Exploiting Specifications to Improve Program Performance

Mark T. Vandevoorde

©Massachusetts Institute of Technology, 1994

This report is a revised copy of the author's thesis, submitted to the Department of Electrical Engineering and Computer Science on May 28, 1993 in partial fulfillment of the requirements for the degree of Doctor of Philosophy, at the Massachusetts Institute of Technology. The author's current address is: MIT Lab for Computer Science, 545 Technology Square, Cambridge, Ma 02139. Internet: `mtv@lcs.mit.edu`

Abstract

Many approaches to programming emphasize the use of interfaces. The basic idea is to decompose programs into modules and to specify how each module's interface behaves. This makes it easier to reason about programs because one can rely on a module's specification rather than examining its implementation, which is more complicated.

Although programmers benefit from specifications when reasoning about programs, existing compilers do not. In this thesis, I discuss how to incorporate specifications into a programming language to improve performance. I use specifications in two ways: (1) to allow programmers to define new optimizations that make general interfaces more efficient to use, and (2) to enhance conventional optimizations. The specifications can be written incrementally, so programmers can choose to write only the parts of specifications needed to improve performance.

I demonstrate my approach using Speckle, a statically typed, imperative programming language that incorporates specifications. Users define optimizations in Speckle by providing multiple implementations for a single procedure. One implementation must be general enough to work in any context. The other implementations are more efficient but require an additional precondition specified by the user. The compiler uses specifications to prove that particular calls to the procedure can use the specialized implementations.

The prototype Speckle compiler (PSC) incorporates primitive, automated theorem-proving technology to optimize programs. In addition to user-defined optimizations, PSC identifies opportunities to perform three kinds of conventional optimizations: eliminating common subexpressions, moving code out of loops, and eliminating dead code.

Because specifications are simpler than code, PSC detects optimizations that most compilers cannot, such as hoisting procedure calls out of loops. Also, because specifications contain information not found in code, PSC detects optimizations that are impossible without specifications.

Keywords: Formal Specifications, Program Optimization, Compilers, Partial Specifications, Speckle, Larch, Programming Languages, Specification Languages, Theorem-Provers, CLU.

Acknowledgements

If it weren't for the continued guidance and support of John Guttag, this thesis would not have survived its infancy, its adolescence, its youth, or any of the later stages. John has been a patient mentor and a friend. I thank him for his advice and all of his help, and I resolve never to use the "B" word again.

Jim Horning and Bill Weihl gave me many useful comments on how to separate the wheat from the chaff and how to explain my ideas to a broader audience. I am particularly grateful that they read the draft so promptly, and I hope that this thesis does justice to their suggestions.

Many past and present members of the Systematic Program Development group (Daniel Jackson, Steve Garland, Niels Mellergaard, Anna Pogosyants, Mark Reinhold, Raymie Stata, Yang-Meng Tan, and Kathy Yelick) have contributed to my graduate years, both in technical matters and in comradery. Daniel, Kathy, Mark, and Steve deserve special thanks for their help at various times, as does returning SPD member Jeannette Wing.

Many friends outside the lab have helped me with their humor along the way. I thank Amy, Andrew, Jim, Sal, and Tom for the visits and phone calls, and for helping me keep things in perspective. I also thank Greg for his companionship, and for taking me to camp and letting me watch the bears.

Words cannot repay my family for their love and encouragement, without which I would have been lost long ago. They knew when to step in and help, and when to step back and wait. I thank them for their patience, and implore them never to resort to chocolate strikes again!

Finally, I especially thank Colleen for her humor, her support, and her love. When she is finishing her thesis, I hope that I can help her as much as she has helped me.

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.

Contents

List of Figures

Chapter 1

Introduction

Many approaches to programming emphasize the use of module interfaces (or abstractions) [37, 45]. The basic idea is to achieve a separation of concerns. The *client* of an interface looks at its specification and writes code that uses the interface. 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.

Programming with interfaces offers many advantages. The principal advantage is *modularity*. The separation of concerns embodied by a specification allows the client and the implementor each to design, construct, test, and change his code without having to examine the other's code. Such independence is vital in software systems of any appreciable size. A second advantage is *simplicity*. The implementation of an interface is usually more complicated than the specification, so the specification allows clients to reason about the interface at a simpler level. A third advantage is *reuse*. Once an interface is specified and implemented for one system, the interface can be re-used in other systems, thereby reducing the cost of developing software.

Programming with interfaces also presents some challenges. When designing interfaces, a software engineer often faces the dilemma of whether to make an interface general or to specialize the interface to the task at hand [32]. For example, consider the interface of a procedure to insert an element into a set. For efficiency, the interface of `insert` might require that the element not already be in the set—if sets are represented as unsorted lists without duplicates, this would avoid having the implementation examine each element. For generality, however, the interface for `insert` should have no precondition so that `insert` can be called from any context.

This thesis addresses the problem of how to make programming with simple and general interfaces more efficient. The approach is to make specifications available to the compiler by incorporating them into the programming language.

1.1 Specifications Can Improve Performance

Specifications have been advocated primarily for two reasons:

1. *Specifications make it easier to understand code.* A specification only needs to describe the result of a computation rather than how the result is computed. Thus, a specification is usually more compact and easier to understand than an implementation, which might use complicated data structures and invariants to improve efficiency. This makes it easier to understand code that uses the implementation.

2. *Specifications make it easier to change programs.* As the contract between the implementor of an interface and its clients, a specification describes the required behavior of an interface. Without a specification, it is impossible to distinguish the parts of an implementation's behavior that are required from those that can be changed. Thus, specifications make it easier to alter programs in ways that preserve correctness.

Normally, specifications directly benefit people who construct, test, port, and maintain a program, but the specifications are ignored by the compiler. However, specifications are useful at compile time for essentially the same reasons that they are useful to programmers.

In this thesis, I discuss how to incorporate specifications into a programming language to improve performance. I use specifications in two ways: (1) to allow programmers to define new optimizations that make general interfaces more efficient to use, and (2) to enhance conventional optimizations.

Similar benefits in performance might be obtained without specifications, e.g., by writing pragmas or transformation rules. However, the cost of writing specifications is amortized over other uses, such as documenting interfaces. Thus, the effort a programmer spends writing specifications to improve performance also improves modularity, makes it easier to understand code, and encourages reuse.

1.1.1 Defining New Optimizations

Specifications can be used to let programmers define new optimizations to be performed by the compiler. In this thesis, I consider one kind of programmer-defined optimization: a specialized procedure implementation (SPI).

From the client's perspective, calling a procedure with an SPI is like calling any other procedure. From the implementor's perspective, SPIs allow a single procedure interface to have multiple implementations. One implementation—the general implementation—can be used anywhere. The other implementations—the SPIs—are usually faster than the general one but can be used only when certain conditions are met. The programmer defines these conditions formally using the specification language, and the compiler substitutes an SPI for the general implementation when it can prove, using specifications, that the conditions are met at a particular call site.

SPIs reduce the conflict between generality and efficiency. The client sees only a single, general interface while the compiler substitutes a more efficient SPI in contexts where the full generality is unnecessary. For the `insert` example described earlier, the programmer can provide clients with the efficiency of both implementations and the convenience of a single, general interface.

Without SPIs, a programmer might choose to write a separate interface for each procedure implementation. Using SPIs has two advantages. First, because SPIs are not directly accessible to the caller, they eliminate the possibility of a client calling an SPI whose precondition is not satisfied. Second, because they conceal information that might otherwise be visible to clients, SPIs improve modularity. When an SPI is added to or removed from a procedure, the client code only needs to be recompiled. It does not need to be edited.

1.1.2 Enhancing Conventional Optimizations

Specifications can also be used to enhance conventional optimizations. In this thesis, I examine common subexpression elimination, hoisting expressions out of loops, and dead code elimination.

Most conventional compilers restrict common subexpression elimination and code hoisting to expressions that don't contain procedure calls. The reason is that it is difficult to determine when it is safe to eliminate or hoist a procedure call, which may modify or allocate data structures. With specifications, however, it is easy to determine whether a procedure call can be eliminated or whether it performs visible side effects or allocates data.

Specifications also enhance common subexpression elimination when a procedure is called between two common expressions. Without specifications, the compiler must perform interprocedural analysis to determine if the call changes the value of the expression. This information is more readily available in the procedure's specification.

1.2 Speckle

Speckle is a combined programming language and formal specification language that I designed to enhance the efficiency of programs that make use of interfaces. The programming language portion is mostly a subset of CLU [36], and the specification language portion is based on Larch [21, 22].

CLU has several features that make it an appropriate starting point for Speckle. CLU supports both procedural and data abstraction, which are the primary ways to simplify reasoning about programs. CLU has static typing, so there is no need to optimize away runtime type checks. CLU has side effects and pointers,[1] so the compiler must handle aliasing.

I chose Larch because of the tools available for checking and reasoning about Larch specifications [16] and because there were already techniques for specifying CLU programs using Larch [52].

I implemented a prototype Speckle compiler that incorporates parts of a general-purpose theorem-prover, LP [16], to identify opportunities to perform optimizations. The compiler recognizes three kinds of conventional optimizations: common subexpression elimination,[2] moving code out of loops, and dead code elimination. It also identifies opportunities to use SPIs.

1.2.1 Design Goals

Several key ideas drove the design of Speckle. The first idea is that the compiler should use the information supplied in specifications to perform more optimizations. It is the user's responsibility to make sure that the specifications are correct; if they are incorrect, the compiler may perform unsafe optimizations.

[1] In CLU, pointers are implicit, as in LISP, rather than explicit, as in C. Pointer arithmetic is not allowed.

[2] The expressions need not be in the same basic block.

```
remove_duplicates = proc (a: int_array)
    j: int := a.low
    s: int_set := int_set$create()

    for i: int in int_array$indexes(a) do
        if not int_set$member(s, a[i])
           then int_set$insert(s, a[i])
                a[j] := a[i]
                j := j + 1
           end
        end

    int_array$trim(a, a.low, j - a.low)
    end remove_duplicates
```

Figure 1.1: Procedure `remove_duplicates`

The second idea is that specifications and assertions should be optional. The specification language should allow partial specifications, i.e., ones that are either entirely missing or only partially written. Furthermore, the compiler should make use of any relevant information in partial specifications. It is not acceptable for the compiler to ignore specifications until all parts of the program are specified in full, because this may never happen.

The third idea is that programmers should be able to define new optimizations to be performed by the compiler. Speckle currently supports only one kind of user-defined optimization—SPIs.

1.2.2 Example: remove_duplicates

Fig. 1.1 is an example that illustrates some of the methods and ideas of this work. Procedure `remove_duplicates` uses two user-defined data types: `int_set`, a type for integer sets, and `int_array`, a type for integer arrays that can grow and shrink dynamically. As in CLU, the syntactic expressions `a.low`, `a[i]`, and `a[j] :=...` are shorthands for calls to the procedures `int_array$get_low`, `int_array$fetch`, and `int_array$store`. `int_array$trim` takes an array, a starting index, and an element count and discards all elements outside the index range `start...start+count-1`.

Using formal specifications and SPIs, the compiler identifies the following optimizations automatically:

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 specifications of `member` and `Insert` to show that `a` is unchanged since the call to `fetch` on line 4.

2. An SPI of `insert` can be used to avoid checking whether `a[i]` is in `s`. (The `int_set` implementation maintains the invariant that no duplicates occur in the representation of an `int_set`.)

   This optimization relies on the semantics of `if` and the specification of `member` to determine `a[i]` $\notin$ `s`. It also relies on the specification of `fetch` to show that `s` is unchanged between the calls to `member` and `insert`.

3. An SPI of `fetch` can be used to avoid the bounds checks for `a[i]` on line 4.

4. An SPI of `store` can be used to avoid the bounds checks for `a[j]` on line 6.

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

Optimizations 3-5 require proof-by-cases, proof-by-induction, and the specifications of the procedures inside the loop to determine that the bounds of the array are invariant over the loop.

To get theses optimizations, the author of `remove_duplicates` did not have to write any specifications. Instead, the compiler used the specifications of the procedures and data types used in `remove_duplicates`.

On the surface, many of the array optimizations seem similar to those in [20]. However, there is a significant difference. In [20], the compiler relies on the semantics of arrays as defined by the programming language. The technique does not work for optimizations of user-defined data types, e.g., sets. Here, the compiler relies on the specifications of procedures and data types to perform optimizations, so the technique works for any data type.

1.3 Assumptions

This thesis rests primarily on two assumptions. The first is that future compilers will have sufficient computing resources to use theorem-proving technology during compilation. While the prototype compiler is not a practical one, I believe that both improvements in compilation techniques that exploit specifications and advances in computing power will indeed make theorem-proving a practical component of future compilers.

The second assumption is that it is practical to rely on unverified specifications to optimize code. The validity of this assumption rests on the development of techniques for detecting and locating errors in specifications.

1.4 Overview

In Chapter 2, I define the Speckle program state, the central notion that establishes the Speckle model of computation. Then, I describe how to specify data types, procedures, and iterators in Larch/Speckle using program states.

In Chapter 3, I formalize the notion of a program and give proof rules for reasoning about programs using the specifications of Chapter 2.

In Chapter 4, I describe how specifications enhance three conventional optimizations: common subexpression elimination, moving code out of loops, and dead code elimination. I give formal proof obligations for each of these optimizations and show how to discharge the proof obligations using the proof rules of Chapter 3. Many of the improvements rely on improved side effect analysis that would also benefit other optimizations.

In Chapter 5, I present specialized procedures and describe how they reduce the conflict between efficiency and generality. Then, I discuss the need to propagate the proof obligations of specialized procedures up the call stack to preserve modularity.

In Chapter 6, I describe the prototype Speckle compiler (PSC). PSC incorporates primitive automated theorem-proving technology to detect optimizations. The technology is a combination of term rewriting and automated proofs by cases and induction.

In Chapter 7, I extend Larch/Speckle to support partial specifications, and I describe the strategy used by PSC to deduce some of the missing portions of partial specifications.

In Chapter 8, I report on a case study using PSC on pieces of a large program.

Chapter 9 contains a summary and conclusion.

Related work is discussed throughout the thesis. To my knowledge, only Hisgen has previously examined the idea of letting programmers define optimizations in an imperative language [25]. Other closely related works are discussed in Chapters 4 and 5.

Chapter 2

Larch/Speckle

This chapter describes most of the Larch/Speckle specification language—it omits only the features for partial specifications, which are described in Chapter 7.

Section 2.1 provides some essential background about Larch specifications, and Section 2.2 provides some essentials about the Speckle programming language. The remainder describes the semantics of Larch/Speckle using examples and discusses related work on specification languages.

2.1 Larch

Larch is a family of specification languages designed for specifying programs written in one of a number of different programming languages. Larch uses a two-tiered approach. The *shared tier*, which is common to all programming languages, consists of the Larch Shared Language (LSL). LSL is used to define useful functions in a fragment of multisorted first-order predicate logic. The glue between a programming language and LSL is the *interface tier*, which provides an interface language for each programming language, e.g., Larch/CLU [52], Larch/C [22], Larch/C++ [34], etc.

Each interface language formalizes the notion of a program state and provides a syntax and semantics for specifying procedure interfaces and data abstractions.

- A procedure specification is a predicate on pre- and post-states. The predicate, which is defined using LSL functions, specifies the post-states that are possible when the procedure is called from a given pre-state.

- A data abstraction is a module that implements an abstract type, e.g., `set`, using some concrete type, e.g., `hash_table`. The interface for a data abstraction specifies an LSL sort for modeling values of the abstract type in a program state. Thus, program states are more abstract than if they were defined using only the sorts corresponding to primitive types.

2.1.1 The Larch Shared Language

The semantics of LSL is defined precisely in [23]. This section is just an informal synopsis of LSL features that are relevant to Speckle.

LSL specifications are written in units called *traits*. Fig. 2.1 is an example of a trait for sets. A trait begins with a name followed by sort parameters, if any. In Fig. 2.1, sort `S` is used for sets, and sort `E` is used for elements.

The `includes` section lists other traits whose sort and function definitions may be used by the trait. In Fig. 2.1, the `Integer` trait is included to provide the sort `Int` and the functions `+`, $\geq$, `0`, and `1`.

The `introduces` section lists the names and signatures of functions used by the trait. These functions may use either infix notation, like $\_\_\in\_\_$, or prefix notation, like `insert`. Mixfix notation, e.g., `__[__]`, is also allowed.

The `asserts` section lists axioms that hold about the various functions. The axioms may use the builtin Boolean functions, e.g., $\neg$, $\vee$, and $\wedge$. The symbols `==` and `=` are equivalent, except that `==` has lower precedence.

Typically, most of a trait's assertions are given as equations, but two other forms are common. A `generated by` clause defines an induction schema by listing functions that are sufficient to construct all values of a sort. In Fig. 2.1, the `generated by` clause asserts that all values of sort `S` can be constructed using only $\{\}$ and `insert`.

A `partitioned by` clause lists functions that are sufficient for distinguishing unequal values of a sort. In Fig. 2.1, the `partitioned by` clause asserts that two sets are equal if and only if they contain the same elements.

The `implies` section lists formulas that should follow from the axioms using the normal inference rules of predicate logic. Implications are a source of redundant information that can be used to detect inconsistencies and omissions in specifications [15]. In Speckle, implications are used as additional information for proving that an optimization is safe.

The semantics of LSL defines a *theory*—an infinite set of formulas—for a trait. The theory of a trait is the consequence closure of its axioms and inference rules, which include the normal inference rules of predicate logic.

Typically, the theories of traits used in specifications are undecidable—there is no way to tell if an arbitrary formula is in a theory. However,

```
Set (E, S): trait
  includes Integer
  introduces
    {}: → S
    insert: E, S → S
    __∈__: E, S → Bool
    __∪__,
    __∩__: S, S → S
    size: S → Int
  asserts
    ∀ s,s1,s2: S, e,e1,e2: E
      ¬(e ∈ {});
      e1 ∈ insert(e2, s) == e1 = e2 ∨ e1 ∈ s;
      e ∈ (s1 ∪ s2) == e ∈ s1 ∨ e ∈ s2;
      e ∈ (s1 ∩ s2) == e ∈ s1 ∧ e ∈ s2;
      size({}) == 0;
      size(insert(e, s)) == size(s) + (if e ∈ s then 0 else 1);

    S generated by {}, insert
    S partitioned by ∈
  implies
    ∀ e: E, s: S
      e ∈ s == insert(e, s) = s;
      size(s) ≥ 0;
```

Figure 2.1: A Set Trait

theorem-proving techniques can be used to show that some formulas are in the theory of a trait. Also, there is no guarantee that the theory of a trait is consistent. Speckle requires, but cannot check, that all traits used in specifications are consistent.

2.2 Key Aspects of Speckle as a Programming Language

The programming language portion of Speckle is mostly a subset of CLU [36]. It has the following features of CLU:

- *Static Typing.* Type checking is done at compile time, so runtime type checks are not needed.
- *Side Effects.* Programs may modify data structures.
- *Data Abstraction.* Users may define new data types. The implementations of such a type can encapsulate the representation of the type.
- *Garbage Collection.* There is no explicit mechanism to deallocate memory, so dangling references cannot occur.
- *Pointers.* Data values may contain pointers into the garbage-collected store, and pointers can be used to create cyclic data structures. However, pointer arithmetic is not allowed.
- *Procedures.* Procedures may have any number of arguments and any number of results. All arguments and results, which may be pointers, are passed by value.
- *Iterators.* Iterators are a restricted form of coroutines that can be implemented on a single stack.
- *Exceptions.* Routines may terminate either normally or by signalling an exception. If a routine signals an exception that is not handled by the caller, a fatal runtime error occurs. Exceptions may return any number of results.
- *No Global Variable Names.* There is no global scope for variable identifiers.
- *Syntactic Shorthands.* Syntactic shorthands are provided to abbreviate calls to certain procedures. For example, "`a[i]:= e`" denotes a call to `store` operation of the type of `a`. When not followed by "`:=`", "`a[i]`" denotes a call to the `fetch` operation. Similarly, "`x.fld := v`" denotes a call to the `set_fld` operation of the type of `x`, and "`x.fld`" denotes a call to the `get_fld` operation.

As a simplification, I omitted some features of CLU—polymorphism, procedures as data, the type `any`, and `own` variables.

CLU's primitive data types are divided into two categories: immutable and mutable. An instance of an immutable type cannot be modified during the the execution of a program, whereas an instance of a mutable type

```
int = immutable type
   based on Int

int_queue = mutable type
   based on IntQueue
```

Figure 2.2: Data Type Specifications

can be modified. For example, integers and sequences of real numbers are immutable types, while arrays of integers and arrays of reals are mutable. Unlike CLU, Speckle provides a formal way to specify whether a user-defined type is mutable or immutable.[1]

A value of a mutable type is a pointer to a value in the garbage-collected store. Thus, pointers appear implicitly wherever a mutable type is used. This model is similar to LISP, which has implicit pointers, as opposed to C, where pointers are explicit.

2.3 Specifying Interfaces in Larch/Speckle

Larch/Speckle specifications consists of three basic parts: data type specifications, program states, and procedural specifications. Program states are defined in part by data type specifications, which describe the values manipulated by programs. Procedural specifications, i.e., specifications of procedures and iterators, are written as predicates on program states.

2.3.1 Data Type Specifications

The specification of a data type, T, indicates whether T is mutable and specifies an LSL sort, S, whose values are used to model instances of type T. S is called the *value sort* of T.

Fig. 2.2 gives part of two data type specifications. Type `int` is immutable. The `based on` clause specifies that sort `Int` is the value sort of type `int`. Type `int_queue` is mutable, and its value sort is `IntQueue`.

An instance of a mutable type has both a value and an identity. The value may change during a program's execution, but the identity does not.

[1] In CLU, the compiler must assume that a user-defined type is mutable.

To model identities, a *location sort* is implicitly defined for each mutable type. A location sort provides an infinite supply of unique identifiers that get assigned to instances of a mutable type as they are allocated. The identifiers are called locations because they are used as the addresses of data in program states.

In the example above, sort `int_queueLoc` is implicitly defined as the location sort for `int_queue`. This sort can be used in LSL specifications that define other value sorts. For example, consider a data type for sets of `int_queue`s:

```
int_queue_set = immutable type
   based on QueueSet
```

The value sort `QueueSet` would be defined in LSL using `int_queueLoc` for the elements of `QueueSet`s, e.g., by instantiating the `Set` trait of Fig. 2.1 with `int_queueLoc` for `E` and `QueueSet` for `S`.

A term of a value sort may contain locations of mutable data. For example, `QueueSet` terms contain locations of sort `int_queueLoc`. Similarly, a value of an array with a mutable element type contains the location of each element. In contrast, a value of an array of immutable integers contains no locations.

Implementations of data types must not "expose the representation." The representation is exposed if code outside the implementation of the type can access the representation directly, i.e., without calling operations of the type.

2.3.2 Program States

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

$$\begin{array}{ll} \mathit{Prog\ State} & = \mathit{Env}\ X\ \mathit{Store} \\ \mathit{Env} & = \mathit{Ident} \rightarrow (\ \mathit{LSLValue} + \mathit{Loc}\ ) \\ \mathit{Store} & = \mathit{Loc} \rightarrow \mathit{LSLValue} \end{array}$$

The domains LSLValue and Loc are determined by specifications of data types used in a program. LSLValue is the disjoint sum of the value sorts of the types, and Loc is the disjoint sum of the location sorts of the mutable types.

The environment maps identifiers to their values, and the store maps locations of mutable data to their values. An identifier is the name of a program variable, e.g., '`x`' or '`sum`'. If the type of an identifier is immutable, the environment maps the identifier to an LSLValue. If the type of an identifier is mutable, the environment maps the identifier to a location. The environment of program state σ is written as σ^{Env}, and the store is written as σ^{Str}.

