j: Int := a.low
  2   s: IntSet := IntSet_Create()

  3   for i: Int in IntArray_Indexes(a) do
  4       if not IntSet_Member(s, a[i])
  5          then IntSet_Insert(s, a[i])
  6               a[j] := a[i]
  7               j := j + 1
  8          end
  9       end

 10   IntArray_Trim(a, a.low, j - a.low)
      end RemoveDuplicates
```

Fig. 7: Procedure RemoveDuplicates

5 Discussion

All of the optimizations that we have considered here can be hand-coded by the user at the source level. Why, therefore, should one bother using specifications to optimize programs? One reason is that other optimizations that are not expressible at the source level could also benefit from specifications.

Another reason is that relying on the compiler to perform optimizations makes programs easier to build, understand, and maintain. For example, to get the benefit of specialized procedures without using an optimizer, one must introduce a separate interface for each specialized implementation. Each client must decide which interface is appropriate. If they choose an overly general interface, they sacrifice performance, and if they choose an inappropriate specialized interface, they introduce an error. Suppose that later in the life cycle of the program, the reason for the specialized implementation disappears. (E.g., in the original example of Fig. 3, suppose the representation for tables is changed to use binary trees.) Either all interfaces will have to be maintained, or all client code will have to be updated.

Ultimately, the question of whether to use specifications to optimize code boils down to whether the costs of writing formal specifications and running the optimizer justify the benefits of elegant and efficient code. To avoid the cost of writing formal specifications for the entire program, Speckle has features to support partial or missing specifications [17]. This allows a user to amortize costs by focusing on widely-used, lower-level abstractions. Currently, the prototype compiler is too expensive to run outside a research lab; making it practical will require more research.

To guage the potential benefits, we have applied our optimizer to pieces of existing programs. Our initial experience is encouraging. In a study of a program that performs AC-unification, we identified four different specializations whose

pre-conditions were discharged by our compiler in several contexts. Optimizing these contexts resulted in an 11% improvement in speed.

5.1 Debugging

All optimizing compilers complicate debugging because an optimized program differs from the unoptimized one. The only new wrinkle added by our strategy is that the optimized program sometimes calls specialized implementations whose existence was unknown to client code.

A more difficult problem is how to identify bugs in specifications. Because the compiler relies on specifications, bugs in specifications can lead to unsound optimizations. One way to eliminate such errors is to verify the specifications, but there are other possibilities too. The compiler could perform sanity checks on specifications. For example, an interface cannot modify an immutable value. Another possibility is for the user to supply code to check the pre-condition of a specialization and for the debugger to insert this code wherever the optimizer has "proved" that the pre-condition is satisfied. The compiler might list the optimizations and/or the proof obligations that it discharges so that the user could check the list for suspicious ones. Finally, the user could direct the compiler to ignore suspect specifications and see if the problem disappears.

5.2 Status

The Speckle compiler demonstrates that our strategy can improve the efficiency of programs that use abstraction. The compiler detects optimizations that are impossible to find without specifications as well as optimizations that, to our knowledge, no other optimization technique discovers because the optimizations are too difficult. The reason is that specifications conceal irrelevant implementation details, such as benevolent side effects and overly deterministic implementations, which foil techniques based only on code analysis.

Initial experience indicates that when our strategy for defining new optimizations is used, the compiler is able to detect optimizations that improve the performance of mature programs. We are continuing to test the compiler to measure how often it detects legal optimizations and how much the optimizations impact efficiency.

Acknowledgements

We thank John Guttag for his many useful suggestions for both improving this work and improving this paper. We also thank William Weihl, Jeannette Wing, the members of the MIT Systematic Program Development Group, and the referees for their comments on earlier drafts.

Support for this research has been provided in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research Research under contract N00014-89-J-1988, and in part by the National Science Foundation under grant 9115797-CCR.

[1] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, pages 29–41. ACM, January 1979.

[2] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310. ACM, June 1990.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, July 1987.

[4] R. W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–31. AMS, 1967.

[5] S. Garland and J. Guttag. A guide to LP, The Larch Prover. TR 82, DEC Systems Research Center, Palo Alto, CA, December 1991.

[6] R. Gupta. A fresh look at optimizing array bound checking. In *PLDI*, pages 272–282. ACM, June 1990.

[7] J. Guttag, J. Horning (eds.), with S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[8] A. Hisgen. *Optimization of User-Defined Abstract Data Types: A Program Transformation Approach.* PhD thesis, Carnegie-Mellon University, 1985.

[9] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Program Flow Analysis: Theory and Application*, pages 102–131. Prentice-Hall, 1981.

[10] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI*, pages 21–34. ACM, June 1988.

[11] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, Ma, 1986.

[12] J. M. Lucassen. Types and effects: Towards the integration of functional and imperative programming. MIT/LCS/TR 408, August 1987.

[13] A. Neirynck, P. Panangaden, and A. J. Demers. Computation of aliases and support sets. In *POPL*. ACM, 1987.

[14] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL*. ACM, 1988.

[15] W. L. Scherlis. Program improvement by internal specialization. In *POPL*, pages 41–49. ACM, 1981.

[16] B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *TAPSOFT '91 (LNCS 494)*, pages 394–415. Springer-Verlag, April 1991.

[17] M. T. Vandevoorde. Optimizing programs with partial specifications. In *Proceedings of the 1992 Larch Workshop.* Springer Verlag. To appear.