Program states have several properties that follow from the features of the programming language. Because Speckle is statically typed, each program state σ is well-typed. If identifier '`x`' is declared to have immutable type `IT`, the sort of $\sigma^{\mathrm{Env}}(\text{'}\mathtt{x}\text{'})$ is the value sort of `IT`. Similarly, if '`x`' is declared to have mutable type `MT`, the sort of $\sigma^{\mathrm{Env}}(\text{'}\mathtt{x}\text{'})$ is `MTLoc`, and the sort of $\sigma^{\mathrm{Str}}(\sigma^{\mathrm{Env}}(\text{'}\mathtt{x}\text{'}))$ is the value sort of `MT`.

There may be several aliases for a given location l. For example, the environment may map any number of identifiers to l. (Each identifier would have the same type as l.) Also, an LSLValue in *range*(σ^{Str}) or *range*(σ^{Env}) may contain l.

Because LSLValues may contain locations, data may be cyclic and may contain multiple levels of indirection. However, memory is reclaimed only by garbage collection, so a program state never has dangling references. I.e., the set of locations contained by values in *range*(σ^{Env}) and *range*(σ^{Str}) is a subset of *domain*(σ^{Str}).

A program can alter its state in two ways. One way is to assign a new value to an identifier. This changes only the environment. Furthermore, it changes only the binding of the assigned identifier, i.e., assigning to identifier '`x`' never affects the binding of identifier '`y`'.

The other way to alter the program state is to call a procedure that modifies locations in the store. A procedure call *modifies* a location l if $\sigma^{\mathrm{Str}}_{\mathrm{pre}}(l) \neq \sigma^{\mathrm{Str}}_{\mathrm{post}}(l)$, where σ_{pre} and σ_{post} are the program states before and after the call. Because of the scope rules, a procedure call never changes the environment except for identifiers that are assigned result values.

2.3.3 Procedure Specifications

The specification of a procedure, `Prc`, is a set of predicates. The precondition, *Prc.Pre*, must hold whenever `Prc` is called. Otherwise, the behavior of the procedure is undefined. The normal postcondition, *Prc.Post*[norm], holds if the `Prc` returns normally. If `Prc` signals exception *sig*, the postcondition *Prc.Post*[*sig*] holds. The guard condition *Prc.Guard*[*sig*] specifies when a procedure is allowed to signal exception *sig*. The guard condition for the normal return, *Prc.Guard*[norm], is the conjunction of the negation of the guards for the exceptional returns.

Predicate	Definition
$Prc.Pre(\sigma^{\mathrm{Str}}_{\mathrm{pre}}, Args)$	`requires`
$Prc.Post[\mathrm{norm}](\sigma^{\mathrm{Str}}_{\mathrm{pre}}, \sigma^{\mathrm{Str}}_{\mathrm{post}}, Args, Res_{\mathrm{norm}})$	`ensures` $\wedge$ `modifies`
$Prc.Post[sig](\sigma^{\mathrm{Str}}_{\mathrm{pre}}, \sigma^{\mathrm{Str}}_{\mathrm{post}}, Args, Res_{sig})$	$\mathtt{ensuring}_{sig} \wedge \mathtt{modifies}$
$Prc.Guard[\mathrm{norm}](\sigma^{\mathrm{Str}}_{\mathrm{pre}}, Args)$	$\bigwedge_{sig} \neg\, (\mathtt{when}_{sig})$
$Prc.Guard[sig](\sigma^{\mathrm{Str}}_{\mathrm{pre}}, Args)$	$\mathtt{when}_{sig}$

Figure 2.3: Procedure Specification Predicates

Syntactically, the predicates are decomposed into a number of clauses. The `requires` clause defines the precondition. The postconditions are defined by the `modifies`, `ensures`, and `ensuring` clauses. The guard conditions are defined by the `when` clauses. Fig. 2.3 summarizes the association between the predicates and the syntactic clauses. *Args* denotes a list of LSL variables—one per formal argument of `Prc`. The sort of each variable is determined by the formal argument's type. For immutable types, the corresponding value sort is used, and for mutable types, the corresponding location sort is used. Similarly, *Res* is used for the results of `Prc`.

Fig. 2.4 lists several example specifications that rely on the data type specifications of Fig. 2.2. The precondition for `dequeue` is vacuous, i.e., `true`. The precondition for `head` is that the queue is not empty. The syntax `q^` denotes the `IntQueue` obtained by dereferencing `q` in the pre-state. Similarly, `q'` denotes the value obtained by dereferencing `q` in the post-state. The superscripts `^` and `'` can be applied to any term denoting a location.

The `modifies` clause specifies the set of locations that a procedure is allowed to modify. Thus, it restricts the side effects that a procedure is allowed to perform, whether the procedure returns normally or signals an exception. For example, `dequeue` may modify only the `int_queueLoc q`, and `head` and `create` may not modify any locations.

Larch/Speckle automatically defines the sort `LocSet` to model sets of locations. `LocSets` are heterogeneous because they may contain locations of more than one mutable data type. In a `modifies` clause, one can write a list of terms to specify the set of locations, S, that a procedure may modify. Each term must be either a location or a `LocSet`, and S is simply the union of the locations and the `LocSets`. Semantically, the `modifies` clause adds

```
dequeue = proc (q: int_queue) returns (i: int)
  requires --
  modifies q
  ensures  q' = deq(q^)  ∧  i = head(q^)
  except
    signals empty when IsEmpty(q^) ensuring q' = q^

head = proc (q: int_queue) returns (i: int)
  requires ¬IsEmpty(q^)
  modifies --
  ensures  i = head(q^)

create = proc () returns (q: int_queue)
  requires --
  modifies --
  ensures  q' = empty  ∧  New(q)
```

Figure 2.4: Sample Procedure Specifications

the conjunct

$$\forall\ l : Loc \in domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}})\ \ [\ l \notin S \implies \sigma^{\mathrm{Str}}_{\mathrm{post}}(l) = \sigma^{\mathrm{Str}}_{\mathrm{pre}}(l)\ ]$$

to a postcondition. This conjunct is abbreviated as the predicate *OnlyModifies*$(pre, post, S)$. In addition, every postcondition has the conjunct

$$domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}}) \subseteq domain(\sigma^{\mathrm{Str}}_{\mathrm{post}})$$

This guarantees that the post-store contains every location in the pre-store.

The `ensures` clause specifies the postcondition for a normal return. For the procedure `head`, the postcondition specifies that the return value is equal to the head of the queue. This assertion is an equation, not an assignment—the same assertion can be written as `head(q^) = i`.

For each exception that a procedure may signal, a `when` clause specifies a guard that must hold for the exception to be signalled, and an optional `ensuring` clause specifies a postcondition. The `ensures` clause does not apply when a procedure signals an exception.

A procedure may list any number of exceptions, and the guards for the exceptions need not be mutually exclusive. If one or more guards are satisfied, the procedure must signal one of the corresponding exceptions. The choice may or may not be implemented non-deterministically.

To specify allocation, the `ensures` and `ensuring` clauses may use the special function `New`. For example, in Fig. 2.4, the specification of procedure `create` states that the return value is a newly allocated location, i.e., that the location is unaliased to all locations in the pre state. In the common case, `New` has a single argument denoting a location. The meaning of `New(x)` is the assertion

$$x \notin domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}}) \;\wedge\; x \in domain(\sigma^{\mathrm{Str}}_{\mathrm{post}})$$

The first conjunct suffices to show that `x` is unequal to all previously existing locations. The second conjunct would be needed to prove that `x` is not a dangling reference in the post-state.

In general, `New` may have any number of arguments, each of which is either a location or a `LocSet`. Each location is treated as a singleton `LocSet`. The meaning of `New(`$ls_1, ls_2, \ldots, ls_n$`)` is that each ls_i contains new locations:

$$\bigwedge_{i=1\ldots n} ls_i \;\cap\; domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}}) = \{\} \;\wedge\; ls_i \;\subseteq\; domain(\sigma^{\mathrm{Str}}_{\mathrm{post}})$$

and that each pair of `LocSet`s in the assertion is disjoint:

$$\bigwedge_{i,j=1\ldots n} i \neq j \implies ls_i \;\cap\; ls_j = \{\}$$

Thus, `New(x,y)` implies `x` $\neq$ `y`, but `New(x)` $\wedge$ `New(y)` does not.

As a convenience when specifying `LocSet`s, Larch/Speckle provides the special functions `reach^` and `reach'`. Both functions map `LocSet`s to `LocSet`s. `reach^(s)` denotes the set of locations reachable from `s` in the pre-store, and `reach'(s)` denotes the set of locations reachable from `s` in the post-store.

The definitions of `reach^` and `reach'` require some way to determine which locations are contained by an LSLValue in the range of a program state's store. Therefore, to fully define `reach^` and `reach'`, each type's value sort must provide a function, `contents`, that maps a term of the value sort to the set of locations contained in the term. Unfortunately, the `contents` functions cannot be inferred mechanically and therefore must be supplied by specifiers.

Fig. 2.5 is an example of a specification for `contents` for mappings. Suppose a program uses a data type that maps strings to mutable

```
Map_Contents: trait
  includes String, Map(String, int_queueLoc, SIQ_Map),
           LocSortFunctions(int_queueLoc)
  introduces
    contents: SIQ_Map → LocSet
  asserts
    ∀ siq: SIQ_Map, s: String, iql: int_queueLoc
     contents(empty_map) == {};                      % the empty LocSet
     contents(bind(siq, s, iql)) == {iql} ∪ contents(siq);
```

Figure 2.5: Specifying `contents`

`int_queue`s. The sort `SIQ_Map` is the value sort for the mapping type. The trait in Fig. 2.5 specifies that the contents of a `SIQ_Map` value is the range of the mapping.

As a shorthand, specifiers may abbreviate `reach^(contents(v))`, where `v` is a term of some value sort, as `reach^(v)`. The same shorthand applies for `reach'`.

2.3.4 Iterator Specifications

Iterators are a restricted form of coroutine that can be used to iterate over collections of values, e.g., sets, mappings, and trees. An iterator can only be invoked by a `for` statement, which defines a loop.

An iterator is first called when the execution of a `for` statement begins. Subsequently, it is resumed each time control reaches the end of the body of the `for` statement. Each time an iterator is called or resumed, it may either yield results, return, or signal an exception. If it yields, another iteration begins. If it returns, control is transferred to the statement after the `for`. If it signals an exception, control is transferred to the handler for the exception.[2]

Iterators are specified much like procedures, but with some additional constructs. Fig. 2.6 contains a iterator specification and Fig. 2.7 contains a procedure that uses the iterator to construct the inverse of a mapping. The `requires` clause of the specification specifies a precondition that must

[2] If there is no handler for the exception, a fatal runtime error occurs.

```
map$elements = iter (m: map) yields (d: dom, r: ran)
   requires --
   modifies --
   ensures  d ∈ domain(m@)  ∧  r = image(m@, d)  ∧  d ∉ d prev
   returns when seq2set(d prev) = domain(m@)
```

Figure 2.6: An Iterator Specification

```
invert = proc (m: map) returns (i: inv_map)
   requires --
   modifies --
   ensures  i' = inverse(m^) & New(i)
     except signals many_to_one when ¬invertible(m^)

   i: inv_map := inv_map$create()
   for d: dom, r: ran in map$elements(m) do
       if inv_map$defined(i, r) then signal many_to_one end
       inv_map$define(i, r, d)
       end
   return(i)
   end invert
```

Figure 2.7: Calling an Iterator

hold each time the iterator is called or resumed. The `modifies` clause specifies the set of locations the iterator is allowed to modify. The `ensures` clause specifies the postcondition that applies each time the iterator yields. The `returns when` clause specifies the guard that determines whether the iterator yields or returns. It may be followed by an `ensuring` clause that specifies a postcondition that holds when the iterator returns.

Iterator specifications may refer to values from previous points in a loop. The syntax `m@` denotes the value obtained by dereferencing the location `m` in the program state in which the iterator was first called. Thus, `m@` and `m^` are synonymous the first time the iterator is called, but may differ when the iterator is resumed. The suffix `@` may be applied to any term denoting a location.

Another way to refer to values at previous points in a loop is with the syntax `T prev`, where `T` is some term. `T prev` denotes the sequence of the values `T` had each time the iterator previously yielded. On the first iteration, `T prev` is an empty sequence. Typically, the `prev` suffix is applied to yielded values. For example, in Fig. 2.6, `d prev` denotes the sequence of `dom` values that were previously yielded. The assertion `d` $\notin$ `d prev` specifies that no domain element is yielded more than once.

Finally, iterator specifications may use the special term `n_iter` to denote the number of times the iterator has yielded to the loop body. This is useful in specifying iterators such as `int$from_to`, which yields a subrange of integers in ascending order.

2.4 Related Work

Larch/Speckle has many features in common with other interface languages, e.g., Larch/CLU [52], Larch/C [22], and Larch/C++ [34], as well as features in common with generic interface languages [8, 24, 35, 53]. Larch/Speckle is unique in that, as explained in Chapter 7, Larch/Speckle supports partial specifications.

Because Speckle is based on CLU, Larch/Speckle is most like Larch/CLU. One difference is the formalization of `New`, which in Larch/CLU is a separate clause that identifies all locations that were allocated by a procedure. In Larch/Speckle, a specification need not identify locations that were allocated to be used as temporaries. Another difference is in the specifications of iterators. Instead of the shorthands `@`, `prev`, and `n_iter`, Larch/CLU provides facilities to introduce names for specification variables, to specify their initial values, and to specify how the values are updated on each iteration.

Larch/Speckle's program states are similar to those in Euclid [31] and FX [38]. In each language, the program store is partitioned into disjoint pieces. In Euclid, the store is composed of disjoint "collections" of data. In FX, the store is composed of disjoint "regions" of data. In Larch/Speckle, the store can be viewed as the union of disjoint mappings—one for each mutable data type. Euclid and FX are more general than Larch/Speckle because they allow programs to use multiple collections or regions of data of a single type.

I chose to use Larch because it came with both a theorem-prover for reasoning about Larch specifications (LP [16]) and techniques for specifying CLU programs using Larch [52]. However, there are many generic specification languages other than Larch that could also be used as the basis for program optimization.

Perhaps the language most similar to Larch is VDM [27, 28]. Both VDM and Larch are designed to facilitate reasoning about specifications in checking designs and to support reasoning about programs. Like LSL, VDM is not tied to a particular programming language. VDM specifications are predicates on abstract program states, whose definition presumably depends on the programming language used. For the purpose of optimization, one drawback is that VDM has nothing akin to an interface language, so one would first have to connect the semantics of a programming language to VDM in a way that allows mechanical analysis. Such analysis might use VDM's generic proof rules for generic programming language constructs like `if` and `while`.

The Z specification language [48] is one of a class of languages designed for reasoning about specifications that are independent of any programming language. To make such specifications useful to a compiler, one would first have to develop an interface language that relates the semantics of Z to that of the programming language.

Another approach in developing specification languages is to design them for particular programming languages. German [18] uses a language tailored for Pascal, and McHugh [41] uses one tailored for Gypsy, a derivative of Pascal. Neither of these languages supports data abstraction as well as CLU. Since data abstraction is a primary way that specifications simplify reasoning about programs, these programming languages seemed less attractive than CLU. Luckham and others designed ANNA [40], a language for annotating Ada programs. Although Ada supports data abstraction, CLU is more attractive because it is simpler than Ada, which has both stack and heap allocation.

Chapter 3

Programs and Proof Rules

This chapter describes the formalization of programs and the proof rules that will be used to discharge the proof obligations for performing various optimizations.

3.1 Limitations and Assumptions

My purpose for formalizing programs and writing proof rules is to provide a framework for a compiler to prove that optimizations are safe. The formalization is not intended to prove properties such as whether a program terminates, as in [47], or whether it references uninitialized variables.

The formalization of the program state relies on interfaces of data types and LSL traits. I assume that no data types have exposed representations, i.e., that reading or modifying a location of one type has no effect on locations of other types.

The proof rules rely on a procedure's specification to define the effects of calling the procedure. Thus, the soundness and completeness of the proof rules depends on the accuracy and completeness of the specifications.

Finally, I assume that the program does not call any procedure whose precondition is not satisfied—this is evident because the proof rules rely on the postconditions without checking preconditions. Although the proof rules could be used to verify that the preconditions are satisfied, this has been extensively studied by others, e.g. [19, 26, 39], and is not a part of my research.

3.2 Programs as Annotated Control Flow Graphs

Hoare rules are a standard way of defining proof rules for programs in terms of their structure [26]. However, in a language with exceptions, it is awkward to use Hoare rules for the same reason that it is awkward to give Hoare rules for statements like `break` and `continue`. Therefore, I use Floyd's approach [13] and define proof rules for Speckle programs that have been converted into control flow graphs (FGs). These rules represent a (partial) semantics for Speckle.

A program is an implementation of a procedure. It has a unique entering edge labeled *enter*, one or more exiting edges, and zero or more internal edges. Each exiting edge corresponds either to a normal return or to signaling an exception. There are six kinds of nodes: assignment, branch, procedure call, iterator call, merge, and loop. Because Speckle is a structured programming language, all FGs are reducible.

Associated with each FG edge e is a program state symbol, σ_{e}, and a theory, $\mathcal{T}_{\mathrm{e}}$. $\mathcal{T}_{\mathrm{e}}$ is induced from the FG using the proof rules. In $\mathcal{T}_{\mathrm{e}}$, σ_{e} denotes an arbitrary member of the set of program states that can occur at edge e. Recall that a theory is an infinite set of formulas. The formulas in $\mathcal{T}_{\mathrm{e}}$ constrain the possible values for σ_{e}. To prove that some predicate P holds at the program point denoted by an edge e, one must prove that the formula $P(\sigma_{\mathrm{e}})$ is in $\mathcal{T}_{\mathrm{e}}$.

The formulas in the theory of an edge, e, also contain the program state symbols for any edge, d, that dominates e.[1] This is useful for defining the program state at edge e in terms of the program states of edges that were traversed to reach e. For example, suppose e is an edge exiting the node `x:= x+1`, and d is the edge entering the node. In $\mathcal{T}_{\mathrm{e}}$, σ_{e} is defined in terms of σ_{d} using formulas such as $\sigma_{\mathrm{e}}^{\mathrm{Env}}(\text{`}\mathtt{x}\text{'}) = \sigma_{\mathrm{d}}^{\mathrm{Env}}(\text{`}\mathtt{x}\text{'}) + \mathit{1}$.

$\mathcal{T}_{\mathrm{e}}$ may contain more formulas constraining σ_{d} than $\mathcal{T}_{\mathrm{d}}$ contains. For example, suppose d enters the branch node `branch b` and e is the exiting edge for when `b` is true. $\mathcal{T}_{\mathrm{e}}$ contains the formula $\sigma_{\mathrm{d}}^{\mathrm{Env}}(\text{`}\mathtt{b}\text{'}) = \mathit{true}$, but $\mathcal{T}_{\mathrm{d}}$ does not (unless the branch is always taken).

In $\mathcal{T}_{\mathrm{e}}$, σ_{e} is analogous to the "collecting state" of edge e in the framework of abstract interpretation [1, 11]. The collecting state of an edge denotes the set of program states that can occur at the edge. Here, $\mathcal{T}_{\mathrm{e}}$ corresponds to the characteristic predicate for the set of program states, and σ_{e} is the predicate's formal variable.

[1] 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 $i \neq j$.

3.2.1 Soundness and Completeness

To ensure soundness, the proof rules must not over-constrain the program states defined by the theories: every formula constraining σ_e in $\mathcal{T}_e$ must be true of every program state that could arise at edge e at runtime. Thus, it is conservative to omit a formula from a theory.

Soundness does not require that each edge's theory be consistent. A theory is inconsistent when it contains the formula $true = false$, which, together with the normal inference rules of predicate logic, can be used to prove that the theory contains all formulas. If $\mathcal{T}_e$ is inconsistent, no program state can satisfy all the formulas constraining σ_e because some of these formulas will contradict one another. If edge e is unreachable at runtime, however, it is sound for $\mathcal{T}_e$ to be inconsistent since σ_e represents the empty set of program states.

To ensure completeness, the proof rules must not under-constrain the program states defined by the theories: the formulas constraining σ_e in $\mathcal{T}_e$ must admit only those program states that can arise at edge e at runtime.

Because the proof rules rely on the specifications of called procedures instead of their implementations, it is sometimes impossible to prove conjectures that are true for a particular implementation of a specification. Thus, the proof rules are in some sense incomplete. The reason is that specifications typically abstract away implementation details that are irrelevant to clients, so a specification often admits more post-states than an implementation will actually generate.

However, the cost of such incompleteness is outweighed by the fact that the specifications simplify reasoning about the program states, which allows the compiler to detect more optimizations. In practice, I found that the details abstracted away by specifications were usually irrelevant to the optimizations considered in this thesis.

I will not prove the soundness or the completeness, modulo abstraction by specification, of the proof rules. Such a proof would require a formal semantics for Speckle and an abstraction function from concrete program states to Larch/Speckle program states.

3.3 Proof Rules for Flow Graphs

To define the theories for each edge in a FG, I use structural induction on flow graphs. First, I define the theory of the entry edge. Next, for each way one or more nodes can be appended to a flow graph, I give a proof rule that defines the theories of the new exiting edges in terms of the theories of the old exiting edges.

The relation $\in$ is used to define the theories at each edge: $F \in \mathcal{T}_{\mathrm{e}}$ means that formula F is in theory $\mathcal{T}_{\mathrm{e}}$. Also, the notation for the hypothesis of a proof rule is extended to include the template of a subgraph appearing in the program.

As a convenience, there are two additional proof rules. The first is that every edge's theory is closed under the usual inferences rules of predicate logic:

$$\frac{F \in \mathit{ConsequenceClosure}(\mathcal{T}_{\mathrm{e}})}{F \in \mathcal{T}_{\mathrm{e}}} \qquad \text{(Closure)}$$

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:

$$\frac{\begin{array}{c} F \in \mathcal{T}_{\mathrm{i}} \\ \text{Edge } i \text{ dominates edge } j \end{array}}{F \in \mathcal{T}_{\mathrm{j}}} \qquad \text{(Extension)}$$

This rule propagates the formulas defining σ_{i} in $\mathcal{T}_{\mathrm{i}}$ to $\mathcal{T}_{\mathrm{j}}$, so it allows $\mathcal{T}_{\mathrm{j}}$ to define σ_{j} in terms of σ_{i}.

A related invariant that follows from the proof rules is that a theory $\mathcal{T}_{\mathrm{j}}$ can contain formulas constraining σ_{i} only if i dominates j.

3.3.1 Entry Edge

The theory $\mathcal{T}_{\mathrm{enter}}$ of the entering edge comes from the specifications of procedures and data types used in the FG. $\mathcal{T}_{\mathrm{enter}}$ is the consequence closure of the union of:

1. the theories of all LSL specifications used by the program;

2. the theory of the program state and procedure predicates defined by Larch/Speckle; and

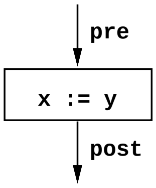
3. the precondition specified by the user, if any, for entering the FG.

   The precondition comes from the `requires` clause of the procedure implemented by the program.

The extension proof rule ensures that every edge's theory is an extension of $\mathcal{T}_{\text{enter}}$, so $\mathcal{T}_{\text{enter}}$ contains formulas that are true globally.

3.3.2 Assignment Nodes

An assignment node has exactly one entering and one exiting edge. The left and right sides of the assignment must each consist of a single identifier. The proof rule for an assignment is:



$$\frac{}{\begin{array}{ll} \sigma^{\text{Str}}_{\text{post}} = \sigma^{\text{Str}}_{\text{pre}} & \in \mathcal{T}_{\text{post}} \\ \sigma^{\text{Env}}_{\text{post}}(\text{`}\mathtt{x}\text{'}) = \sigma^{\text{Env}}_{\text{pre}}(\text{`}\mathtt{y}\text{'}) & \in \mathcal{T}_{\text{post}} \\ \mathit{OnlyAssigns}(\sigma^{\text{Env}}_{\text{pre}}, \sigma^{\text{Env}}_{\text{post}}, \text{`}\mathtt{x}\text{'}) & \in \mathcal{T}_{\text{post}} \end{array}}$$

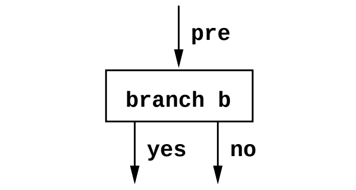
The first consequent states that the store is unchanged—this formalizes the fact that assignment never affects the store. The second consequent states that the value of `x` after the assignment is equal to the value of `y` before the assignment. The third consequent states that the values of all Idents other than `x` are not affected by the assignment. The meaning of $\mathit{OnlyAssigns}(\sigma^{\text{Env}}_{\text{pre}}, \sigma^{\text{Env}}_{\text{post}}, \text{`}\mathtt{x}\text{'})$ is

$$\forall \mathit{var} : \mathit{Ident}[\mathit{var} \neq \text{`}\mathtt{x}\text{'} \Longrightarrow \sigma^{\text{Env}}_{\text{post}}(\mathit{var}) = \sigma^{\text{Env}}_{\text{pre}}(\mathit{var})]$$

This captures the fact that identifiers are never aliased in Speckle. The *OnlyAssigns* predicate is analogous to the *OnlyModifies* predicate of Section 2.3.3. The difference is that the former constrains the environment, while the latter constrains the store.

3.3.3 Branch Nodes

Branch nodes are used to represent `if` statements. The branch condition must be an identifier. The proof rule for branch nodes is:



$$
\begin{array}{ll}
\sigma_{\text{pre}}^{\text{Env}}(\text{‘}\mathtt{b}\text{’}) = true & \in \mathcal{T}_{\text{yes}} \\
\sigma_{\text{yes}} = \sigma_{\text{pre}} & \in \mathcal{T}_{\text{yes}} \\
\sigma_{\text{pre}}^{\text{Env}}(\text{‘}\mathtt{b}\text{’}) = false & \in \mathcal{T}_{\text{no}} \\
\sigma_{\text{no}} = \sigma_{\text{pre}} & \in \mathcal{T}_{\text{no}}
\end{array}
$$

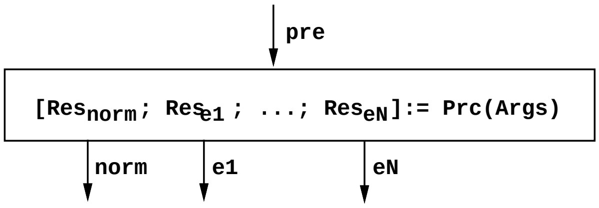
The first and third consequents capture the control-dependent information of whether the branch test was true or false. These contradictory formulas would create an inconsistency if they were in the same theory. It is for this reason that I use a theory per edge rather than one theory for entire flow graph.

The symbol σ_{pre} has different different meanings in the theories $\mathcal{T}_{\text{pre}}$, $\mathcal{T}_{\text{yes}}$, and $\mathcal{T}_{\text{no}}$. In $\mathcal{T}_{\text{pre}}$, σ_{pre} denotes an arbitrary program state at edge *pre*. In $\mathcal{T}_{\text{yes}}$, σ_{pre} denotes an arbitrary program state at edge *pre* that causes the branch to be taken. Similarly, in $\mathcal{T}_{\text{no}}$, σ_{pre} denotes an arbitrary program state at edge `pre` that causes the branch to be not taken.

3.3.4 Procedure Call Nodes

A procedure call node has exactly one entering edge, but it may have several exiting edges because a procedure may terminate either normally or by signalling an exception. Either form of termination may return results.

The proof rule for a call to a procedure `Prc` that may signal exceptions `e1` ... `eN` is:



For status = norm, e1, ..., eN

$$
\begin{array}{ll}
Prc.Post[status](\sigma^{\mathrm{Str}}_{\mathrm{pre}}, \sigma^{\mathrm{Str}}_{\mathrm{status}}, Args, Res_{status}) & \in \mathcal{T}_{\mathrm{status}} \\
Prc.Guard[status](\sigma^{\mathrm{Str}}_{\mathrm{pre}}, Args) & \in \mathcal{T}_{\mathrm{status}} \\
OnlyAssigns(\sigma^{\mathrm{Env}}_{\mathrm{pre}}, \sigma^{\mathrm{Env}}_{\mathrm{status}}, Res_{status}) & \in \mathcal{T}_{\mathrm{status}}
\end{array}
$$

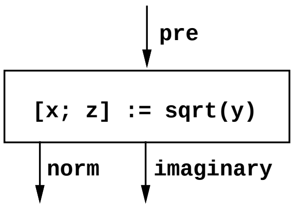
Args denotes the arguments to `Prc`, if any. Only the value of an identifier can be passed as argument, so temporary identifiers are introduced for more complex expressions. Res_{status} denotes the results, if any, of `Prc` when it terminates with status *status*. (See Section 2.3.3 on p. 27 for a definition of `Prc`'s predicates.)

Example: Calling a Procedure

The following is a specification of procedure to compute the square root of a `real` number:

```
sqrt = proc (r1: real) returns (r2: real)
   requires --
   modifies --
   ensures  square(r2) ≃ r1
   except
     signals imaginary (c: complex)
     when r1 < 0 ensuring square(c.imag) ≃ -r1  ∧  c.real = 0
```

Here is a flow graph with only a call to the procedure `sqrt`:



This flow graph corresponds to the Speckle code fragment:

```
x := sqrt(y)
  except when imaginary(z: complex): ...
```

where `x` and `y` are type `real` and `z` is type `complex`.

The proof rule for procedure calls implies that the several formulas are in $\mathcal{T}_{\text{norm}}$. (Recall that the postcondition is the conjunction of the `modifies` and `ensures` clauses.)

$$\forall\ l : \mathit{Loc} \in \mathit{domain}(\sigma^{\text{Str}}_{\text{pre}})\ \ [\ \sigma^{\text{Str}}_{\text{norm}}(l) = \sigma^{\text{Str}}_{\text{pre}}(l)\ ] \qquad (\texttt{modifies})$$

$$\texttt{square}(\sigma^{\text{Env}}_{\text{norm}}(\text{`}\texttt{x}\text{'})) \simeq \sigma^{\text{Env}}_{\text{pre}}(\text{`}\texttt{y}\text{'}) \qquad (\texttt{ensures})$$

$$\neg(\sigma^{\text{Env}}_{\text{pre}}(\text{`}\texttt{y}\text{'}) < 0) \qquad (\text{negation of } \texttt{when} \text{ guard})$$

$$\mathit{OnlyAssigns}(\sigma^{\text{Env}}_{\text{pre}}, \sigma^{\text{Env}}_{\text{norm}}, \text{`}\texttt{x}\text{'})$$

Similarly, the proof rule implies that the following formulas are in $\mathcal{T}_{\text{imaginary}}$:

$$\forall\ l : \mathit{Loc} \in \mathit{domain}(\sigma^{\text{Str}}_{\text{pre}})\ \ [\ \sigma^{\text{Str}}_{\text{imaginary}}(l) = \sigma^{\text{Str}}_{\text{pre}}(l)\ ] \qquad (\texttt{modifies})$$

$$\texttt{square}(\sigma^{\text{Env}}_{\text{imaginary}}(\text{`}\texttt{z}\text{'}).\texttt{imag}) \simeq -\ \sigma^{\text{Env}}_{\text{pre}}(\text{`}\texttt{y}\text{'}) \\ \wedge\ \ \sigma^{\text{Env}}_{\text{imaginary}}(\text{`}\texttt{z}\text{'}).\texttt{real} = 0 \qquad (\texttt{ensuring})$$

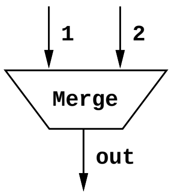
$$\sigma^{\text{Env}}_{\text{pre}}(\text{`}\texttt{y}\text{'}) < 0 \qquad (\texttt{when} \text{ guard})$$

$$\mathit{OnlyAssigns}(\sigma^{\text{Env}}_{\text{pre}}, \sigma^{\text{Env}}_{\text{norm}}, \text{`}\texttt{z}\text{'})$$

3.3.5 Merge Nodes

A merge node has two or more entering edges but only one exiting edge. Merge nodes are used to join flows of control that separated because of branching or calls to procedures that signal exceptions. A merge node may not be used to create a loop, i.e., the entering edges must not be backward edges.[2] Loop nodes will be used to merge backward edges.

The proof rule for merge nodes is:



$$\frac{F[\sigma_1/\sigma_{\text{out}}] \in \mathcal{T}_1 \qquad F[\sigma_2/\sigma_{\text{out}}] \in \mathcal{T}_2}{F \in \mathcal{T}_{\text{out}}}$$

The notation $F[\sigma/\sigma_{\text{i}}]$ denotes F with σ substituted for σ_{i} and with bound variables renamed to avoid capture.

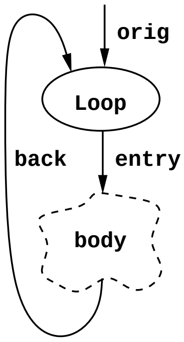
The rule for merge nodes is merely an instance of proof-by-cases. To prove that a formula, F, is true about the program state exiting a merge node, one must prove that the formula is true of each program state that enters the merge node.

3.3.6 Loop Nodes

A loop node has two entering edges, *orig* and *back*, and a single exiting edge, *entry*. The edge *orig* is the header of the loop, *entry* is the first edge of the body, and *back* is the backward edge that comes from the end of the body. The body itself is a control flow graph and typically has an edge that exits the loop.

The proof rule for loop nodes is:

[2] A backward edge is one whose target node dominates its source node.



$$\frac{\begin{array}{ll} F[\sigma_{\mathrm{orig}}/\sigma_{\mathrm{entry}}] & \in \mathcal{T}_{\mathrm{orig}} \\ F \Longrightarrow F[\sigma_{\mathrm{back}}/\sigma_{\mathrm{entry}}] & \in \mathcal{T}_{\mathrm{back}} \end{array}}{F \in \mathcal{T}_{\mathrm{entry}}}$$

This rule is merely an instance of proof-by-induction. To prove that a formula, F, is true about the program state at the entry edge, one must prove that F is true of the program state at the header of the loop and that the body of the loop preserves the truth value of F.

3.3.7 Iterator Call Nodes

Calls to iterators are basically the same as calls to procedures. One minor difference is that the normal form of termination for an iterator is to yield results. The `returns when` guard is treated just like an exception that has no result values. Another minor difference is that iterator specifications may use the suffix `@` to refer to the program store that existed at the loop header, i.e., when the iterator was first called. This is handled by making $\sigma^{\mathrm{Str}}_{\mathrm{orig}}$ an argument of the precondition, the postcondition, and the guard conditions.

For each iterator call node, I add a specification variable to count the number of iterations. The variable is initialized to 0 by an assignment above the header edge, and the variable is incremented after the iterator call node. The value of the variable is given as an argument to iterator preconditions and postconditions that refer to `n_iter`.

Similarly, for each term t suffixed by `prev` in the iterator's interface, I introduce a specification variable. The variable is initialized to the empty sequence by an assignment above the header, and, after the iterator call node, the variable is updated by appending the value of t to the end of the sequence. The value of the variable is given as an argument to the iterator preconditions and postconditions that refer to `t prev`.

Example: Calling an Iterator

The following code iterates over the bindings in a mapping to construct the domain and the range.

```
dom: int_set := int_set$create()
ran: str_set := str_set$create()
for d: int, r: string in bindings(m) do
   int_set$insert(dom, d)
   str_set$insert(ran, r)
   end
```

The specification for the iterator `bindings` is:

```
bindings = iter (m: int_string_map)
             yields (i: int, s: string)
   requires --
   modifies --
   ensures  image(m@, i) = s ∧ i ∉ i prev
   returns when size(m) = n_iter
```

Note that this specification refers to both `n_iter` and `i prev`.

The flow graph for the code is shown in Fig. 3.1.

3.4 Summary and Related Work

The proof rules for programs are a form of Hoare rules [26] for programs that are formalized as flow graphs, as in [13], rather than parse trees. The proof rules rely on data type specifications to define the program state (see Chapter 2), and on procedure specifications to describe the effect of procedure calls. This simplifies reasoning about programs because the program states are more abstract (because of data abstraction) and because the effect of a procedure call does not have to be approximated by interprocedural analysis.

For each edge in a flow graph, the proof rules define an LSL theory that constrains the possible values of program states that can arise at the edge. The set of possible program state values corresponds to the "collecting state" in the literature on abstract interpretation [1, 11]. However, the proof rules are more powerful than the data flow framework used in abstract interpretation.

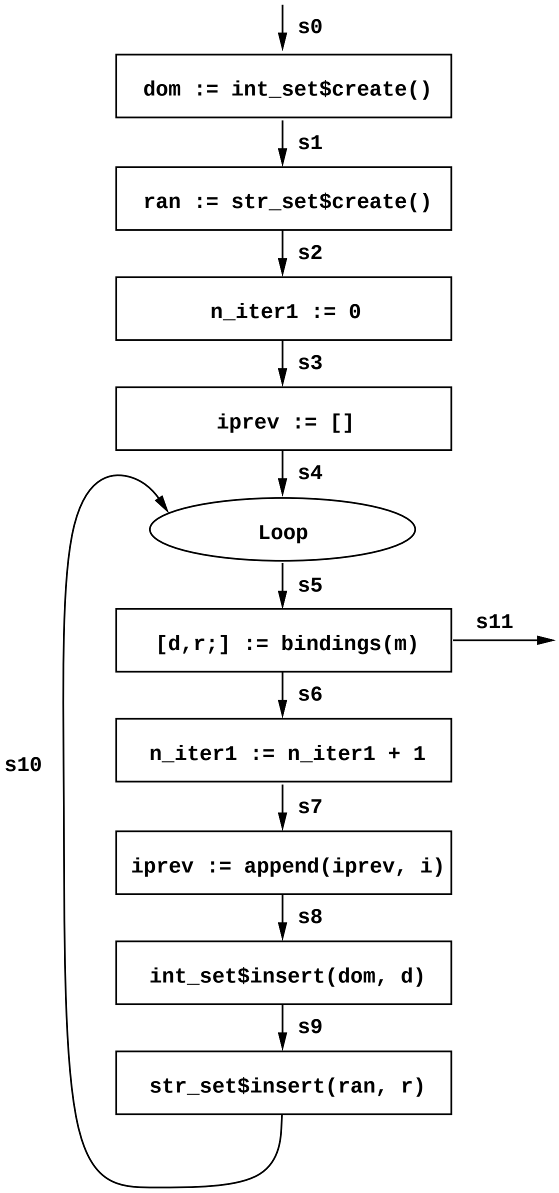


Figure 3.1: A Flow Graph with a Call to an Iterator

Chapter 4

Enhancing Conventional Optimizations

In this chapter, I describe how to exploit specifications to enhance three kinds of conventional optimizations—common subexpression elimination, hoisting expressions out of loops, and dead code elimination. For each kind of optimization, I give formal proof obligations that are sufficient to preserve the correctness of the original program, and I give some examples of how to discharge the proof obligations using the specifications and proof rules of chapters 2 and 3. In Chapter 6, I present a strategy for mechanically discharging the proof obligations.

Specifications permit optimizations that are impossible by analyzing only code because they abstract away irrelevant implementation details. Furthermore, because specifications are simpler than code, they facilitate optimizations that are difficult to perform by analyzing only code.

4.1 Common Subexpression Elimination

Common subexpression elimination is a conventional optimization for reusing results that were previously computed. For example, in the code

```
x := a[i]
    ⋮
y := a[i]
```

the second occurrence of `a[i]` can be eliminated provided that the value of `a[i]` is unchanged.[1] The compiler may replace `a[i]` by `x` or, if `x` is assigned between the occurrences of `a[i]`, by a temporary variable introduced by the compiler.

[1] If the two occurrences of `a[i]` lie in different basic blocks, this optimization is sometimes called *global common subexpression elimination* [2].

While common subexpressions are often syntactically identical, as in the example above, they may be syntactically different, as in the code fragment

```
x := a+b
y := b
z := y+a
```

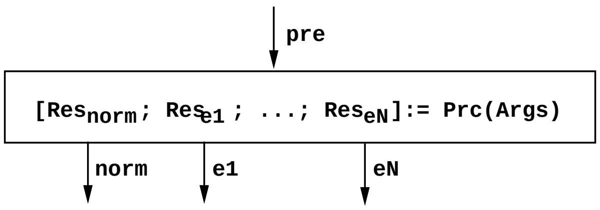
Here, `y+a` can be replaced by `x` if, as is the case in Speckle, assignment to one identifier cannot change the value of another.

Although compilers are good at eliminating expressions that use only primitive operations like `__+__` and `__[__]`, they are less effective at eliminating procedure calls. The problem is that while the semantics of primitive operations are simple and known to the compiler writer, the semantics of calling an arbitrary procedure must typically be determined by examining the procedure's implementation, which may require interprocedural analysis of the whole program.

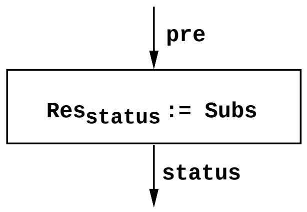
As discussed in Chapter 1, a key idea in Speckle is to use specifications to define the semantics of procedure calls. In fact, for the purpose of source-level optimization, even primitive operations like `__+__` and `__[__]` are treated as calls to procedures whose specifications are supplied as part of the language. Thus, Speckle has only three kinds of expressions: literals, identifiers, and results of procedure calls.

4.1.1 Proof Obligations

In Speckle, common subexpression elimination constitutes replacing a call to a procedure:



by an assignment:



where, according to the specification of `Prc` and the values *Args*, *status* is a legal termination status for the call, and where *Subs* are available values that satisfy the postcondition of `Prc` when it terminates with *status*.

A value is *available* at edge e if it is bound to an identifier at an edge d that dominates e. If the identifier to which the value is bound might be assigned a different value between edges i and j, the compiler must introduce a temporary to save the value. When attempting to eliminate a call, the compiler can try to substitute any available value of the proper type. The available values include any value that was computed at nodes that dominate the call.

To prove that it is legal for the call to *Prc* to terminate with status *status*, the compiler must prove

$$Prc.Guard[status](\sigma^{\mathrm{Str}}_{\mathrm{pre}}, Args) \in \mathcal{T}_{\mathrm{pre}}$$

To prove that the substitute values satisfy the postcondition and that the call need not modify or allocate any locations in the store, the compiler must prove

$$Prc.Post[status](\sigma^{\mathrm{Str}}_{\mathrm{pre}}, \sigma^{\mathrm{Str}}_{\mathrm{pre}}, Args, Subs) \in \mathcal{T}_{\mathrm{pre}}$$

Note that $\sigma^{\mathrm{Str}}_{\mathrm{pre}}$ is passed as both the pre- and post-store to the postcondition since an assignment has no effect on the store.

The proof obligations above are sufficient to preserve the correctness of the original program under the assumption that the caller is relying on only the specification of `Prc`, not its implementation. The proof obligations demonstrate that `Prc`'s specification permits an implementation to terminate the call with status *status*, with results *Subs*, and without modifying or allocating locations in the store. Thus, it is safe to replace the call by the assignment, to transfer control to edge *status*, and to delete all other outgoing edges.

```
can_link_ends = proc (pa, pb: polymer) returns (b: bool)
  return(
    monomer$can_link(pa.left,  pb.left) cor
    monomer$can_link(pa.left,  pb.right) cor
    monomer$can_link(pa.right, pb.left) cor
    monomer$can_link(pa.right, pb.right)
  )
    except when not_linear: return(false) end
```

Figure 4.1: Procedure `can_link_ends`

```
can_link_ends = proc (pa, pb: polymer) returns (b: bool)
  begin
    al,bl,ar,br: monomer
    al := pa.left
    bl := pb.left
    if monomer$can_link(al, bl) then return(true) end
    br := pb.right
    if monomer$can_link(al, br) then return(true) end
    ar := pa.right
    if monomer$can_link(ar, bl) then return(true) end
    return(monomer$can_link(ar, br))
    end except when not_linear: return(false) end
```

Figure 4.2: Hand-optimized Version of `can_link_ends`

4.1.2 Example: can_link_ends

Fig. 4.1 is an example that illustrates the idea of eliminating procedure calls. Procedure `can_link_ends` manipulates polymers and monomers. A monomer is a chemical compound used as a building block to make polymers. Here, a polymer is either linear, meaning a sequence of monomers, or cyclic, meaning a ring of monomers. The procedure `can_link_ends` takes two polymers and returns a boolean to indicate whether the two polymers could be joined to form one linear polymer by linking together monomers from an end of each polymer. If either polymer is cyclic, `can_link_ends` returns

```
monomer = immutable type

  based on Monomer

  can_link = proc (m1, m2: monomer) returns(b: bool)
    requires --
    modifies --
    ensures  b = (m1 ⋆ m2)

    ⋮

polymer = mutable type spec

  based on Polymer

  get_left = proc (p: polymer) returns (m: monomer)
    requires --
    modifies --
    ensures  m = p^.left
    except signals not_linear when ¬is_linear(p^)

  get_right = proc (p: polymer) returns (m: monomer)
    requires --
    modifies --
    ensures  m = p^.right
    except signals not_linear when ¬is_linear(p^)

  bond_right = proc (p: polymer, m: monomer)
    requires --
    modifies p
    ensures p' = (p^ ⊢ m)
    except
      signals not_linear when ¬is_linear(p^) ensuring p' = p^
      signals bad_bond when ¬(p^.right ⋆ m) ensuring p' = p^

    ⋮
```

Figure 4.3: Monomer and Polymer Interfaces

```
Monomer: trait
  includes Int, Commutative(⋆, Monomer, Bool)
  Monomer enumeration of EG,   % ethylene glycol
                         TTA,  % teraphthalic acid
                         AA,   % adipic acid
                         HMD   % hexamethyldiamine
  introduces                   % The bonding relation:
    __ ⋆ __: Monomer,          %  m1 ⋆ m2 ==
             Monomer → Bool    %   "m1 will bond with m2"
  asserts forall m: Monomer
      ¬(m ⋆ m);
      EG ⋆ TTA;     EG ⋆ AA ;  ¬( EG ⋆ HMD);
                ¬(TTA ⋆ AA);    TTA ⋆ HMD ;
                                 AA ⋆ HMD ;
Polymer: trait
  includes Monomer, Cycle(Monomer, MonomerSeq, MonomerCycle)
                        % MonomerSeqs are non-empty sequences
  Polymer union of linear: MonomerSeq, cyclic: MonomerCycle
  introduces
    __ .left,
    __ .right: Polymer          → Monomer
    is_linear: Polymer          → Bool
    __ ⊢ __:  Polymer, Monomer → Polymer   % adds a monomer
    lin2cyc:   Polymer          → Polymer
    __ ⋆ __:   Monomer, Polymer → Bool  % bonding predicates
    __ ⋆ __:   Polymer, Polymer → Bool
  asserts
    forall p,p1,p2: Polymer, m,m1,m2: Monomer, ms: MonomerSeq
      p1 ⋆ p2 == is_linear(p1)
                 ∧ (p1.head ⋆ p2 ∨ p1.tail ⋆ p2);
      m ⋆ p == is_linear(p) ∧ (m ⋆ p.head ∨ m ⋆ p.tail);

      linear(ms).left == first(ms);
      linear(ms).right == last(ms);
      is_linear(p) == tag(p) = linear;
      linear(ms) ⊢ m == linear(ms ⊢ m);
      lin2cyc(linear(ms)) == cyclic(seq2cycle(ms));
```

Figure 4.4: Monomer and Polymer Traits

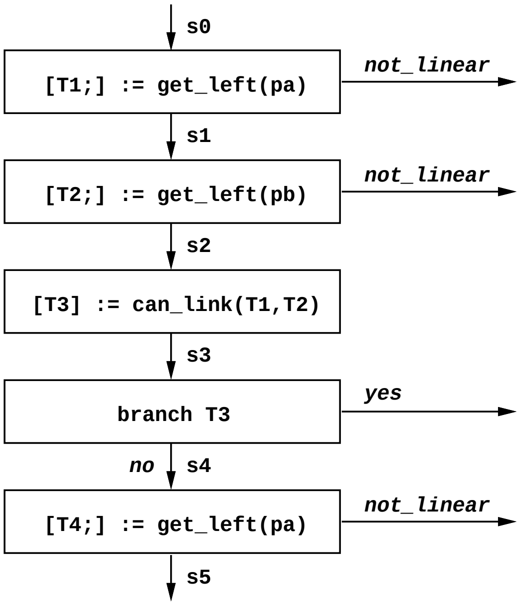


Figure 4.5: Part of FG for `can_link_ends`

false since cyclic polymers have no bonding sites available. Fig. 4.2 is a less legible version of `can_link_ends` where redundant procedure calls have been eliminated by hand.

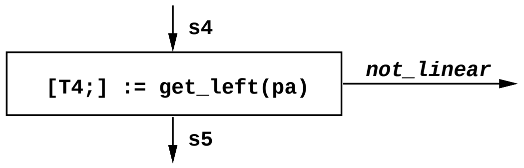
The procedure `can_link_ends` calls the procedures `monomer$can_link`, `polymer$get_left`, and `polymer$get_right`, whose specifications appear in Fig. 4.3.[2] The pertinent traits for these interfaces are in Fig. 4.4. The operator `cor` is short-circuit or.

From the interfaces in Fig. 4.3, it is clear that half of the calls to `get_left` and `get_right` are redundant. Both procedures may be called twice per polymer even though the second call returns the same result as the first call since the polymers are never modified.

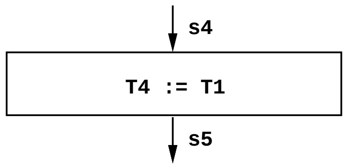
I will explain how to eliminate the second call to `get_left(pa)`; similar analysis can eliminate the other redundant calls. Fig. 4.5 shows the FG for `can_link_ends` from the entry edge to the second call to `get_left`. Temporary identifier names have been introduced systematically so that the arguments to procedures are either literals or identifiers.

[2] Recall that the syntax `pa.left` is a syntactic shorthand for `polymer$get_left(pa)`.

The goal is to replace



by



The first obligation is to prove that the call will return normally:

$$\texttt{get\_left}.\mathit{Guard[norm]}(\sigma_4^{\mathrm{Str}}, \sigma_4^{\mathrm{Env}}(\text{`}\texttt{pa}\text{'})) \in \mathcal{T}_4$$

Given the specification of `get_left`, this is equivalent to

$$\mathit{is\_linear}(\sigma_4^{\mathrm{Str}}(\sigma_4^{\mathrm{Env}}(\text{`}\texttt{pa}\text{'}))) \in \mathcal{T}_4$$

Simple analysis can prove the lemma

$$\sigma_4^{\mathrm{Str}}(\sigma_4^{\mathrm{Env}}(\text{`}\texttt{pa}\text{'})) = \sigma_0^{\mathrm{Str}}(\sigma_0^{\mathrm{Env}}(\text{`}\texttt{pa}\text{'})) \in \mathcal{T}_4$$

This lemma follows from the fact that `pa` is never the target of an assignment and from the `modifies` clauses of the procedures called between edges 0 and 5. This lemma can be used to simplify the goal to:

$$\mathit{is\_linear}(\sigma_0^{\mathrm{Str}}(\sigma_0^{\mathrm{Env}}(\text{`}\texttt{pa}\text{'}))) \in \mathcal{T}_4$$

This formula is in $\mathcal{T}_1$ because it is the guard for the normal return of the first call to `get_left`. Because edge 1 dominates edge 5, the extension proof rule propagates this formula to $\mathcal{T}_4$, so the goal is discharged.

The second proof obligation is to show that the value of `T1` at edge 1 satisfies the postcondition of the call:

$$\texttt{get\_left}.\mathit{Post[norm]}(\sigma_4^{\mathrm{Str}}, \sigma_4^{\mathrm{Str}}, \sigma_4^{\mathrm{Env}}(\text{`}\texttt{pa}\text{'}), \sigma_1^{\mathrm{Env}}(\text{`}\texttt{T1}\text{'})) \in \mathcal{T}_4$$

Given the specification of get_left, this is equivalent to two subgoals. The first, from the modifies clause, is

$$domain(\sigma_4^{\text{Str}}) \subseteq domain(\sigma_4^{\text{Str}}) \ \bigwedge \ \forall\, l \in domain(\sigma_{\text{pre}}^{\text{Str}}) \ \ [\ \sigma_4^{\text{Str}}(l) = \sigma_4^{\text{Str}}(l)\ ] \in \mathcal{T}_4$$

which is trivially true. The second, from the ensures clause, is

$$\sigma_1^{\text{Env}}(\text{`}\mathtt{T1}\text{'}) = \sigma_4^{\text{Str}}(\sigma_4^{\text{Env}}(\text{`}\mathtt{pa}\text{'})) \ \in \mathcal{T}_4$$

Using the lemma from before, the right side of the equation can be replaced to yield

$$\sigma_1^{\text{Env}}(\text{`}\mathtt{T1}\text{'}) = \sigma_0^{\text{Str}}(\sigma_0^{\text{Env}}(\text{`}\mathtt{pa}\text{'})) \ \in \mathcal{T}_4$$

This formula is in $\mathcal{T}_1$ because it is part of the postcondition of the first call to get_left. Because edge 1 dominates edge 5, the extension proof rule propagates this formula to $\mathcal{T}_4$, so the goal is discharged. Thus, the optimization is safe.

4.1.3 Syntactically Distinct Expressions

In the previous example, each call that was eliminated was syntactically identical to a previous call. In general, this is not necessary.

For example, in the code

```
bond_right(p, monomer_bag$choose(ms))
monomer_bag$delete(ms,p.right)
```

p.right can be replaced by the result of choose. The pertinent interfaces are those of get_right and bond_right, which adds a monomer to the right end of a polymer. When bond_right returns normally, p must be linear, so p.right must return normally. Furthermore, the second argument to bond_right becomes the right end of p, so it can replace p.right.

4.1.4 Optimizations Impossible without Specifications

The proof obligations for common subexpression elimination allows optimizations that would be considered unsound by ordinary code analysis. For example, consider the code

```
i1 := intset$least(s)
i2 := intset$choose(s)
```

The specifications in Fig. 4.6 allow the compiler to replace `choose` by `i1`. The key here is that the specification of `choose` is non-deterministic—more than one value can satisfy the postcondition. Furthermore, the specification does make any guarantees about "randomness." The optimization is sound because the caller can only rely on the specification of `choose`, which might always return the least element.

Unless `choose` is in fact implemented as `least`, the optimization appears unsound without the specifications because it might alter the result computed by the program.

Although it is interesting that specifications enable optimizations that are otherwise impossible, it is unclear how often such optimizations can be applied in practice.

4.2 Hoisting Expressions out of Loops

Hoisting expressions out of loops is a conventional optimization related to common subexpression elimination. The basic idea is to move loop-invariant expressions out of loops. For example, in the code

```
while b do
   x := a[i]
   ⋮ % Code that doesn't modify a or assign a or i.
   end
```

`a[i]` need only be computed once, rather than once per iteration. Thus, the code can be replaced by

```
first := true
while b do
   if first then
      save := a[i]
      first := false
      end
   x := save
   ⋮ % Code that doesn't modify a or assign a or i.
   end
```

where `first` and `save` are new identifiers generated by the compiler. In some cases, the compiler may be able to dispense with `first` and compute

```
intset = mutable type

  based on IntSet

  least = proc (s: intset) returns (i: int)
    requires --
    modifies --
    ensures  i = smallest(s^)
    except signals empty when s^ = {}

  choose = proc (s: intset) returns (i: int)
    requires --
    modifies --
    ensures  i ∈ s^
    except signals empty when s^ = {}
```

Figure 4.6: Interface for `least` and `choose`

`a[i]` before the `while`, but in general, this may be unsound (suppose, for example, `b` implies `inbounds(a,i)`) or inefficient (suppose `b` is false).

As described above, common subexpression elimination suffices to handle the case where available values computed before a loop can be used to eliminate a procedure call inside a loop. For example, common subexpression elimination can eliminate the second occurrence of `a[i]` in the code

```
x := a[i]
while true do
   y := a[i]
   ⋮ % Code that doesn't modify a or assign a or i.
   end
```

To handle the general case, the compiler needs a way to determine when the results of a procedure called in the first iteration can be used to eliminate calls to the procedure on all subsequent iterations. In my formulation, there is no way to distinguish values computed on the first iteration from values computed on arbitrary iterations: different points in a computation are distinguished by the different edges, but an edge in the body of a loop[3] does not distinguish one iteration from another.

One work-around is to hoist a copy of the entire loop body, i.e., to unroll the first iteration. This distinguishes the first iteration from all subsequent ones, so common subexpression elimination can reuse results from the first iteration to eliminates calls in subsequent ones. After the optimizations have been identified, the hoisted loop body can be un-hoisted prior to code generation.

This strategy is able to hoist even an expression whose value might change on each iteration. For example, for the code

```
while true do
    e: elem := set$choose(s)
    ⋮ % Code that doesn't delete e from s or assign e or s.
    end
```

the strategy would determine that the result of set$choose on the first iteration could be reused to replace the calls to set$choose on subsequent iterations—even when set$choose might return a different value on each iteration.

4.3 Dead Code Elimination

Dead code elimination is an optimization that removes parts of the program that are never executed. The principal benefit of dead code elimination is that it makes the code smaller, but it can have the secondary benefit of improving the performance of an instruction cache by reducing fragmentation.

Initially, I did not set out to perform dead code elimination, but in the course of building PSC (see Chapter 6), I discovered that dead code elimination came for free.

4.3.1 Proof Obligation

Recall that an edge is unreachable at run time if its theory is inconsistent. Thus, to eliminate an edge from the FG, the compiler must prove that $\mathcal{T}_e$ is inconsistent, i.e.,

$$true = false \in \mathcal{T}_e$$

[3] Edge e is in the body of a loop if e dominates the back edge of the loop and if the entry edge of the body dominates e.

If a theory $\mathcal{T}_i$ is inconsistent, the extension proof rule will propagate the inconsistency to the theory of each edge j dominated by i. This is sound because if i is unreachable and j could only be reached by going through i, j is unreachable.

Once all edges entering a node are eliminated, the node can be eliminated since it is unreachable via ordinary graph traversal starting from the entry edge.

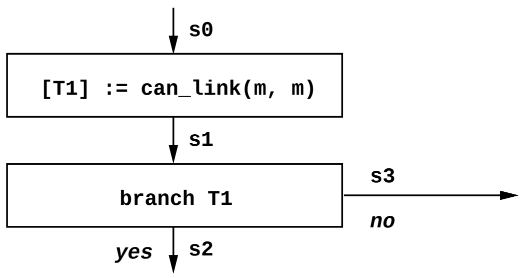
4.3.2 Example: can_link

In the code

```
if can_link(m, m)
   then ...
   end
```

the `then` arm of the `if` statement is dead code. (Such code might arise through the use of macros.) The interface of `can_link` (Fig. 4.3, p. 53) specifies that the return value is equal to the result obtained by applying the $\_\_\star\_\_$ relation, which denotes whether two monomers can bond together. Because $\_\_\star\_\_$ is irreflexive (see the Monomer trait, Fig. 4.4, p. 54), `can_link(m, m)` is always false.

The flow graph for the code above is



The proof obligation is

$$true = false \in \mathcal{T}_2$$

The `ensures` clause of `can_link` asserts

$$\sigma_1^{\mathrm{Env}}(\text{`T1'}) = (\sigma_0^{\mathrm{Env}}(\text{`m'}) \star \sigma_0^{\mathrm{Env}}(\text{`m'})) \in \mathcal{T}_1$$

The extension proof rule propagates this assertion as

$$\sigma_1^{\mathrm{Env}}(\text{`T1'}) = (\sigma_0^{\mathrm{Env}}(\text{`m'}) \star \sigma_0^{\mathrm{Env}}(\text{`m'})) \in \mathcal{T}_2$$

Because the rule for branch nodes asserts $\sigma_1^{\mathrm{Env}}(\text{`}\mathtt{T1}\text{'}) = \mathit{true} \in \mathcal{T}_2$, this simplifies to

$$\mathit{true} = (\sigma_0^{\mathrm{Env}}(\text{`}\mathtt{m}\text{'}) \star \sigma_0^{\mathrm{Env}}(\text{`}\mathtt{m}\text{'})) \in \mathcal{T}_2$$

Finally, using the Monomer trait axiom $\forall\ m\ [m\star m == \mathit{false}]$, this simplifies to

$$\mathit{true} = \mathit{false} \in \mathcal{T}_2$$

4.4 Improving Side Effect Analysis

Side effect analysis plays a key role in many optimizations. To eliminate common subexpressions, a compiler must prove that the code between the common expressions does not alter their value. To hoist an expression from a loop, a compiler must prove that the body does not alter the expression's value. Specifications enhance side effect analysis in several ways.

4.4.1 Benevolent Side Effects

Specifications can conceal what Hoare calls *benevolent* side effects from clients. These side effects are benevolent because they are invisible to clients but they improve performance.

For example, Fig. 4.7 is an implementation of `polymer$get_left` that performs a benevolent side effect. Here, polymers are represented using trees whose leaves are non-empty arrays of monomers.[4] The implementation of `get_left` caches its result in the `left` field of the representation, but this side effect is invisible to clients of `polymer`.

Specifications also conceal temporary side effects. An implementation may modify some data, perform some computation, and then restore the data to its original state. From the client's perspective, the data is unchanged.

4.4.2 Immutable Types

Immutable types can obviate the need to perform side effect analysis. For example, consider the code

```
b1 := monomer$can_link(m1, m2)
⋮
b2 := monomer$can_link(m1, m2)
```

[4]This representation allows efficient implementations for the operations to reverse a polymer or to link two linear polymers together.

```
polymer = mutable type

  rep = record[root:   tree,   % Tree of arrays of monomers
               cyclic: bool,   % Is the polymer cyclic?
               left,           % Leftmost monomer, if known
               right:  mono_q] % Rightmost monomer, if known
  tree = oneof[leaf:   leaf,
               pair:   pair]
  leaf = record[reverse: bool, % Interpret data as if reversed
                data:    am]   % R.I.: Non-empty
  pair = record[left,
                right:   tree]
  am = array[monomer]
  mono_q = oneof[known: monomer,
                 none:  null ]

  get_left = proc (p: rep) returns (monomer)
      except signals cyclic
    if p.cyclic then signal cyclic end
    return(mono_q$value_known(p.left))
      except when wrong_tag: end
    root: tree := p.root
    while true do
      tagcase root
         tag leaf (l: leaf):
             ans: monomer
             if l.reverse
                then ans := am$top(l.data)
                else ans := am$bottom(l.data)
                end
             p.left := mono_q$make_known(ans)
             return(ans)
         tag pair (pr: pair):
             root := pr.left
         end
      end
    end get_left
```

Figure 4.7: A Benevolent Side Effect in `get_left`

The interface of can_link specifies that its return value only depends on the monomers passed in as arguments (see Fig. 4.3, p. 53): In particular, the result of can_link does not depend on the store of the program state since neither ^ nor ' appear in the interface. Thus, side effects to the store are irrelevant. Instead, it is sufficient to check that the identifiers m1 and m2 are not assigned, which is easy given Speckle's scoping rules and lack of call-by-reference.

The reason that can_link ignores the store is that monomers are immutable and do not contain locations. If monomers were mutable, can_link would be specified using m1^ and m2^.

4.4.3 Abstract Data Types

Data abstraction, which is embodied in data type specifications, simplifies reasoning about side effects because it reduces the possibilities for aliasing. Each data type must ensure that outside its implementation, locations of the type are never aliased to locations of other types. This makes it easier to perform optimizations.

For example, consider the code

```
a: array[monomer] := ...
p: polymer := ...
a[1] := p.left
m := p.left
```

If this code is outside the implementation of polymer, it is easy to prove that it is safe to eliminate the second call to get_left: the result depends only on the value stored in polymerLoc p, and the only location modified by the code is that of a, which is not a polymerLoc. Thus, it is safe to eliminate the second occurrence of p.left, even if polymers are represented using arrays of monomers, e.g., as in Fig. 4.7.

4.4.4 Assertions about Allocation

To prove that two locations of the same type are distinct, the compiler can rely on allocation assertions made with New. For example, in the code

```
p: polymer := ...
if monomer$can_link(p.left, p.right) then
   p2: polymer := copy(p)
   polymer$change_cycle(p2)
   return(p2, p.left, p.right)
   end except when not_linear: end
```

```
copy = proc (p: polymer) returns (p2: polymer)
  requires --
  modifies --
  ensures  p2' = p1^ ∧ New(p2)

change_cycle = proc (p: polymer)
  requires --
  modifies p
  ensures  p' = lin2cyc(p^)
  except signals not_linear when ¬is_linear(p^)
```

Figure 4.8: Interfaces for `copy` and `change_cycle`

the occurrences of `p.left` and `p.right` in the `return` statement can be eliminated—even though the polymer `p2` is modified between the common subexpressions.

The key interfaces appear in Fig. 4.8. The interface for `copy` specifies that `p` is not modified. Furthermore, the assertion `New(p2)` implies that `p2` is not an alias for `p`. Since `change_cycle` modifies only `p2`, `p` is not modified, so the values of `p.left` and `p.right` are unchanged.

4.5 Related Work

4.5.1 Conventional Techniques

Conventional optimization techniques, e.g. [9, 12, 42, 46, 50], all use some form of symbolic evaluation based on the semantics of the programming language. For example, value numbering [9], one of the earliest techniques, can eliminate primitive expressions such as sums and products. It relies on the semantics of addition (e.g., that `+` is commutative) to recognize when lexically distinct expressions like `a+b` and `b+a` are equal, and it relies on the semantics of assignment to determine how an assignment statement can alter the values of identifiers that appear in expressions.

The work described here is an extension of the conventional techniques. The key difference is to use symbolic evaluation based on the semantics of the programming language *and* the semantics of specifications, i.e., to combine the two. The specifications allow several generalizations:

1. Expressions are generalized to include procedure calls.

   I.e., procedure calls can be eliminated or hoisted from loops.

2. "Common" expressions need not be equal.

   A procedure call can be eliminated whenever an available value satisfies the procedure's specification. For procedures with non-deterministic specifications, the available value may be different from the value the procedure would return.

3. Hoisted expressions need not be constant.

   A procedure call can be hoisted out of a loop if the result from the first iteration satisfies the procedure's specification for all iterations.

4.5.2 Alias Analysis

Alias analysis is a crucial component of any optimizer because aliasing increases the likelihood that a side effect will foil an optimization. Much work on alias analysis has been restricted to languages that do not allow pointers [3, 10] or that restrict pointers to at most one level of indirection [43]. Some exceptions are [7, 29, 33], which use interprocedural analysis.

The alias analysis techniques in [7, 29, 33] annotate each edge in a FG with a summary graph approximating the data structures at that point in the program. Because the size of the summary graphs must be bounded at compile-time even though the program's data structures are unbounded at run-time, a single node in the summary graph must sometimes be used to represent distinct data structures. This approximation can foil some optimizations.

A similar problem arises for Speckle. Each theory $\mathcal{T}_e$ describes the state of the data structures at edge e. $\mathcal{T}_e$ is a full description of the program state, not an approximation. However, the theorem-prover needed to detect optimizations will sometimes fail to discharge the proof obligation for a legal optimization. Thus, legal optimizations may still be missed.

4.5.3 Pragmas

Previous work on program optimization has used pragmas to enhance common subexpression elimination and code motion. Programmers write pragmas—hints for the compiler—to identify procedures that might be eliminated or hoisted out of loops. With the Gnu C Compiler, users can declare that a procedure is "`const`" [49]. The pragma "`const`" asserts that a procedure is referentially transparent, i.e., that given equal actual values, the procedure returns the same results and has no side effects. For example, `sine` and `cosine` are typically "`const`" procedures. The PL/I pragma "`reducible`" is similar to "`const`."

When the Gnu C Compiler detects that a "`const`" procedure is called more than once with the same arguments, the compiler eliminates the second call. The compiler can also hoist calls to "`const`" procedures out of loops when the actuals are loop-invariant.

Pragmas like "`const`" have two limitations. First, the pragmas do not describe the relation between a procedure's arguments and results—the pragmas only state that a relation exists. Thus, the pragmas can only eliminate common subexpressions when the *same* procedure is called more than once with the *same* actual values. Speckle specifications, on the other hand, do describe the relation between the actuals and the results (including the pre- and post-states), and this information can be used to detect a larger class of common subexpressions, e.g., the `polymer$bond_right` / `polymer$get_right` example in Section 4.1.3 on p. 57.

The second limitation is that pragmas like "`const`" only work for referentially transparent procedures. This precludes procedures that dereference pointers to compute their results—a side effect can change the target of a pointer without changing the pointer itself, so dereferencing a pointer is not referentially transparent. Thus, pragmas like "`const`" cannot eliminate calls to procedures like `polymer$get_left`, which dereferences a polymerLoc.

4.5.4 FX

The FX language incorporates specifications of side effects to enhance optimizations like common subexpression elimination and code motion [38]. FX divides the program store into disjoint regions somewhat like collections in Euclid [31]. This is similar to the way Speckle divides Loc, the domain of the program store, into disjoint subdomains—one per data type. In FX, however, there is no association between types and regions. A region may contain values of multiple types.

For each region accessible to a procedure, the procedure's specification states whether the procedure may read, modify, or allocate locations in the region. This information allows an FX compiler to eliminate successive calls to a procedure `p` by checking several conditions:

1. that each call has the same actual values,
2. that the regions read by `p` are not modified by the code between the successive calls, and
3. that `p` does not visibly modify or allocation locations.

Unlike the pragmas of the previous section, FX specifications can be used to eliminate calls to procedures that are not referentially transparent. The key difference is that FX specifications can refer to any region accessed by a procedure, not just the regions containing the actual values. However, FX shares the other limitation of pragmas: FX does not describe the relation between the actuals, results, and pre- and post-states of a procedure. This means that FX can be used to eliminate only successive calls to the same procedure. FX is insufficient to handle cases like the `bond_right` / `get_right` example in Section 4.1.3 on p. 57.

The FX compiler checks that an implementation satisfies its FX specification. However, the compiler does not use specifications to conceal benevolent side effects or temporary side effects.

4.6 Summary

Because specifications are simpler than code, they enable optimizations that are difficult to perform by analyzing only code. Specifications allow the compiler to eliminate calls to procedures that dereference pointers, and they make it easier to exploit relationships between procedures, such as the fact that a call to `get_right` returns the value passed in to `bond_right`.

Because specifications contain information not found in code, they permit optimizations that are impossible by analyzing only code: specifications make it possible to eliminate procedure calls that perform benevolent side effects, and specifications make it possible to substitute different values for the results of procedures with non-deterministic specifications.

Specifications also provide supplementary information that bounds the side effects of a procedure call, as well as information about control dependencies that can be used to detect unreachable nodes in the flow graph.

Chapter 5

Specialized Procedure Implementations

In this chapter, I present specialized procedure implementations (SPIs), the only kind of programmer-defined optimization supported in Speckle. SPIs are designed to reduce the conflict between generality and efficiency. The basic idea is to provide multiple implementations for a single procedure interface.

First, I explain how generality conflicts with efficiency when designing interfaces. Next, I describe how to alleviate these conflicts with SPIs. Then, I describe the obligation for proving that an SPI can be used in place of the general implementation. Finally, I discuss related work.

5.1 Motivation

When designing interfaces, a software engineer often faces the dilemma of whether to make an interface general or to specialize the interface to the task at hand [32]. Consider the interface for the data type `table` in Fig. 5.1. (This example is taken from the AC-Unify case-study discussed in Chapter 8.) A table is a mapping from keys to values. The procedure `store1` takes a table, a key, and a value and adds (or replaces) the binding for the key. The procedure `lookup` returns the value bound to a key in a table or signals `missing` if the key is unbound.

The interface of `store1` is more general than that of `store2` (Fig. 5.2) which requires that the key is not already defined in the table. However, the implementation of `store2` can be more efficient than that of `store1`. For example, suppose a table is represented as an unsorted list of key/value pairs with the invariant that no key appears twice. To preserve the invariant, `store1` will have to search the list to see if a binding exists for `k`, but this search is unnecessary for `store2`.

In some contexts, the generality of `store1` is unnecessary. Consider the

```
table = mutable type

  based on Table

  store1 = proc (t: table, k: key, v: value)
    requires --
    modifies t
    ensures  t' = bind(t^, k, v)

  lookup = proc (t: table, k: key) returns (v: value)
    requires --
    modifies --
    ensures v = image(t^, k)
    except
      signals misssing when ¬defined(t^, k)
```

Figure 5.1: A Table Interface

```
  store2 = proc (t: table, k: key, v: value)
    requires ¬defined(t^, k)
    modifies t
    ensures  t' = bind(t^, k, v)
```

Figure 5.2: An Alternative Interface for `table$store`

```
v: value := table$lookup(t, k)
  except when missing:
             v := value$create()
             table$store(t, k, v)
         end
```

Figure 5.3: Calling `table$store`

```
substitution = mutable type

  based on Substitution

  extend1 = proc (s: substitution, v: variable, t: term)
    requires --
    modifies s
    ensures  s' = bind(s^, v, t)
    except signals cyclic_definition
           when v ∈ vars(apply(unbind(s, v), t))
           ensuring s' = s^

  extend2 = proc (s: substitution, v: variable, t: term)
    requires v ∉ vars(apply(unbind(s, v), t))
    modifies s
    ensures  s' = bind(s^, v, t)
```

Figure 5.4: Two Interfaces for `substitution$extend`

code fragment in Fig. 5.3, which is taken from the source code of LP [16], a theorem-prover. Because the call to `lookup` signals `missing` only when `k` is not defined in `t`, and because the call to `value$create()` modifies nothing, `k` cannot be defined in `t` when `store` is called.

Fig. 5.4 contains another example of the conflict between generality and efficiency. The figure contains two possible interfaces for `substitution$extend`, which adds or replaces a binding for a variable in a substitution. Here, a `substitution` is a data type that maps logical variables to logical terms, and a desired invariant is that no values of type `substitution` contain cyclic definitions, e.g., that no substitution maps `x` to `y` and `y` to `x`. The interface of `extend1`, which preserves the invariant, is more robust than that of `extend2`, which relies on the client to preserve the invariant. However, because `extend1` must always check whether the invariant is preserved, it will be substantially slower than `extend2` in contexts where the invariant is guaranteed to be true.

With conventional programming languages, there are three solutions to the `table$store` dilemma. One is to sacrifice efficiency and choose only the general interface. Another is to sacrifice generality and choose only the efficient interface. A third is to provide both interfaces.

Providing both interfaces has several problems. First, it creates more work for clients, who will have to decide which interface to call. If a client makes the wrong choice, either performance or correctness will be compromised. Second, as the program evolves, the distinction between the two interfaces may become irrelevant. In the case of `table$store`, the representation may be changed to a sorted list, in which case `store2` is no faster than `store1`. Finally, a minor problem is that in a language like CLU that allows one store-like operation per type to be written as `t[k] := v`, only one of the interfaces can use this syntax.

Instead of providing both interfaces, it is better to provide one interface with multiple implementations and to have the compiler choose the appropriate implementation for the caller. In Speckle, this is accomplished using SPIs.

5.2 Syntax and Semantics

From the client's perspective, calling a procedure with an SPI is like calling any other procedure. From the implementor's perspective, SPIs allow a single procedure interface to have multiple implementations. One implementation—the general implementation—can be used anywhere. The other implementations—the SPIs—are usually more efficient than the general one but can be used only when certain conditions are met. The programmer defines these conditions formally using the specification language. The compiler substitutes an SPI for the general implementation when it can prove, using specifications, that the conditions are met at a particular call site. The general implementation must exist because a caller may rely on the full generality of the interface and because the compiler may fail to prove that an SPI suffices.

Fig. 5.5 is an implementation of `table$store` that uses an SPI. The general implementation calls the procedure `find_pair` to search for a pair whose key is `k`. If no such pair is found, a new pair is added to the list. Otherwise, the `val` field of the pair is updated.

The `special when` construct delimits the beginning of an SPI. Here, the additional precondition is `¬defined(t^, k)`, which the compiler must discharge at call sites of `store` in order to use the specialized code instead of the general code.

In general, there can be many SPIs for a procedure, as is the case in Fig. 5.6. The procedure `array$fetch`, which fetches an element from an array that can grow or shrink dynamically, uses SPIs to avoid bounds checks.

```
table = mutable type

  rep = list[pair]
  pair = record[key: key, val: value]

  store = proc(t: rep, k: key, v: value)
    %
    % General implementation
    %
    find_pair(t, k).val := v
      except when not_found:
                  rep$addh(t, pair${key: k, val: v})
             end
    %
    % Specialized implemenetation
    %
    special when ¬defined(t^, k)

      rep$addh(t, pair${key: k, val: v})

    end store
```

Figure 5.5: An SPI for `table$store`

Here, each additional precondition is named, and the name is used by a macro facility. The code is equivalent to the lengthier version in Fig. 5.7.

5.2.1 Compilation Issues

To optimize a caller of a procedure with SPIs, the compiler must know the additional preconditions, or *guards*, of the SPIs, e.g., the compiler must know that `table$store` has one SPI whose guard is `¬defined(t^, k)`. This means that full separate compilation is no longer possible: the compiler must know secrets about the implementation of a called procedure. However, the secret information is very stable—changes to the guards are likely to be infrequent.

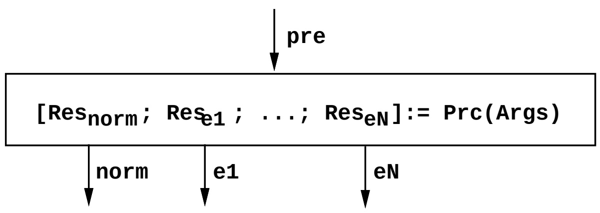
The simplest way to compile a procedure with SPIs is to output an ordinary procedure for each SPI, e.g., `store1` and `store2` for Fig. 5.5. This strategy probably leads to the most efficient code.

```
array = mutable type

  based on Array

  array_rep = record[
     low:    int,                    % low bound
     size:   int,                    % size
     elems:  bounded_array[elem]]    % elements


  array$fetch = proc (a: array, i: int) returns (e: elem)
     special when LOW_OK:  low(a^) ≤ i
             when HIGH_OK: i ≤ high(a^)

     ar: array_rep := down(a)
     ind: int := i - ar.low
#if ¬LOW_OK
     if ind < 0 then signal bounds end
#endif
#if ¬HIGH_OK
     if ind > ar.size then signal bounds end
#endif
     return(ar.elems[ind])
```

Figure 5.6: SPIs in `array$fetch`, Version 1

5.3 Proof Obligation

The proof obligation for substituting an SPI for the general implementation is the guard, *SW_Guard*, supplied by the user in the `special when` clause. This guard must be instantiated with the actual values passed to the procedure. For the procedure call:



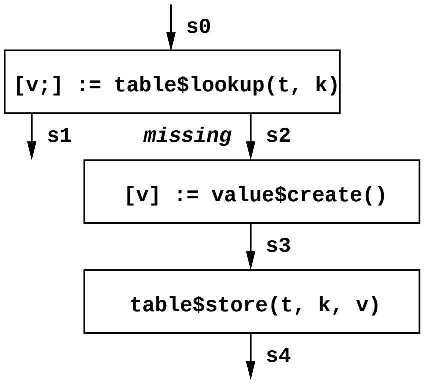
the proof obligation is $SW\_Guard(\sigma^{\mathrm{Str}}_{\mathrm{pre}}, Args) \in \mathcal{T}_{\mathrm{pre}}$

```
array = mutable type

  based on Array

  array_rep = record[
     low:    int,                      % low bound
     size:   int,                      % size
     elems:  bounded_array[elem]]      % elements


  array$fetch = proc (a: array, i: int) returns (e: elem)

     ar: array_rep := down(a)
     ind: int := i - ar.low
     if ind < 0 cor ind > ar.size then signal bounds end
     return(ar.elems[ind])

     special when low(a^) ≤ i ∧ i ≤ high(a^)

       ar: array_rep := down(a)
       ind: int := i - ar.low
       return(ar.elems[ind])

     special when low(a^) ≤ i

       ar: array_rep := down(a)
       ind: int := i - ar.low
       if ind > ar.size then signal bounds end
       return(ar.elems[ind])

     special when i ≤ high(a^)

       ar: array_rep := down(a)
       ind: int := i - ar.low
       if ind < 0 then signal bounds end
       return(ar.elems[ind])
```

Figure 5.7: SPIs in `array$fetch`, Version 2

5.3.1 Example: Calling table$store

The flow graph for the code in Fig. 5.3 is:



To call the SPI of store in Fig. 5.5, the proof obligation is:

$$\neg \mathit{defined}(\sigma_3^{\mathrm{Str}}(\sigma_3^{\mathrm{Env}}(\text{`}\mathtt{t}\text{'})),\ \sigma_3^{\mathrm{Env}}(\text{`}\mathtt{k}\text{'})) \in \mathcal{T}_3$$

By simple analysis, this goal follows from the exception guard of `lookup`, which asserts:

$$\neg \mathit{defined}(\sigma_0^{\mathrm{Str}}(\sigma_0^{\mathrm{Env}}(\text{`}\mathtt{t}\text{'})),\ \sigma_0^{\mathrm{Env}}(\text{`}\mathtt{k}\text{'})) \in \mathcal{T}_2$$

The analysis requires using the `modifies` assertions of `lookup` and `value$create` and the extension proof rule from p. 38.

5.4 Propagating Proof Obligations

One problem with SPIs is that the immediate caller of a procedure with SPIs may not contain enough contextual information to discharge the guard condition of an SPI. In these situations, the compiler must propagate the guard condition to the caller's caller.

Fig. 5.8, which contains part of an implementation of a data type for directed graphs, illustrates the problem. The LSL trait for graphs appears in Fig. 5.9. In the implementation, graphs are represented as tables, whose interface appeared in Fig. 5.1, with the types key and value renamed to node and node_set. The table representing a graph maps each node to a node_set containing the node's successors.

```
node = immutable type
  ⋮

node_set = immutable type
  based on NodeSet

  create = proc () returns (ns: node_set)
    requires --
    modifies --
    ensures  ns' = {} ∧ New(ns)
  ⋮

graph = mutable type
  based on Graph

  rep = table     % with key = node, value = node_set

  insert_node = proc (g: rep, n: node)
    requires --
    modifies g
    ensures  g' = add_node(delete_edges(g^, n), n)

    %%%%%%%%% Code %%%%%%%%%

    succs: node_set := node_set$create()
    table$store(g, n, succs)
    end insert_node
  ⋮
```

Figure 5.8: Part of a Graph Data Type

In Fig. 5.8, `insert_node` calls the procedure `table$store` (see Fig. 5.5), but there is not enough contextual information to prove the guard of `table$store`'s SPI, i.e., to prove `¬defined(g, n)`. Nevertheless, the guard is satisfied when `insert_node` is called from a context where `¬(n ∈ g.nodes)`. Fig. 5.10 contains such an example: at the call site `insert_node(g2, n)`, the condition `¬(n ∈ g2.nodes)` holds because the interface of `graph$nodes` ensures that it yields no duplicate nodes (interface not shown).

```
Graph: trait
  includes Set(Node, NodeSet), Set(Edge, EdgeSet)
  Graph tuple of nodes: NodeSet, edges: EdgeSet
  Edge tuple of source: Node, dest: Node
  introduces
    add_node: Graph, Node → Graph
    add_edge: Graph, Node, Node → Graph
    delete_edges: Graph, Node → Graph
    remove_edges: EdgeSet, Node → EdgeSet
    reverse: Graph → Graph
  asserts
    ∀ g: Graph, n,n1,n2: Node, es: EdgeSet
      add_node(g, n) == [insert(n, g.nodes), g.edges];
      add_edge(g, n1, n2) ==
        [g.nodes, insert([n1, n2], g.edges)];
      delete_edges(g, n) ==
        [g.nodes, remove_edges(g.edges, n)];
      remove_edges({}, n) == {};
      remove_edges(insert([n1, n2], es), n) ==
        if n = n1 then remove_edges(es, n)
                  else insert([n1, n2], remove_edges(es, n));
      reverse(g).nodes = g.nodes;
      [n1, n2] ∈ reverse(g).edges == [n2, n1] ∈ g.edges;
  implies
    converts add_node, add_edge, delete_edges, remove_edges,
             reverse
```

Figure 5.9: Graph Trait

One would like to allow the optimization, but the question is how. A possibility would be to write an SPI for `insert_node`, but this has two problems. First, the implementor of `insert_node` is a client of `table$store`, so he is not supposed to rely on the existence of an SPI in `table$store`. Second, Speckle doesn't provide a way for him to call the SPI directly, even if he wanted to.

Instead, the compiler should create the SPI for `insert_node`, because the compiler *does* have access to the SPI of `table$store`. The problem is that

```
reverse_graph = proc (g1: graph) returns (g2: graph)
  requires --
  modifies --
  ensures  g2' = reverse(g1^)  ∧  New(g2)

  g2: graph := graph$create()
  for n: node in graph$nodes(g1) do
      graph$insert_node(g2, n)
      end
  for n: node in graph$nodes(g1) do
      for succ: node in graph$successors(g1, n) do
          graph$insert_edge(g2, succ, n)
          end
      end
  end reverse_graph
```

Figure 5.10: Procedure `reverse_graph`

the compiler must be able to translate the guard condition `¬defined(g, n)` at the call site of `table$store` to the guard `¬(n ∈ g.nodes)` at the entry point of `insert_node`. Part of the problem is easy to solve: given the specification of `node_set$create`, it is straightforward to propagate `¬defined(g, n)` backwards to the entry point of `insert_node`. The difficult part is to translate this guard, which is in terms of tables, into the desired guard, which is in terms of graphs.

My solution is to have implementors of data types write stylized abstraction functions. Because the guards of SPIs will generally be expressed using observer functions, the definition of the abstraction function should translate observers of representation values into observers of abstract values.

Fig. 5.11 contains a trait defining the abstraction function for the graph example. It defines `defined` and `image`, which are observers of tables, in terms of `__.nodes` and `__.edges`, which are observers of graphs. For the example, only the first equation is needed to translate the condition `¬defined(g, n)` to `¬(n ∈ AF(g).nodes)`. To simplify this to `¬(n ∈ g.nodes)`, the compiler must use some mechanism to recognize that `AF(g)`, where `g` is a table, is equal to `g`, the graph in `insert_node`'s interface.

```
GraphAbsFunc: trait
  includes Graph, Table
  introduces
    AF: Table → Graph
  asserts
    ∀ t: Table, n,n1,n2: Node
      defined(t, n) == n ∈ AF(t).nodes;
      n2 ∈ image(t, n1) == [n1, n2] ∈ AF(t).edges;
```

Figure 5.11: Abstraction Function for Graph

5.5 Related Work

5.5.1 Transformation Rules

Speckle is not the first language that allows users to define optimizations. In [25], Hisgen presents an unimplemented design of a strategy based on transformation rules rather than specifications. The source language is a derivative of ADA.

To define an optimization, an implementor describes transformations to be performed by the compiler. For example, the implementor of `table` might provide a transformation rule to replace two calls to `lookup`, each with the same arguments, by a single call; this rule is analogous to common subexpression elimination for calls to `lookup`.

Transformations may have preconditions expressed using applications of side-effect free functions, which play a role analogous to that of LSL functions. Thus, the transformation language is sufficiently powerful to express any optimization defined by an SPI. In fact, the transformation language is more expressive than Speckle. For example, one can write a rule to replace the pattern

```
s3 := concat(s1, s2)
print(s3)
```

by

```
print(s1)
print(s2)
```

One problem of the transformation rule strategy is that it lacks modularity. To apply a transformation rule, the compiler must reorder the program so as to match the pattern of the rule. For the pattern above, the compiler may have to commute a call to `print` with other calls so that the calls to `concat` and `print` are consecutive. Likewise, the compiler must make common subexpressions consecutive before it can eliminate one of them.

The problem is that to commute one procedure call with another, the compiler must in general rely on "commutative" transformation rules supplied by the user. To maximize the compiler's ability to perform transformations, the user must consider all pairs of procedures. In contrast, Speckle uses `modifies` clauses—one per procedure—to determine whether a procedure call interferes with an optimization.

5.5.2 Eliminating Runtime Checks

A common use of SPIs is to eliminate runtime checks. Many have focussed on eliminating such checks for operations that are primitive to the source language, e.g., array bounds checking, nil checks in pointer dereferences, overflow, assignments from supertypes to subtypes, etc. SPIs are more general because they can be used to eliminate runtime checks that are not primitive to the source language.

In [47], Sites describes a technique for proving that programs written in a language like Algol 60 terminate without runtime errors. This requires proving properties sufficient to eliminate runtime checks in array references, numeric operations, assignments from supertypes to subtypes, etc. The language does not have pointers, so the problem of aliasing is simpler than in Speckle. Sites simulates his technique manually on several examples.

In [18], German develops a tool for verifying the absence of runtime errors, such as arithmetic overflow and invalid array indices. Users write formal specifications for procedures (entry and exit assertions) and decorate their code with sufficiently strong assertions so that the verifier can discharge all of the assertions plus the absence of runtime errors. German's work focuses on defining Pascal formally and expressing assertions sufficient to preclude a runtime error. He does not describe the strategies used to discharge assertions. (My strategy is presented in Chapter 6.)

In [41], McHugh examines all of the static checks of Gypsy, a derivative of Pascal. Gypsy is a programming environment for verified software, so programs typically contain entry, exit, and other assertions. McHugh's compiler generated optimization conjectures that, when discharged by the UT Interactive Prover [6], resulted in the elimination of code supporting exceptions—i.e., a broad category of runtime checks. McHugh does not describe strategies used to prove the conjectures.

In [20], Gupta reduces the overhead of array bounds checks by eliminating redundant checks that occur in code fragments such as "`a[i] := a[i]+1`" and by moving checks out of loops. The strategy used relies on the programming language semantics of arrays and does not extend to user-defined types.

Currently, Greg Nelson and David Detlefs are studying array bounds checking, nil checks, and other runtime checks in Modula-3 [44].

5.6 Summary

SPIs are a form of user-defined optimization that allow one procedure interface to have multiple implementations. Rather than compromising generality for efficiency, a programmer can use an SPI to have the compiler substitute a specialized implementation for the general one in calling contexts where the specialized implementation suffices.

Chapter 6

Prototype Speckle Compiler

There are two important issues to consider when designing a compiler that uses specifications. The first is how to translate specifications written in a declarative style into a logical system with an operational semantics. This translation must be done carefully so that the logical system can automatically discharge the proof obligations of many optimizations that are safe. The second issue is that of compiler performance. Most of the optimizations attempted by a compiler are unsafe, e.g., most procedure calls cannot be replaced by an assignment. Therefore, the compiler cannot afford to spend much time trying to prove any one conjecture.

To explore these issues and to test the ideas of the previous chapters, I constructed a prototype Speckle compiler, PSC. The inference engine of PSC is a stripped-down, automated version of the interactive theorem-prover LP (version 2.2a) [16, 22]. LP is particularly well-suited for Speckle 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 important because most conjectures PSC tries to prove are false.

In this chapter, I describe PSC, what it does, and how it works. In particular, I focus on how the formalization of Speckle programs is adapted to exploit LP's capabilities and on how LP's capabilities are extended to enhance reasoning about conditionals and loops.

6.1 What PSC Does

PSC implements most of the optimizations described in chapters 4 and 5. However, instead of generating code, PSC outputs a list of successful and unsuccessful attempts at performing optimizations. PSC does not attempt to propagate the guards of SPIs up the call graph and does not attempt to hoist expressions unless they are loop constants.

Before running PSC, the user must convert the LSL specifications used by a program into an LP logical system. This process could (and should) be entirely mechanized. Currently, the user must run the LSL Checker to translate LSL into LP's syntax. Then, the user must run LP to convert automatically the declarative specifications into a logical system with operational semantics.

Once the logical system is created from the LSL specifications, the user runs PSC, which outputs a list of attempted optimizations and indicates which succeeded and which failed. PSC takes the following actions:

1. Reads the logical system derived from LSL specifications.
2. Adds axioms for location sorts and `LocSet`s as specified in data type interfaces, and translates procedure and iterator interface specifications into predicates on program states. (See Chapter 2.)
3. Translates the program (a procedure body) into a flow graph.
4. Uses the proof rules from Chapter 3 to generate a logical system for each edge in the graph.
5. Uses the logical systems to try to discharge the proof obligations for performing optimizations given in chapters 4 and 5.

The second and third steps are straightforward, so I will not discuss them further. The fourth step—constructing a logical system for each edge—is an expensive one. However, these logical systems are constructed once per compilation and then used to try to discharge the proof obligations for each attempted optimization.

The remainder of this chapter begins with a description of LP, its logical systems, and its facilities for constructing logical systems from LSL specifications. Next, I describe how PSC constructs logical systems for each edge using LP. Then, I describe the strategy for performing automated proof by cases and induction in PSC. The strategy is a simple one that does not attempt to synthesize loop invariants. Finally, I explain how the proof strategy and the logical systems are used to detect optimizations.

6.2 LP Logical Systems

LP (version 2.2a) is an interactive theorem-prover for a fragment of multisorted, first-order, predicate logic. Like LSL, LP does not allow quantifiers in terms. Instead, all variables are assumed to be universally quantified.[1]

In general, LP's logical systems contain five types of axioms: rewrite rules, operator theories, deduction rules, induction rules, and equations. However, PSC uses only the three types of axioms that are used automatically: rewrite rules, operator theories, and deduction rules. Equations and induction rules are not used because they require interaction from the user.

6.2.1 Rewrite Rules

Rewrite rules implement a proof method based on normalization. The basic idea is to rewrite semantically equal terms to syntactically equal normal forms and to rewrite conjectures to `true`.

A *rewrite rule* $LHS \rightarrow RHS$ consists of a pattern LHS and a replacement RHS. A rule can be used to *reduce* a term T to a simpler term by matching LHS to a subterm of T, applying the resulting substitution to RHS, and replacing the matched subterm. A term is *irreducible* by a rewrite rule if none of its subterms match the rule's pattern. The verbs "simplify" and "rewrite" are synonyms for "reduce."

A set of rewrite rules can be combined to form a *rewriting system*. A term is reducible by a rewriting system if it is reducible by any of the system's rules. A *normal form* of a term is computed by repeatedly reducing a term until it is irreducible. A term may have more than one normal form. A rewriting system is *terminating* if no term is forever reducible by the system.

LP also supports *conditional rewrite rules*. A conditional rule is a rewrite rule prefixed by a guard that must be satisfied before the basic rule can be used to reduce a term. As before, the first step in reducing a term T by $G\colon LHS \rightarrow RHS$ is to obtain a substitution σ by matching LHS to T. The guard must then be discharged by reducing $\sigma(G)$ to `true`. Only then can the original term be reduced.

6.2.2 Operator Theories

Operator theories enhance term rewriting by generalizing matching [54]. Each function symbol has an operator theory. The three operator theories handled by LP are: **empty**, **commutative**, and **associative-commutative**.

[1] These restrictions on the use of quantifiers will be removed in the next release of LP.

A function's operator theory determines how LP performs matching. By default, a function `f` has the empty theory, so, for example, `f(a,b)` does not match `f(b,a)`. However, if `f` is commutative, `f(a,b)` does match `f(b,a)`. Furthermore, if `f` is associative-commutative, `f(a,f(b,c))` matches `f(f(a,b),c))`, etc.

6.2.3 Deduction Rules

A *deduction rule* has the form

$$\textit{when } [\forall x_1, \ldots, x_n <\!\textit{hypotheses}\!>] \textit{ yield } <\!\textit{conclusions}\!>$$

where $<\!\textit{hypotheses}\!>$ and $<\!\textit{conclusions}\!>$ are sequences of equations. Logically, a deduction rule is equivalent to an implication of the form

$$([\forall x_1, \ldots, x_n <\!\textit{hypotheses}\!>]) \Longrightarrow (<\!\textit{conclusions}\!>)$$

Because LP's terms cannot have nested quantifiers, deduction rules cannot be expressed as implications unless n is 0, i.e., the hypotheses can are expressed without nested quantifiers.

Operationally, a deduction rule adds the assertion $\sigma(<\!\textit{conclusions}\!>)$ when there is a substitution σ that matches $\sigma(<\!\textit{hypotheses}\!>)$ to axioms in the logical system.

Because PSC is an unusual user of LP, PSC can discard all of the deduction rules once the logical systems have been constructed, i.e., before using the logical systems to prove optimization obligations. PSC can discard the deduction rules because they are used only when facts are added to a logical system, and PSC, unlike typical LP users, discards a conjecture once it has been proved rather than adding the conjecture as an axiom.

6.2.4 Generating Logical Systems from LSL

LSL specifications can be translated into LP logical systems mechanically. The LSL checker translates LSL syntax into LP commands, from which LP constructs a logical system. The primary concern for PSC is whether the logical system can automatically discharge conjectures that arise during optimization. A logical system discharges a conjecture if the rewrite rules normalize the conjecture to `true`.

One potential problem is that most LSL axioms are equations, which PSC does not use. LP converts sets of equations into terminating sets of rewrite rules whenever possible. Occasionally, LP cannot convert an equation because the resulting rewriting system would not be terminating. PSC ignores such equations. In my experience, all equations were converted to rewrite rules.

A more serious problem is that process of translating LSL specifications is unable to generate some deduction rules that are needed for optimization. Currently, LSL's `partitioned by` clauses are the only axioms that are converted into deduction rules. However, there are many deduction rules that cannot be expressed with `partitioned by`.

One way around the problem is to extend LSL with a syntax for deduction rules. This is the approach I use in PSC. The disadvantage of this approach is that the specifier is forced to consider the operational semantics of specifications that are supposed to be declarative. A better solution would be to make LP infer the deduction rules from declarative specifications. Unfortunately, such a solution is not currently available.[2]

Example: Table Trait

Fig. 6.1 is a trait for tables. When this trait is translated by the LSL Checker and the output is given to LP, the result is the logical system in Fig. 6.2. LP converts the `partitioned by` clause into a deduction rule and converts all equations to rewrite rules. (The third assertion in the trait is a shorthand for the equation ¬`defined(empty, k) == true`.) The `generated by` assertion is ignored.

6.3 Constructing Logical Systems for a Program

PSC uses the formalization of programs described in chapters 2 and 3 to model programs. Recall that a program consists of a single procedure body, which calls other procedures that are optimized separately. The program is translated into a flow graph, and each edge has a theory that describes the program state at the edge.

For each edge e, PSC constructs a logical system, $\mathcal{R}_e$, to approximate the theory $\mathcal{T}_e$. The purpose of $\mathcal{R}_e$ is to discharge proof obligations of the form $F \in \mathcal{T}_e$ by seeing if $\mathcal{R}_e$ normalizes F to `true`.

$\mathcal{R}_e$ may fail to reduce some formulas in $\mathcal{T}_e$ to `true`, so the compiler may miss some optimizations. However, the compiler does not perform unsound

[2]Future versions of the LSL checker and LP are expected to alleviate this problem.

```
Table: trait
 introduces
   empty:                      → Table
   bind:    Table, Key, Value → Table
   image:   Table, Key        → Value
   defined: Table, Key        → Bool
 asserts
   Table partitioned by image, defined
   Table generated by empty, bind
   ∀ k,k1,k2: Key, v: Value, t: Table
     image(bind(t, k1, v), k2) ==
       if k1 = k2 then v else image(t, k2);
     ¬defined(empty, k);
     defined(bind(t, k1, v), k2) == k1 = k2 ∨ defined(t, k2);
```

Figure 6.1: Table Trait

```
   when [∀ k,k1: Key  image(t1, k) == image(t2, k),
                      defined(t1, k1) == defined(t2, k1)]
   yield t1 == t2

   ∀ k,k1,k2: Key, v: Value, t: Table
     image(bind(t, k1, v), k2) →
       if k1 = k2 then v else image(t, k2);
     defined(empty, k) → false;
     defined(bind(t, k1, v), k2) → k1 = k2 ∨ defined(t, k2);
```

Figure 6.2: Logical System Generated for Table Trait

optimizations, i.e., $\mathcal{R}_e$ is constructed using the proof rules from Chapter 3 so that it normalizes only formulas in $\mathcal{T}_e$ to `true`.

This section begins with a description of the term proof strategy used in PSC. Next, I revise the model of the program store slightly to make proofs involving `modifies` clauses succeed more often. Subsequently, I explain how PSC creates the logical system for each edge. Finally, I present a few minor implementation issues.

6.3.1 Proof Strategy

Typically, the proof obligations for various optimizations are terms that contain program state symbols. Since the proof method is rewriting terms to normal forms, it is useful to have a strategy for normalizing program state symbols.

PSC constructs logical systems to support the proof strategy of rewriting terms "towards the entering edge." This strategy is to rewrite a term, t, that contains the program state symbol of an internal graph edge, e, to either:

1. a term that contains program state symbols of only edges that dominate e, or
2. a term that doesn't contain any program state symbols.

   (This case can be viewed as a special case of (1).)

LP uses a simplification ordering to choose whether to convert an equation `a = b` into `a` $\rightarrow$ `b` or `b` $\rightarrow$ `a` [17]. To encourage LP to rewrite terms towards the entering edge, PSC imposes the *dominates* partial order onto LP's simplification ordering. If e_1 dominates e_2, PSC makes the term σ_1 simpler than σ_2 in the simplification ordering. The effect is that when an `ensures` clause of the form `x' = f(x^)` is instantiated and converted into a rewrite rule, the rule is usually ordered as `x'` $\rightarrow$ `f(x^)` rather than `f(x^)` $\rightarrow$ `x'`.

Likewise, the `modifies` clause must be converted into a rule that simplifies terms towards the entering edge. In Chapter 2, the semantics of *OnlyModifies*$(pre, post, S)$ was defined as

$$\forall\ l : Loc \in domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}})\ \ [\ l \notin S\ \implies\ \sigma^{\mathrm{Str}}_{\mathrm{post}}(l) = \sigma^{\mathrm{Str}}_{\mathrm{pre}}(l)\ ]$$

where S is the set of modified locations. Recall that $l \in domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}})$ is to distinguish locations that were allocated by a procedure call from locations that existed before the call.

One possibility is to order the definition of *OnlyModifies* into a conditional rewrite rule

$$(l \in domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}})\ \wedge\ l \notin S)\ :\ \sigma^{\mathrm{Str}}_{\mathrm{post}}(l) \rightarrow\ \sigma^{\mathrm{Str}}_{\mathrm{pre}}(l)$$

Unfortunately, using this rules leads to two problems. One is that even for the common case $S = \{\}$ (i.e., `modifies --`), the guard is non-trivial to discharge automatically. Discharging $l \in domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}})$ involves many facts, such as New assertions from specifications, axioms about reachability, and program state invariants about the absence of dangling references. The second problem is that later, when partial specifications are allowed, the New assertions are likely to be omitted. Therefore, I use a slightly different formalization of the program state that simplifies the guard of the conditional rewrite rule.

6.3.2 An Alternate Model for the Program Store

Recal that in Chapter 2, the store is defined as a finite mapping $Loc \rightarrow$ *LSLValue*, and the semantics of allocation is to bind an undefined location l to some value, v. In PSC, the store is defined as an infinite mapping where each possible LSLValue is the image of an unbounded number of locations. The semantics of allocating a location to have initial value v is to select a previously unreachable (i.e., inaccessible) location l that is bound to v and make l reachable to the caller. Because the store is infinite whereas a program can allocate only a finite number of locations, there is always an unreachable location l for every possible initial value v.

When the store is modeled as an infinite mapping, the definition of *OnlyModifies*$(pre, post, S)$ simplifies to

$$\forall\ l : l \notin S \implies \sigma^{\mathrm{Str}}_{\mathrm{post}}(l) = \sigma^{\mathrm{Str}}_{\mathrm{pre}}(l)$$

which can be ordered into the conditional rewrite rule[3]

$$l \notin S\ :\ \sigma^{\mathrm{Str}}_{\mathrm{post}}(l) \rightarrow\ \sigma^{\mathrm{Str}}_{\mathrm{pre}}(l)$$

The meaning of `New(x)` must also be revised for infinite program stores. In Chapter 2, the meaning of `New(x)` was defined as

$$x \notin domain(\sigma^{\mathrm{Str}}_{\mathrm{pre}})\ \wedge\ x \in domain(\sigma^{\mathrm{Str}}_{\mathrm{post}})$$

With infinite stores, the meaning of $\texttt{New}(ls_1, ls_2, \ldots, ls_n)$ is that each ls_i contains only previously unreachable locations:

$$\bigwedge_{i=1 \ldots n, var \in \sigma^{\mathrm{Env}}_{\mathrm{pre}}} ls_i\ \cap\ \texttt{reach\^{}}(var) = \{\}$$

and, just as before, that each pair of distinct `LocSets` in the assertion is disjoint. Recall that `reach^(x)` is the set of locations reachable from `x` in the pre-state (see p. 30).

[3]For minor reasons discussed later, PSC axiomatizes the `modifies` clause in a slightly different form.

6.3.3 The Entering Edge

Recall from Chapter 3 that the theory of the entering edge, $\mathcal{T}_{\text{enter}}$, is an extension of the theories of LSL specifications used by the program. These specifications are converted into a logical system, which PSC uses as a starting point for $\mathcal{R}_{\text{enter}}$.

Given this logical system as a starting point, PSC constructs $\mathcal{R}_{\text{enter}}$ by declaring and axiomatizing functions to look up the value of an identifier in the environment, to look up the value of a location in the store, to construct and examine `LocSets`, etc. The precise axiomatization is determined by the interface specifications of data types (see Chapter 2).

6.3.4 Edges Exiting Assignment, Branch, Procedure Call, and Iterator Call Nodes

For each assignment, branch, procedure call, and iterator call node in the graph, PSC constructs a logical system for each of the node's exiting edges by copying the logical system of the node's entering edge and asserting the conclusions of the proof rules in Chapter 3. If the conclusion of a proof rule is $F \in \mathcal{T}_{\text{e}}$, PSC adds the equation $F == \mathit{true}$ to $\mathcal{R}_{\text{e}}$ and then uses LP to convert the equation into a rewrite rule.

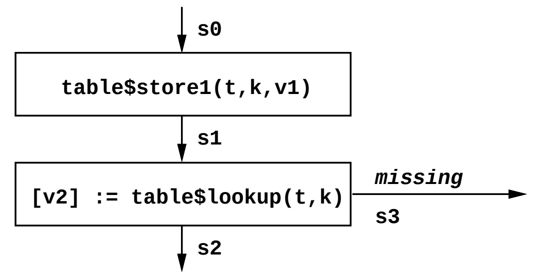
Example

As a simple example, consider the code fragment

```
table$store1(t, k, v1)
v2 := table$lookup(t, k)
```

which uses the table data type specified in Fig. 5.1 on p. 70. Fig. 6.3 contains the flow graph for this code and two tables. The first table lists, for each edge, the assertions for that edge's theory. The second table contains the rewrite rules generated by LP. The tables are condensed—the assertions and rewrite rules for edge e are not repeated for edges dominated by e, e.g., the entries for `s1` are not repeated for `s2`.

Note that the logical system for `s3` is inconsistent. This means that edge `s3` is unreachable. The call `lookup(t,k)` will never signal `missing` because `k` is defined in `t` by the call to `store1`.



Edge	Assertions
s0	See Fig. 6.1
s1	$\sigma_1^{\text{Str}}(\sigma_1^{\text{Env}}(\text{`}\mathtt{t}\text{'})) = \mathtt{bind}(\sigma_0^{\text{Str}}(\sigma_0^{\text{Env}}(\text{`}\mathtt{t}\text{'})),\ \sigma_0^{\text{Env}}(\text{`}\mathtt{k}\text{'}),\ \sigma_0^{\text{Env}}(\text{`}\mathtt{v1}\text{'}))$ $\forall\ l : [l \neq \sigma_0^{\text{Env}}(\text{`}\mathtt{t}\text{'}) \Rightarrow \sigma_1^{\text{Str}}(l) = \sigma_0^{\text{Str}}(l)]$ $\sigma_1^{\text{Env}} = \sigma_0^{\text{Env}}$
s2	$\sigma_2^{\text{Env}}(\text{`}\mathtt{v2}\text{'}) = \mathtt{image}(\sigma_1^{\text{Str}}(\sigma_1^{\text{Env}}(\text{`}\mathtt{t}\text{'})),\ \sigma_1^{\text{Env}}(\text{`}\mathtt{k}\text{'}))$ $\sigma_2^{\text{Str}} = \sigma_1^{\text{Str}}$ $\mathtt{defined}(\sigma_1^{\text{Str}}(\sigma_1^{\text{Env}}(\text{`}\mathtt{t}\text{'})),\ \sigma_1^{\text{Env}}(\text{`}\mathtt{k}\text{'}))$ $\mathrm{v} \neq \text{`}\mathtt{v2}\text{'} \Rightarrow \sigma_2^{\text{Env}}(v) = \sigma_1^{\text{Env}}(v)$
s3	$\sigma_3^{\text{Str}} = \sigma_1^{\text{Str}}$ $\neg\mathtt{defined}(\sigma_1^{\text{Str}}(\sigma_1^{\text{Env}}(\text{`}\mathtt{t}\text{'})),\ \sigma_1^{\text{Env}}(\text{`}\mathtt{k}\text{'}))$ $\sigma_3^{\text{Env}} = \sigma_1^{\text{Env}}$

Edge	Rewrite Rules
s0	See Fig. 6.2
s1	$\sigma_1^{\text{Str}}(\sigma_1^{\text{Env}}(\text{`}\mathtt{t}\text{'})) \rightarrow \mathtt{bind}(\sigma_0^{\text{Str}}(\sigma_0^{\text{Env}}(\text{`}\mathtt{t}\text{'})),\ \sigma_0^{\text{Env}}(\text{`}\mathtt{k}\text{'}),\ \sigma_0^{\text{Env}}(\text{`}\mathtt{v1}\text{'}))$ $l \neq \sigma_0^{\text{Env}}(\text{`}\mathtt{t}\text{'}) : \sigma_1^{\text{Str}}(l) \rightarrow \sigma_0^{\text{Str}}(l)$ $\sigma_1^{\text{Env}} \rightarrow \sigma_0^{\text{Env}}$
s2	$\sigma_2^{\text{Env}}(\text{`}\mathtt{v2}\text{'}) \rightarrow \sigma_0^{\text{Env}}(\text{`}\mathtt{v1}\text{'}))$ $\sigma_2^{\text{Str}} \rightarrow \sigma_1^{\text{Str}}$ (assertion reduces to an identity) $\mathrm{v} \neq \text{`}\mathtt{v2}\text{'} : \sigma_2^{\text{Env}}(v) \rightarrow \sigma_1^{\text{Env}}(v)$
s3	$\sigma_3^{\text{Str}} \rightarrow \sigma_1^{\text{Str}}$ (assertion reduces to an inconsistency) $\sigma_3^{\text{Env}} \rightarrow \sigma_1^{\text{Env}}$

Figure 6.3: Flow Graph with Assertions and Rewrite Rules

6.3.5 Edges Exiting Merge and Loop Nodes

Unlike the proof rules for other nodes, those for for merge and loop nodes have hypotheses that involve theories of edges in the program. Thus, merge and loop nodes must be handled differently.

Initially, PSC approximates the logical system of an edge e that exits a merge or loop node by copying the logical system of the edge, d, that immediately dominates e.[4] $\mathcal{R}_{\mathrm{d}}$ is a sound approximation for $\mathcal{T}_{\mathrm{e}}$ because, by the extension proof rule, $\mathcal{T}_{\mathrm{d}} \subseteq \mathcal{T}_{\mathrm{e}}$. Actually, $\mathcal{R}_{\mathrm{d}}$ is a weak initial approximation for $\mathcal{T}_{\mathrm{e}}$ because $\mathcal{R}_{\mathrm{d}}$ contains no information about σ_{e}. PSC then extends $\mathcal{R}_{\mathrm{e}}$ by adding an axiom $\sigma_{\mathrm{e}}^{\mathrm{Env}}(id) \rightarrow \sigma_{\mathrm{d}}^{\mathrm{Env}}(id)$ for each identifier id that is not assigned between d and e.

Despite the added axioms, $\mathcal{R}_{\mathrm{e}}$ contains very little information σ_{e}, so $\mathcal{R}_{\mathrm{e}}$ is too weak to prove many facts about σ_{e}. Instead, proofs involving σ_{e} are done by combining term rewriting with automated proof-by-cases and proof-by-induction. This is explained in Section 6.4, but first I wish to point out two minor ways in which the implementation of PSC, for historical reasons, differs from what I have described.

6.3.6 Minor Implementation Issues

I began implementing PSC before LP had conditional rewrite rules. To this day, PSC axiomatizes `modifies` clauses without conditional rules. To simulate the conditional rule

$$l \notin S \ : \ \sigma_{\mathrm{post}}^{\mathrm{Str}}(l) \rightarrow \ \sigma_{\mathrm{pre}}^{\mathrm{Str}}(l)$$

I use the rule

```
unreduced(post)[l] → if l ∈ S then post[l]
                               else unreduced(pre)[l]
```

where `sig[l]` is the translation of $\sigma_{\mathrm{sig}}^{\mathrm{Str}}(l)$ into LP syntax, and where `unreduced` is the identity function for program stores. Without `unreduced`, the rule above could not be ordered from left to right. Note that the axiom `unreduced(s) = s` must *not* be used; otherwise, LP would normalize away all occurrences of `unreduced` and order the rewrite rule in the opposite direction.

[4] As defined in [2], d immediately dominates e if $d \neq e$ and d dominates e and no other dominator of e is dominated by d. Intuitively, d is the nearest branch point that can cause control to bypass e.

In Chapter 2, I defined the environment as a mapping from identifiers to values. In PSC, the environment is axiomatized as a collection of functions, each of which maps a program state to the value of an identifier. $\mathit{OnlyAssigns}(\sigma^{\mathrm{Env}}_{\mathrm{pre}}, \sigma^{\mathrm{Env}}_{\mathrm{post}}, \text{`}\mathtt{x}\text{'})$ is axiomatized as

$$\bigwedge_{v \neq \text{`}\mathtt{x}'} v(\mathit{post}) = v(\mathit{pre})$$

6.4 Automating Proof-by-Cases and Proof-by-Induction

Using the logical systems constructed in Section 6.3, the strategy of rewriting terms toward the entering edge works well for programs without merge or loop nodes. However, if the program does contain such nodes, ordinary rewriting is often insufficient to simplify a term containing σ_{e}, where e is an edge exiting a merge or loop node, to a term that doesn't contain σ_{e}. Therefore, I combine term rewriting with a strategy that automates the proof-by-cases rule for merge nodes and the proof-by-induction rule for loop nodes. The key issue here is how to restrict the number of proof attempts to reduce compile time while still allowing the proof attempts needed for effective optimization.

6.4.1 Strategy

The automated proof strategy is a simple one. First, PSC uses term rewriting to reduce a proof obligation $F \in \mathcal{T}_{\mathrm{e}}$ to a normal form, $F \downarrow \mathcal{R}_{\mathrm{e}}$. Let E be the set of edges whose program state symbols appear in $F \downarrow \mathcal{R}_{\mathrm{e}}$. If E is empty, proof-by-cases and proof-by-induction are unnecessary.

Otherwise, the strategy is to select max, the edge dominated by all edges in E. Intuitively, max is the edge furthest from the entry edge. max is well-defined only when E is totally ordered by the *dominates* relation. This definition suffices because, in my formalization, a proof obligation $F \in \mathcal{T}_{\mathrm{e}}$ may contain program state symbols of only edges that dominate e (and thus are totally ordered). Likewise, the edges of program state symbols in $F \downarrow \mathcal{R}_{\mathrm{e}}$ all dominate e.

If max exits a merge node, PSC tries proof-by-cases on the merge node to simplify $F \downarrow \mathcal{R}_{\mathrm{e}}$ to a term that does not contain σ_{max}. If max exits a loop node, PSC tries proof-by-induction on the loop node to simplify $F \downarrow \mathcal{R}_{\mathrm{e}}$ to a term that does not contain σ_{max}. Otherwise, the strategy fails.

To better support the goal of simplifying terms towards the entering edge, the strategy is designed to simplify both Boolean and non-Boolean

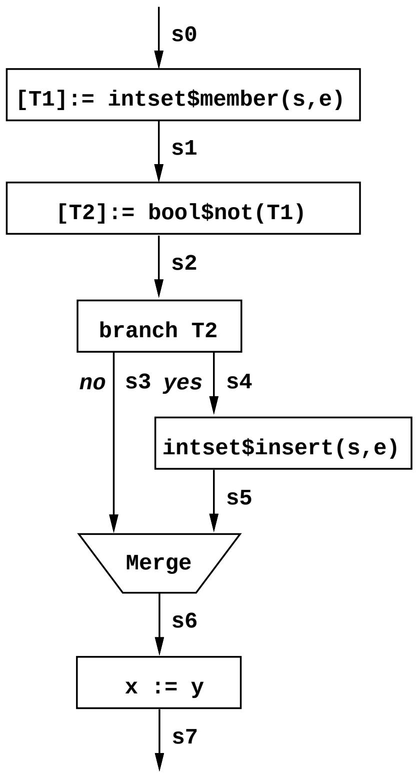


Figure 6.4: A Flow Graph with a Merge Node

terms. For any term t, a "proof-by-cases" is to show that the theory of each edge entering the merge node can be used to reduce t to the same identical term, which is the simplified form of t. Similarly, to simplify a term by induction, one must prove that it is loop-constant, i.e., that it is equal to a term that does not contain the program state symbols of edges in the loop.

6.4.2 Example: Case Proof

Fig. 6.4 contains the flow graph for the code

```
if ¬intset$member(s, e) then intset$insert(s, e) end
x := y
```

Suppose the goal is to simplify the term $\sigma_7^{\mathrm{Env}}(\text{`}\mathtt{e}\text{'}) \in \sigma_7^{\mathrm{Str}}(\sigma_7^{\mathrm{Env}}(\text{`}\mathtt{s}\text{'}))$ using $\mathcal{T}_7$. $\mathcal{R}_7$ will normalize this to $\sigma_6^{\mathrm{Env}}(\text{`}\mathtt{e}\text{'}) \in \sigma_6^{\mathrm{Str}}(\sigma_6^{\mathrm{Env}}(\text{`}\mathtt{s}\text{'}))$. Since σ_6 exits a merge

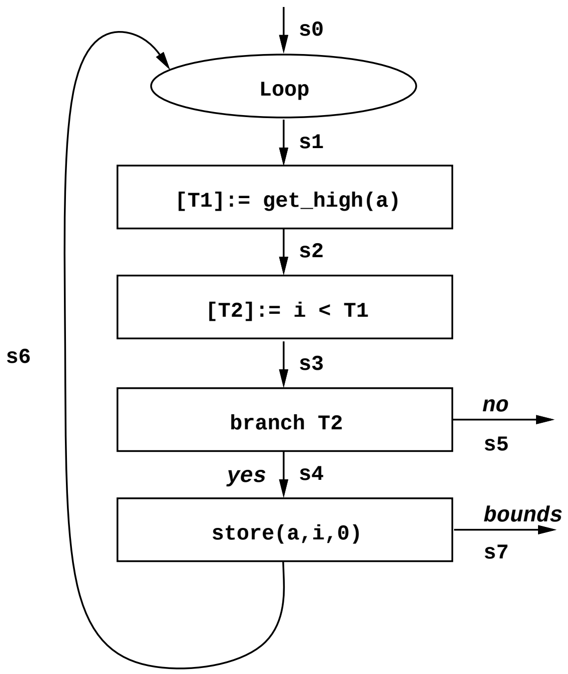


Figure 6.5: A Flow Graph with a Loop Node

node, the strategy is to try proof-by-cases:

- The first case is to simplify $\sigma_3^{\mathrm{Env}}(\text{‘}\mathtt{e}\text{’}) \in \sigma_3^{\mathrm{Str}}(\sigma_3^{\mathrm{Env}}(\text{‘}\mathtt{s}\text{’}))$ using $\mathcal{T}_3$. $\mathcal{R}_3$ normalizes this term to `true`.

- The second case is to simplify $\sigma_5^{\mathrm{Env}}(\text{‘}\mathtt{e}\text{’}) \in \sigma_5^{\mathrm{Str}}(\sigma_5^{\mathrm{Env}}(\text{‘}\mathtt{s}\text{’}))$ using $\mathcal{T}_5$. $\mathcal{R}_5$ normalizes this term to `true`.

The case proof succeeds and the original term simplifies to `true`.

6.4.3 Example: Induction Proof

Fig. 6.5 contains the flow graph for the code

```
while i < a.high do
   a[i] := 0
   end
```

Suppose the goal is to simplify the term $\sigma_2^{\mathrm{Env}}(\text{‘}\mathtt{T1}\text{’})$ using $\mathcal{T}_2$. $\mathcal{R}_2$ will normalize this to $high(\sigma_1^{\mathrm{Str}}(\sigma_1^{\mathrm{Env}}(\text{‘}\mathtt{a}\text{’})))$. Since σ_1 exits a loop node, the strategy is to try proof-by-induction to show that the term denotes a loop-constant.

- The base step is to simplify $\mathtt{high}(\sigma_0^{\mathrm{Str}}(\sigma_0^{\mathrm{Env}}(\text{‘}\mathtt{a}\text{’})))$ using $\mathcal{T}_0$. This term is irreducible.

- The inductive step is to prove

$$\mathtt{high}(\sigma_6^{\mathrm{Str}}(\sigma_6^{\mathrm{Env}}(\text{‘}\mathtt{a}\text{’}))) = \mathtt{high}(\sigma_1^{\mathrm{Str}}(\sigma_1^{\mathrm{Env}}(\text{‘}\mathtt{a}\text{’})))$$

  using $\mathcal{T}_6$. $\mathcal{R}_6$ normalizes this term to `true`. This relies on the axiom `high(store(a,i,e)) = high(a)`.

The induction proof succeeds, so $\sigma_2^{\mathrm{Env}}(\text{‘}\mathtt{T1}\text{’})$ simplifies to $\mathtt{high}(\sigma_0^{\mathrm{Str}}(\sigma_0^{\mathrm{Env}}(\text{‘}\mathtt{a}\text{’})))$.

Note that in this example, the inductive step involves proving an equality. If the original term were Boolean, the inductive step would be to prove an implication.

6.4.4 Recursion

Proof-by-cases and proof-by-induction each introduce subgoals. To discharge these subgoals, PSC recursively applies the strategy for combined term rewriting and automated proof-by-cases and proof-by-induction. A key issue is how to limit the number of proof attempts both to ensure termination and to improve compiler performance.

Consider the flow graph in Fig. 6.6. Suppose that term rewriting reduces a goal to a term containing σ_6. Since σ_6 exits a merge node, this triggers a proof-by-cases on the lower merge node. Next, suppose that $\mathcal{R}_4$ normalizes the subgoal for edge s4 to a term containing σ_3. In a naive implementation, this will trigger a recursive proof-by-cases on the upper merge node. Later, when the subgoal for s4 is discharged, the subgoal for s5 may repeat the proof-by-cases on the upper merge node!

PSC avoids this problem by restricting the merge and loop nodes that trigger recursive proof attempts. These nodes, called the active nodes, are described by a pair of edges, *top* and *bottom*. A merge or loop node n is active if *top* dominates n and n dominates *bottom*.

Initially, for a proof obligation $F \in \mathcal{T}_e$, *top* is the entering edge and *bottom* is e. There are three kinds of subgoals to consider:

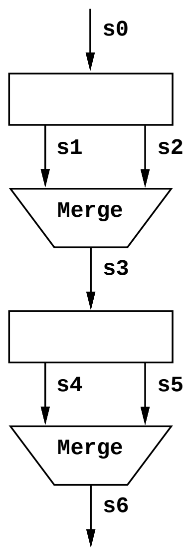


Figure 6.6: Restricting Proof-by-Cases

1. For a subgoal of a proof-by-cases, *top* is the immediate dominator of the merge node, and *bottom* is an edge entering the merge node.

   This restricts the active nodes to one "arm" of the branching code.

2. For the base case of a proof-by-induction, *top* is unchanged and *bottom* is the edge that immediately dominates the loop node.

   This restricts the active nodes to nodes above the loop.

3. For the inductive case of a proof-by-induction, *top* is the edge exiting the loop node, and *bottom* is the back edge.

   This restricts the active nodes to nodes in the loop body.

In the worst case, the number of case and inductive proof attempts is exponential in the nesting of loops. For example, in Fig. 6.7, the base case for the inner loop can trigger an induction over the outer loop, and the inductive step for the outer loop can trigger a second induction over the inner loop. (This process terminates because the second inner induction cannot trigger a second outer induction.) At worst, there may be 2^n proof attempts for a merge or loop node nested in n outer loops.

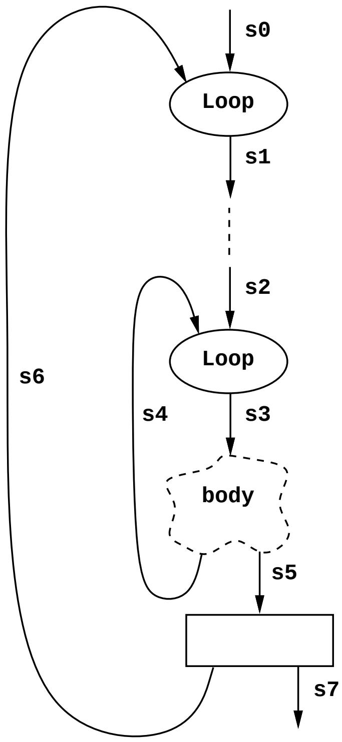


Figure 6.7: Nested Loops

Fortunately, loops are rarely nested deeply (more than three levels) in a single procedure [30]. However, if flow graphs were created by inlining procedures, the nesting depth would be greater, so more restrictions on recursive calls would be necessary.

6.5 Detecting Optimizations

Once the logical systems have been constructed for each edge, they can be used to detect the optimizations discussed in chapters 4 and 5. To discharge a proof obligation $F \in \mathcal{T}_e$, PSC uses term rewriting and automated proof-by-cases and proof-by-induction to try to simplify F to `true`. Only a few issues remain.

6.5.1 Common Subexpression Elimination

To eliminate a procedure call, PSC must first find available values to substitute for the results. Recall from Chapter 4 that a value is available at edge e if it is bound to an identifier at an edge d that dominates e. To eliminate a call at edge, PSC tries to discharge the proof obligations using each the available values of the proper type, one by one, until either a substitute is found or all available values have been tried.

There are a few points worth noting. First, the results of every procedure call are bound to identifiers because either the programmer stores the results himself or else the compiler introduces temporary identifiers. Thus, the values available at a call site include any value that was computed at nodes that dominate the call site. Second, Speckle encourages the use of many types. Because a substitute value must have the same type as the result, PSC needs to attempt proofs for only a fraction of the set of available values. Third, to avoid a combinatorial increase in proof attempts, PSC does not attempts to eliminate procedure calls that return multiple results

To reduce the compile time spent attempting common subexpression elimination, I use a simple trick. PSC tries to eliminate only calls to procedures whose specifications

- assert `modifies --`

  Note that this does not prevent the procedure from performing invisible side effects.

- do not assert `New` in the `ensures` clause

These constraints filter out many procedure calls that could never be safely eliminated.

6.5.2 Hoisting Expressions Out of Loops

PSC does not implement the full loop optimization described in Chapter 4. There, I described a strategy that could hoist an expression whose value might change on each iteration, e.g., a call to a procedure whose specification is non-deterministic.

PSC can hoist only expressions whose values are constant across all iterations of a loop. A result of a procedure call is constant across all iterations of a loop if the result simplifies to a term that either

- does not contain program state symbols, or

- contains only program state symbols of edges that dominate the loop node.

This condition is sufficient, but not necessary. Checking the condition is straightforward—PSC simplifies the result to a normal form, n, and checks whether n contains program states symbols that dominate the loop. For soundness, PSC also checks that the call always returns normally by proving the guard for the normal return.

6.5.3 Dead Code Elimination

Recall that an edge is unreachable if its theory is inconsistent. LP may, in the course of constructing a logical system, detect that the axioms are inconsistent. PSC relies on this feature to identify edges with inconsistent theories. Thus, dead code is detected for "free" in the process of performing the other optimizations.

6.5.4 Order of Optimizations

In conventional optimizing compilers, the order in which optimizations are performed can have a significant impact on the quality of code produced. For example, once dead code is eliminated from a program's FG, the compiler may detect more common subexpressions because there is less code between common expressions. To obtain such synergy, the compiler must reconstruct the FG and related data structures as optimizations are performed.

Because it is expensive to construct the logical systems for each FG edge, it is impractical for PSC to update the FG and its logical systems as optimizations are detected. Instead, PSC uses the same FG and logical systems to detect all optimizations. Such a strategy relies on the fact that none of PSC's optimizations conflict, i.e., performing one optimization never affects the safety of another optimization.

First, dead code is detected as the logical systems are being constructed. Next, PSC eliminates procedure calls. If a call can't be eliminated, PSC tries to discharge the guards for SPIs, if any. Finally, PSC checks to see if the call can be hoisted.

6.6 Summary

PSC relies primarily on conditional term rewriting, the code for which was taken from LP. The basic proof strategy is to simplify terms containing a program state symbol σ_{e} either to a term containing σ_{enter}, the program state symbol for the entering edge, or to a term containing no program state symbols. The proof rules for most kinds of nodes (assignment, branch, procedure call, and iterator call) are converted into rewrite rules that perform the simplification, while the proof rules for merge and loop nodes are implemented by a strategy for performing automated proof-by-cases and proof-by-induction.

Chapter 7

Supporting Partial Specifications

Current practice of software engineering in industry makes little use of formal specifications because the perceived cost of writing them outweighs their perceived benefits. Thus far, I have explained how specifications offer a new benefit—improving performance. In this chapter, I explain how to obtain this benefit without having to write full formal specifications of every procedure.

The basic idea is to let users write specifications of procedures incrementally. In the extreme case, a specification can be omitted entirely. In other cases, part of the specification is written and part of it is omitted. The part of the specification that is written is called a *partial specification*.

In this chapter, I describe how to write partial specifications in Speckle and how such specifications affect optimization. One of the key problems in supporting partial specifications is estimating the `modifies` clause when it is omitted. To this end, I extend data type specifications to make it possible to estimate which locations are reachable by a procedure when one has only the types of the formals. Finally, I report on the use of partial specifications in the case study of AC-Unify.

7.1 Writing Partial Specifications

Speckle allows each clause of a procedure or iterator specification to be written incrementally. Each `requires`, `modifies`, `ensures`, `when`, or `ensuring` clause may contain `?`, which indicates that the clause has not been fully written yet.

The symbol `?` can appear only at the end of a clause. In a `modifies` clause, a comma is used to delimit `?` from the comma-separated list of locations and `LocSets`. In all other clauses, $\wedge$ is used as the delimiter. Only one `?` per clause is allowed.

```
polymer$get_right = proc (p: polymer) returns (m: monomer)
  modifies --
  except signals not_linear

sort = proc (a: int_array) % deduces modifies a
  requires --
  modifies a
  ensures  ascending_order(a') ∧ ?
  except signals empty when ? ensuring ?

reverse_linked_list = proc (c: cons_cell)
                        returns (c2: cons_cell)
  requires ?
  modifies c, ?
  ensures  c'.cdr = nil ∧ ?
```

Figure 7.1: Examples of Partial Specifications

In a Boolean clause, the meaning of Q ∧ `?` is that if the specification were fully written, the clause would be a condition that implies Q. The meaning of `modifies x,?` is that a procedure may modify `x` and possibly other locations.

As a special case, a clause may be omitted. For the `modifies` clause, this is equivalent to a clause with a `?`. For a Boolean clause, this is equivalent to `true` ∧ `?`. If all of the clauses are omitted, the "specification" is nothing more than the procedure's header.

Fig. 7.1 contains several examples of partial specifications. The specification of `get_right` states only that the procedure performs no visible side effects and may signal the exception `not_linear`. The partial specification omits the precondition (if any), the postcondition, the guard for the exception, and the postcondition for the exception.

The specification of `sort` states that when it returns, `a` is in ascending order. However, the intended specification has a stronger postcondition—`a'` must be a permutation `a^`. The missing condition is denoted by ∧ `?` in the `ensures` clause. In this example, the specifier wrote `when ? ensuring ?` to alert the reader to the missing clauses, which could have been omitted without changing the meaning of the specification.

The specification of `reverse_linked_list` also uses `?` to draw attention to places where information is missing. Both the `requires` and `modifies` clauses could be omitted without changing the meaning of the specification.

There are two reasons for writing partial specifications in Larch/Speckle. One is that the specifier, for what ever reason, might not wish to write the full specification. For example, for the `sort` specification, the specifier might be unwilling to write an LSL predicate defining when one array is a permutation of another.

The other reason is that the interface may use data types that were supplied by others without the LSL functions needed to express the full specification. For example, the author of the `cons_cell` data type may not have defined an LSL function to denote whether a linked list is acyclic, i.e., whether `nil` is eventually reached by following the chain of `cdr`'s of a `cons_cell`. Without a way to express whether a list is acyclic, the specifier of `reverse_linked_list` cannot write its precondition.

7.2 Optimizing Programs with Partial Specifications

To optimize programs with partial specifications, I make a minor extension to the semantics of specifications given in chapters 2 and 3. Recall that the proof rules refer to the clauses of the specification of a procedure, `P`, using the auxiliary predicates *P.Pre, P.Post[status]*, and *P.Guard[status]*, where *status* is either *norm* or an exception name. For example,

$$P.Post[norm] = \text{``}\mathit{ensures\ clause}\text{''} \wedge \text{``}\mathit{modifies\ clause}\text{''}$$

To support partial specifications, these definitions are weakened by replacing "=" with "⇒" when an auxiliary predicate refers to a clause that is either omitted or uses `?`. For example, if `P`'s `modifies clause` is omitted, the definition above is replaced by the weaker assertion:

$$P.Post[norm] \Longrightarrow \text{``}\mathit{ensures\ clause}\text{''}$$

The translation of a Boolean clause $Q \wedge$ `?` is Q. The translation of an omitted clause is *true*. The translation of a `modifies` clause that uses `?` is *true*.

Fig. 7.2 contains a partial specification of the procedure `get_left` from the polymer data type, and Fig. 7.3 contains the assertions that constrain the auxiliary predicates for `get_left`. *Guard*[*norm*] is not defined because it depends on the exception guard *Guard*[`not_linear`], which is undefined. *Post*[`not_linear`] is partially defined, and *Post*[*norm*] is fully defined.

```
polymer = mutable type
  based on Polymer

  get_left = proc (p: polymer) returns (m: monomer)
    modifies --
    ensures  m = p^.left
    except signals not_linear when ?
```

Figure 7.2: A Partial Specification of `polymer$get_left`

$$\forall\ pre, post : Store, p : Polymer, m : Monomer$$

$$\begin{array}{lcl}
Pre(pre,p) & \Rightarrow & true \\
Post[norm](pre,post,p,m) & = & \left(\begin{array}{c} OnlyModifies(pre, post, \{\}) \\ \wedge \\ m = pre(p).\texttt{left} \end{array}\right) \\
Guard[norm](pre,p) & \Rightarrow & true \\
Post[\texttt{not_linear}](pre,post,p) & \Rightarrow & OnlyModifies(\mathrm{pre},\ \mathrm{post},\ \{\}) \\
Guard[\texttt{not_linear}](pre,p) & \Rightarrow & true
\end{array}$$

Figure 7.3: Auxiliary Predicates for `polymer$get_left`

7.2.1 Using Partial Specifications to Justify Optimizations

Using partial specifications is no different from using ordinary specifications to prove that an optimization is sound. The only difference is that a partial specification provides less information than might otherwise be available.

Fig. 7.4 contains revised specifications for the intset `least` and `choose` procedures from Chapter 4. The specification of `choose` is unchanged. The specification of `least` is partial because it does not specify that the return value must be the smallest member of the set.

These specifications are still sufficient to replace the call to `choose` by `i1` in the example

```
i1 := intset$least(s)
i2 := intset$choose(s)
```

There are two proof obligations. The first is to prove that `choose` will return normally; this proof is trivial because `choose` and `least` have the same exception guard. The second is to prove that `i1` satisfies the normal

```
intset = mutable type

  based on IntSet

  least = proc (s: intset) returns (i: int)
    requires --
    modifies --
    ensures  i ∈ s^ ∧ ?
    except signals empty when s^ = {}

  choose = proc (s: intset) returns (i: int)
    requires --
    modifies --
    ensures  i ∈ s^
    except signals empty when s^ = {}

  max = proc (s: intset) returns (i: int)
    requires --
    modifies --
    ensures  i ∈ s^ ∧ ?
    except signals empty when s^ = {}
```

Figure 7.4: Partial Specifications of `intset` Procedures

postcondition of `choose`. This is also trivial because the postcondition of `least` implies the normal postcondition of `choose`.

If the order of the statements is reversed to form the code

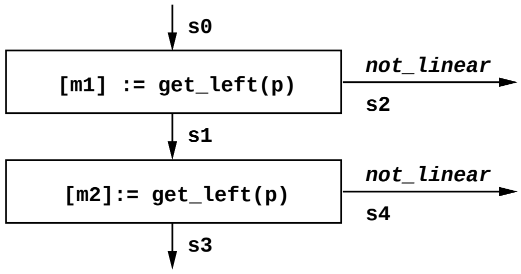
```
i1 := intset$choose(s)
i2 := intset$least(s)
```

it is not possible to replace `least` by `i1` because the normal postcondition of `choose` does not imply the normal postcondition of `least`.

The proof rules are also sufficient to eliminate or hoist calls to procedures that have partial specifications. For example, the specification of `get_left` in Fig. 7.2 suffices to eliminate the second call in the example

```
m1 := p.left
m2 := p.left
```

The flow graph for this code is



One proof obligation is to show that the value of `m1` at edge 1 satisfies the postcondition of the call:

$$\texttt{get\_left}.\mathit{Post}[\mathit{norm}](\sigma_1^{\mathrm{Str}}, \sigma_1^{\mathrm{Str}}, \sigma_1^{\mathrm{Env}}(\text{`}\texttt{p}\text{'}), \sigma_1^{\mathrm{Env}}(\text{`}\texttt{m1}\text{'})) \in \mathcal{T}_1$$

With the assertions in Fig. 7.3, this is equivalent to

$$\mathit{OnlyModifies}(\sigma_1^{\mathrm{Str}},\ \sigma_1^{\mathrm{Str}},\ \{\}) \bigwedge \sigma_1^{\mathrm{Env}}(\text{`}\texttt{m1}\text{'}) = \sigma_1^{\mathrm{Str}}(\sigma_1^{\mathrm{Env}}(\text{`}\texttt{p}\text{'})).\texttt{left} \in \mathcal{T}_1$$

The first conjunct is trivially true, and the second follows from the condition asserted by the proof rule for the first call to `get_left`.

The other obligation is to prove that the call will return normally:

$$\texttt{get\_left}.\mathit{Guard}[\mathit{norm}](\sigma_1^{\mathrm{Str}}, \sigma_1^{\mathrm{Env}}(\text{`}\texttt{p}\text{'})) \in \mathcal{T}_1$$

which is precisely the condition asserted by the proof rule for the first call to `get_left`.

Because the guard condition is not fully defined, the second proof would fail if any location is modified or allocated between the two calls to `get_left`. This is not a problem—if a user wants to facilitate the compiler's ability to eliminate or hoist calls to a procedure, he should give a full specification of that procedure.

7.2.2 Soundness of Proof Obligations

The proof obligations for performing the optimizations described in chapters 4 and 5 are sound even with the extensions for partial specifications. The reasons are that

1. Partial specifications only weaken the theory of an edge because they are translated into auxiliary predicates whose axiomatization is weaker than the axiomatization for the full specification. This means that partial specifications lead to theories containing fewer formulas.

2. The proof obligations have not been weakened.

The second point is obvious for eliminating dead code and for using SPIs because the proof obligations for these optimizations do not depend upon the auxiliary specifications.

The proof obligations for common subexpression elimination or for hoisting expressions out of loops, on the other hand, do depend on the auxiliary predicates. However, this is not a problem. The meaning of the auxiliary predicates has not changed: their axiomatization is consistent with that for fully specified procedures. Only the axiomatization of the auxiliary predicates is weaker.

For example, consider the code

```
i1 := max(s)
i2 := least(s)
```

The partial specifications of `least` and `max` in Fig. 7.4 do not allow the compiler to replace the call to `least` by the result of `max`, even though the specifications are identical. The reason is that there is no way to discharge `least`.*Post*[*norm*] because it is not fully defined.

7.3 Bounding Reachability

The biggest problem of supporting partial specifications is what to do when a `modifies` clause is omitted. Frequently, the safety of an optimization depends upon the `modifies` clause of a procedure call be between two relevant sections of code. Without the `modifies` clause to bound side effects, the optimization cannot be proved to be safe.

The compiler must have a way to deduce a conservative estimate for the `modifies` clause when it is omitted. Otherwise, partial specifications will provide little benefit: a user who invests the time to specify the procedures in one module will find that most of the optimizations that should work don't—because the compiler lacks the `modifies` clause of a procedure from another module. For example, some modules might have no specifications for any of their procedures.

In Speckle, the strategy I use is to bound the `modifies` clause of a procedure from its arguments. The strategy is based on two observations:

1. A procedure can modify only locations that are reachable from its arguments. (Recall that Speckle has no global variable names.)

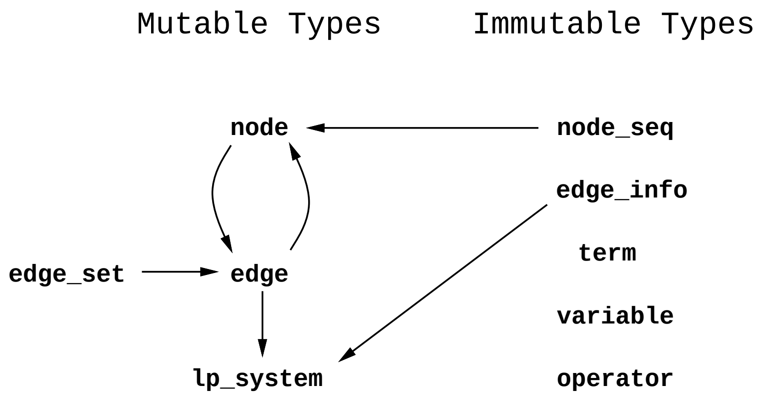


Figure 7.5: Reachability Graph

2. A data type can specify which locations are reachable from one of its values.

In Chapter 2, `reach`$^\wedge$ and `reach`$'$ were introduced to specify reachability. A problem with these functions is that they are not fully defined unless the value sort of every data type has a function `contents` that maps a value to the set of locations contained by the value.

To better support partial specifications, Speckle provides a mechanism for bounding reachability without having to define the `contents` functions. The mechanism is coarse-grained—it does not distinguish locations of the same type—but effective. Each data type must list the mutable types whose locations may be directly contained by instances of the type. These lists are specified in a `contains` clause for each data type. The `modifies` clauses deduced using `contains` clauses are often sufficient to prove that an optimization is safe, so users can avoid defining `contents` functions.

Because `contains` clauses are mandatory, each data type most provide at least a small specification. These specifications can then be used to deduce `modifies` clauses for every procedure.

Fig. 7.5 is a picture of reachability constraints for a set of types like those used to implement PSC. The meaning of an arrow from T_1 to T_2 is that instances of T_1 may contain locations of mutable type T_2. Only a mutable type can be the target of an arrow. Fig. 7.6 shows how the graph in Fig. 7.5 is specified in Speckle.

In the picture, the useful information is the arrows that are absent. For

```
node = mutable type
       contains edge

edge = mutable type
       contains node, lp_system

edge_set = mutable type
       contains edge

lp_system = mutable type
       contains --

node_seq = immutable type
       contains node

edge_info = immutable type
       contains lp_system

term = immutable type
       contains --

variable = immutable type
       contains --

operator = immutable type
       contains --
```

Figure 7.6: Specifications of Reachability

example, the graph in Fig. 7.5 can be used to deduce that a procedure with only arguments of types `edge_info` and `term` can modify only locations of type `lp_system`. The justification is that only `lp_system` is reachable from `edge_info` or `term`.

The information in `contains` clauses cannot always be deduced from code. Suppose, for example, that the implementation of `node` uses `edge_set` to represent the set of edges that enter a merge node. Because the representation of a node is concealed within the implementation of `node`, the values of type `node` never contain locations of type `edge_set`. Thus, there is no arrow from `node` to `edge_set` in Fig. 7.5—even though `edge_set` is reachable from the representation of `node`. It is sound to omit the arrow from `node` to `edge_set` because the implementation of `node` encapsulates any `edge_sets` in the representation, i.e., clients can never access the `edge_sets` directly.

7.3.1 Approximating Omitted `modifies` Clauses

To approximate omitted `modifies` clauses, I extend the formalization of Larch/Speckle. For each location sort `TLoc`, I introduce $\mathit{AllLocs}[T]$, a constant of sort `LocSet` that is axiomatized to denote the set of all locations of type T.

To approximate the `modifies` clause of a procedure whose signature is

```
P = proc (a1: T1, a2: T2, ... an: Tn) ...
```

I construct a `LocSet` L that is a superset of the locations that `P` may modify. The deduced `modifies` clause is *OnlyModifies*(*pre, post,* L), where L is

$$L = \bigcup_{i=1\ldots n} \mathit{Reach1}(\mathtt{T}_i) \cup \begin{cases} \{\mathtt{a}_i\} & \text{if } \mathtt{T}_i \text{ is mutable} \\ \{\} & \text{if } \mathtt{T}_i \text{ is immutable} \end{cases}$$

The definition of *Reach1* is

$$\mathit{Reach1}(T) = \bigcup_{T \to^+ T'} \mathit{AllLocs}[T']$$

$T \to^+ T'$ means T' is reachable from T in one or more steps in the reachability graph.

For example, the deduced modifies clause for

```
assert = proc (l: lp_system, t: term) ...
```

would be *OnlyModifies*(*pre, post,* $\{l\}$), which is equivalent to `modifies l`. Note that the `modifies` clause does *not* include *AllLocs*[`lp_system`].

The deduced modifies clause for

```
foo = proc (e: edge) ...
```

would be *OnlyModifies*(*pre, post,* $\{e\} \cup$ *AllLocs*[`node`] $\cup$ *AllLocs*[`edge`] $\cup$ *AllLocs*[`lp_system`]).

7.4 Related Work

The idea of writing formally only part of a procedure's specification is hardly new. For example, in [25], a procedure header may contain a formal postcondition, but the full postcondition need not be written formally. ANNA [40] provides a similar mechanism for both pre- and postconditions. Speckle differs in that it distinguishes partial specifications from full specifications. This distinction is used to prevent the compiler from performing unsound optimizations.

The idea of writing specifications incrementally has also been studied for algebraic specifications. In [4, 5], Bidoit distinguishes between "achieved" and "draft" specifications—a draft specification is analogous to a partial specification. His focus is on how to combine the theories of achieved and draft specifications in a modular fashion. The specification language is independent of any programming language.

The SETL compiler uses types to reason about reachability [14]. The compiler relies on definitions of containment and reachability for primitive types to identify when the source of a copy operation would always become garbage, obviating the need for the copy. SETL does not support data abstraction explicitly, so there is no use for `contains` clauses.

In Speckle, `contains` clauses could be used to perform storage optimizations, such as copy elimination or freeing an object when it becomes garbage. However, these optimizations might require the more precise specifications of reachability of Chapter 2 to work well in practice.

Other Larch interface languages also allow `requires` and `modifies` clauses to be omitted for terseness. However, the semantics of an omitted clause is very different: an omitted `requires` clause is equivalent to `requires --`, and an omitted `modifies` clause is equivalent to `modifies --`.

7.5 Summary

Speckle allows users to write interface specifications incrementally. A specification can be omitted entirely, or it can be written in part. Each specification clause that is not written fully is flagged as partial, either explicitly by the use of `?`, or implicitly by the absence of the clause. The distinction between partial clauses and regular clauses is used to prevent optimizations that are unsound.

The primary problem in supporting partial specifications is how to estimate the `modifies` clause when it has been omitted. The approach in Speckle is to compute a coarse upper bound for the set of locations accessible to a procedure from the types of its formal arguments. To allow this approximation, each data type uses `contains` assertions to specify the mutable types that may be directly accessible from a value of the type.

Chapter 8

Experience

My experience with PSC falls into two categories: hand-constructed examples used to test PSC during development, and later, a case study on small pieces of a large program. This chapter contains some general observations of PSC and a report on the case study.

8.1 Observations

At a high level, the job of PSC is to take code and specifications and use them to discharge proof obligations for optimizations. On the surface, this task seems no easier than the problem of program verification. Fortunately, however, the proofs required to justify optimizations are often quite simple.

One common proof obligation is to show that a predicate that is true at one point in a program is still true at a later point. PSC is usually able to discharge such obligations. For example, consider the statement `a[i]:= f(a[i])`, where `a` is a dynamic array. Because the `fetch` procedure checks that `i` is in bounds, the `store` procedure need not repeat the check unless `f` can change the bounds of the array. Here, the predicate needed to use the SPI of `store` is identical to the predicate asserted by the fact that `fetch` did not signal an exception. The main task of PSC is to use the `modifies` clause of `f` to show that the predicate is not affected.

The same kind of proof works to eliminate common subexpressions or hoist expressions from loops. Recall the `can_link_ends` example from Chapter 4, p. 52, where the goal is to eliminate a call to `polymer$get_left`. Here, the compiler needs to prove that both the postcondition and the guard condition asserted when the first call to `polymer$get_left` returns normally are still true when the second call is made. Once again, the proofs are easy, given the `modifies` clauses.

Another common proof obligation is to show that a predicate that is true at one point in a program implies another predicate at a later point. This kind of obligation is bit more challenging, because PSC must use information that relates the predicates to one another. For example, a statement `t:= table$create()` might establish `t = empty`. At a later point, a call `table$store(t,k,v)` might be optimized to use an SPI if $\neg$`defined(t,k)`. Here, in addition to showing that `t` is still empty, PSC relies on the LSL assertion $\neg$`defined(empty,k)`, which makes the proof easy.

In the previous example, the necessary LSL assertion was an axiom of tables, i.e., an assertion that is unlikely to be missing from the trait. However, proofs may also rely on assertions in the `implies` section of a trait, i.e., assertions that are intended to provide redundant information. For example, in the substitution example from Chapter 5, p. 71, the precondition for the faster implementation of `extend` is `v` $\notin$ `vars(apply(unbind(s,v),t))`. (Recall that this precondition is used to preserve the invariant that substitutions have no cyclic definitions.) To discharge this precondition in a context where `vars(t)=`$\{\}$, PSC could use the lemma `vars(tm)=`$\{\} \Rightarrow$ `vars(apply(sub,tm))=`$\{\}$ if it is asserted in the `implies` section of a trait.

Because there is no guarantee that LSL traits will contain the lemmas needed to detect optimizations, the author of an SPI may wish to add lemmas that are likely to be useful, such as the one above. One of the problems I encountered, however, is that an implication is essentially useless unless LP can convert the implication into a conditional rewrite rule. E.g., in the previous example, the rewrite rule

$$(\mathtt{vars}(\mathtt{tm}) = \{\}) \Rightarrow (\mathtt{vars}(\mathtt{apply}(\mathtt{sub},\mathtt{tm})) = \{\}) \rightarrow \ \mathtt{true}$$

cannot simplify the goal

$$\mathtt{v} \ \notin \ \mathtt{vars}(\mathtt{apply}(\mathtt{unbind}(\mathtt{s},\mathtt{v}),\mathtt{t}))$$

However, the conditional rewrite rule

$$\mathtt{vars}(\mathtt{tm}) = \{\} : \mathtt{vars}(\mathtt{apply}(\mathtt{sub},\mathtt{tm})) \rightarrow \ \{\}$$

will simplify the goal to `v` $\notin$ $\{\}$ in a context where `vars(t)=`$\{\}$.

Unfortunately, some implications cannot be converted into conditional rewrite rules because they would make the rewriting system nonterminating. For example, the implication

$$\mathtt{x} < \mathtt{y} \ \Rightarrow \ ((\mathtt{y} < \mathtt{x}) = \mathtt{false})$$

cannot be converted into a conditional rewrite rule because `x<y` is no simpler than `y<x`. In this case, the user can explicitly write the implication as a deduction rule. However, many implications cannot be written as deduction rules because they would trigger infinite loops. For example, the implication `x<y` $\Rightarrow$ `x<y+1`, if written as the deduction-rule `when x<y yield x<y+1`, would trigger an infinite loop in the presence of an assertion such as `0<1`.

Generally, the proofs least likely to succeed are those requiring induction, because it is difficult to obtain suitable loop invariants. One exception is the case when the goal is to prove that a value is constant during the execution of a loop. For example, in the `remove_duplicates` procedure from Chapter 1, p. 17, one of the optimizations is to show that `a.low`, the low bound of the dynamic array `a`, has the same value before and after the loop—this makes it possible to replace the occurrences of `a.low` after the loop by the value of `a.low` computed before the loop. Using induction, case analysis, and the LSL axiom that storing an element into an array does not change its size, PSC proves that the optimization is legal. Here, the proof succeeds because the value of `a.low` is constant over the loop.

8.2 Case Study: AC-Unify

To test the ideas in this thesis and to test PSC, I did a case study of AC-Unify, a program that unifies terms containing associative and commutative operators. I chose AC-Unify for several reasons. First, AC-Unify is not a toy. It is roughly 8,000 lines of commented source code, and the VAX executable is roughly 100 kilobytes. Second, AC-Unify is a well-structured program that makes good use of data abstraction. The program has 33 user-defined types and roughly 300 procedures. Finally, AC-Unify is written in CLU, so translating the code to Speckle did not introduce significant changes.

The purpose of the case study was to see what kind of performance improvements might be possible using the ideas in this thesis. Because of the program's size, I focussed on the critical portions of the code rather than translating all of it. The method I used was:

1. Profile the program to identify frequently executed routines.

2. Identify sections where SPIs, common subexpression elimination, or hoisting expressions might improve performance.

3. Translate the relevant sections into Speckle, adding both SPIs and specifications deduced from the comments and code.

4. Run the translated code through PSC.

5. Manually perform the source optimizations identified in the previous step on the original CLU code.

6. Recompile and measure the difference in execution time.

For the SPIs, I simulated my strategy for propagating the guard condition of an SPI through a level of data abstraction (see Section 5.4).

I wrote three SPIs—one for each of three procedures. Of these procedures, one had a guard condition propagated to a caller, so there were effectively four SPIs in total. The four SPIs were called from a total of 14 places in the code. Of the 14 call sites, eight could be optimized and six could not. PSC detected all of the optimizations.

Together, the eight specializations improved performance by 14%. Once I had identified the frequently-executed sections of code, it took me roughly a week to write the necessary specifications and invariants. It took PSC roughly five and a half minutes running on a 25 Mhz MIPS R3000 to process the 74 lines of code surrounding the eight call sites.

I deliberately wrote the weakest partial specifications needed to detect the optimizations. Partial specification worked well: specifications were necessary for only nine of the roughly 300 procedures in AC-Unify. In total, I wrote 67 lines of Larch/Speckle and 137 lines of traits—a small fraction of the 8,000 lines of source code and comments. Using the information in `contains` clauses, PSC deduced six `modifies` clauses that were essential for three of the optimizations.

Although I didn't find any places where eliminating a common subexpression or hoisting an expression would have improved performance noticeably, I did find several places where these optimizations could have been performed. I conjecture that these optimizations, when applied to the whole program, could lead to a noticeable improvement.

The next two sections describe the SPIs and the optimized call sites.

8.2.1 Specialized Procedures Implementations

Fig. 8.1 contains the signatures of the CLU procedures for which I wrote SPIs. (The `[]`'s in type expressions are used to delimit CLU's type parameters.) The guard conditions of the SPIs are given as comments. The SPI of each procedure avoids executing code to check each guard condition. For `set` and

```
set$insert = proc (s: set[elem], e: elem)
    % special when ¬(e ∈ s^)

mapping$insert = proc (m: mapping[dom,ran], d: dom, r: ran)
                 signals (exists)
    % special when ¬defined(m^, d)

assignment$create = proc (env: sequence[var])
                    returns (assignment[var,val])
                    signals (empty, duplicates)
    % special when NoDuplicates(env)
```

Figure 8.1: SPI of AC-Unify

```
substitution$store = proc (s: substitution,
                           v: variable,
                           t: term)
                     signals (exists)
    % special when ¬defined(s^, v)
```

Figure 8.2: Propagated SPI

`mapping`, the savings is linear in the size of `s` or `m`. For `assignment`, the savings is quadratic in the size of `env`.

Fig. 8.2 contains the signature of a procedure that was specialized by propagating the SPI of `mapping$insert`. The type `substitution` is represented as `mapping[variable,term]`, and `substitution$store` is the equivalent of `mapping$insert`. (AC-Unify makes `substitution` a separate data type to maintain invariants about `substitution`s that are not enforced by `mapping`.)

8.2.2 Optimized Call Sites

First Call to `set$insert`

Fig. 8.3 on p. 124 contains the CLU code where a call to `set$insert` was optimized. Irrelevant code is denoted by ellipses. The goal is to prove ¬(`d` ∈ `m.domain`). This goal follows directly from

- the precondition of `insert_pair`: `requires` $\neg$`defined(d,m)`
- a representation invariant maintained by `mapping`

  The invariant is that the `domain` field of the representation is equal to the domain of the mapping.

To make the representation invariant available to PSC, I included it in the precondition of `insert_pair`.

Second Call to `set$insert`

Fig. 8.4 on p. 125 contains the CLU code where a second call to `set$insert` was optimized. The goal is to prove $\neg$(`pt.values[i]` $\in$ `result`). The proof depends on

- the representation invariant `NoDuplicates(pt.values)` maintained by `partition_tree`

  This invariant is encoded as a precondition of `value` in the Speckle version of the code.

- the fact that `t_seq` is immutable

  This fact is true for the Speckle version of the code, in which the type parameter `t` is instantiated by an immutable type.

  Because both `rep` and `t_seq` are immutable, `NoDuplicates(pt.values)` is trivially preserved by any code that does not assign to `pt`. There is no assignment to `pt` in the implementation of `value`.

- the `ensures` clause of `t_set$create`, which establishes that `result` is initially empty
- the `modifies --` clauses of `t_seq$indexes`, `sequence[bool]$fetch`, and `t_seq$fetch`
- the `ensures` clause of `t_seq$fetch`
- the specification of `t_seq$indexes`, which ensures that `i` is within the bounds of `pt.values`
- the `ensures` clause of `t_set$insert`
- LSL axioms for sets

- the three lemmas

```
∀ sq: Sequence, i,j: Int, s: Set

  (NoDuplicates(sq)
   ∧ (s ⊆ seq2set(prefix(sq, i-1)))
   ∧ InBounds(sq, i))
   ⇒ ¬(sq[i] ∈ s);

  (InBounds(sq, i) ∧ i ≤ j)
   ⇒ (sq[i] ∈ seq2set(prefix(sq, j)));

  (s ⊆ seq2set(prefix(sq, i)))
   ⇒ (s ⊆ seq2set(prefix(s, i+1)));
```

  where `prefix(sq, i)` is the subsequence of `sq` from index `1` to `i`, and `seq2set(sq)` converts a `Sequence` to a `Set`.

PSC uses proof-by-induction and proof-by-cases to discharge the goal.

In addition to specializing `set$insert`, PSC detects that no bounds checks are needed for `pt.values[i]` and that `pt.values` can be hoisted out of the loop in Fig. 8.4. However, these optimizations do not improve performance significantly.

Call to `mapping$insert`

Fig. 8.5 on p. 125 contains the CLU code where a call to `mapping$insert` was optimized. The goal is to prove ¬`defined(v, compose)`. The proof of this goal depends on

- the `ensures` clause of `predict`

  The procedure `predict` ensures that it returns a new, empty `mapping`. The integer argument is merely a hint as to how many domain elements are likely to be defined.

- the `ensures` clause of `elements`

  The iterator `elements` ensures that all no variable is yielded twice.

- the `modifies --` clause of `elements`

- the `ensures` clause of `insert`

  The procedure `insert` ensures that the only possible addition to the domain of `compose` is the variable `v`.

- the `modifies` clause of `apply`

  From the `contains --` clauses of types `mapping` and `term`, PSC deduces that `apply` cannot modify `compose`. At most, `apply` may modify `s1`, and `s1` and `compose` are distinct.

- the lemma

  ```
  ∀ s,s1: VariableSet, v: Variable
    when v ∈ s == false
    yield (s1 ⊆ s) ⇒ ¬(v ∈ s1);
  ```

- the LSL axioms for mappings

PSC discharges the proof obligation by induction on the loop.

Calls to `assignment$create`

Fig. 8.6 on p. 126 contains the CLU code where two calls to `assignment$create` were optimized. For the purpose of the optimization, the two call sites are essentially identical. The only differences are that `assigns` and `assign_total` use different instantiations of `assignment`, and they elided code also changes.

The goal is to prove `NoDuplicates(vs)`. This goal follows from

- a representation invariant of solution

  The invariant is that the `vars` field of the representation does not contain duplicates. This invariant establishes `NoDuplicates(s.vars)`.

- the `ensures` clause of `var_vec$v2seq`

  The type `var_vec` is closely related to the type `var_seq`. The only difference is that a `var_vec` has a unique identifier, which a `var_seq` does not. The procedure `v2seq` ensures that the `var_seq` it returns is the equal to the sequence contained in `vs`. Thus, `NoDuplicates(vs) = NoDuplicates(s.vars)`.

Calls to `substitution$store`

Fig. 8.7 on p. 127 contains the CLU code where three calls to `substitution$store` were optimized. For the first call site, the goal is to prove `¬defined(sigma,v2)` The proof of this goal relies on

- the `ensures` clause of `substitution$new`

  The procedure `substitution$new` ensures that it returns an empty substitution.

- the LSL axioms for substitutions (mappings), which specify that nothing is defined in the empty substitution.

- the deduced `modifies --` clause of `term$similar`

  PSC deduces this `modifies` clause from the specification of immutable type `term`, which asserts `contains --`.

The proofs for the second and third call sites also rely on

- the deduced `modifies --` clause of `term$get_vars`

  PSC deduces the `modifies` clause from the specification of `term`.

  (In the `general_unify`, the calls to `term$get_vars` are written as `t2.vars` and `t1.vars`, which exploit CLU's shorthands.)

- the deduced `modifies` clause of `var_set$exists`

  Since the specification of mutable type `var_set` asserts `contains --`, PSC deduces that `var_set$exists` can at most modify its argument, a `var_set`.

8.3 Summary

In a case study on small pieces of a large program, PSC detected optimizations that lead to an 14% improvement in performance. None of the eight optimizations would have been performed using conventional techniques, including interprocedural analysis. Each optimization depended on the representation invariant of a data type or on properties of data values specified in LSL axioms. Also, some optimizations depended on deduced `modifies` clauses.

```
mapping = cluster [dom, ran: type]  ...

  rep = record[domain: dom_set,     % domain of mapping
              ⋮
               ]
  dom_set = set[dom]

    ⋮

  % Requires: "d" is not defined in "m"

  insert_pair = proc (m: rep, d: dom, r: ran)
    dom_set$insert(m.domain, d)          % optimized call
    ⋮
    end insert_pair
```

Figure 8.3: First Optimized Call Site of `set$insert`

```
partition_tree = cluster [t: type] ...

  rep = struct[values: t_seq,   % contains no duplicates
               ⋮
                ]
  t_seq = sequence[t]
  t_set = set[t]
  ⋮

  value = proc (pt: rep) returns (t_set)
    mask: sequence[bool] := ...
    result: t_set := t_set$create()
    for i: int in t_seq$indexes(pt.values) do
        if mask[i] then
           t_set$insert(result, pt.values[i])   % optimized call
           end
        end
    return(result)
    end value
```

Figure 8.4: Second Optimized Call Site of `set$insert`

```
substitution = cluster ...

  rep = mapping[variable, term]
  ⋮

  mul = proc (s1, s2: rep) returns (rep)
    compose: rep := rep$predict(rep$size(s1))
    for v: variable, t: term in rep$elements(s2) do
        rep$insert(compose, v, apply(s1, t))    % optimized call
        end
    ⋮
    end mul
```

Figure 8.5: Optimized Call Site of `mapping$insert`

```
solution = cluster [vartype: type] ...

  rep = record[vars: var_vec,  % vars contains no duplicates
               ⋮
               ]
  var_vec = vector[vartype]
  var_seq = sequence[var]
  var_int_assn = assignment[vartype, int]
  vis_assn = assignment[vartype, int_seq]
  ⋮

  assigns = proc (s: rep) returns (vis_assn)
    vs: var_seq := var_vec$v2seq(s.vars)
    a: vis_assn := vis_assn$create(vs)         % optimized call
    ⋮
    end assigns

  assign_total = proc (s: rep) returns (var_int_assn)
    vs: var_seq := var_vec$v2seq(s.vars)
    a: var_int_assn := var_int_assn$create(vs) % optimized call
    ⋮
    end assign_total
```

Figure 8.6: Optimized Call Sites of `assignment$create`

```
general_unify = iter (t1,t2:term, ur:unif_registry, gs:gen_sym)
                yields (substitution) signals (not_unifiable)
  sigma: substitution := substitution$new()
  if term$similar(t1, t2) then
     yield (sigma) return
     end
  tagcase t1
    tag var (v1: variable):
      tagcase t2
        tag var (v2: variable):
            sigma[v2] := t1              % optimized call
            yield (sigma)
        tag nonvar (nv2: nonvar):
            if var_set$exists(t2.vars, v1)
               then signal not_unifiable
               end
            sigma[v1] := t2              % optimized call
            yield (sigma)
        end
    tag nonvar (nv1: nonvar):
      tagcase t2
        tag var (v2: variable):
            if var_set$exists(t1.vars, v2)
               then signal not_unifiable
               end
            sigma[v2] := t1              % optimized call
            yield (sigma)
        tag nonvar (nv2: nonvar):
            ⋮
        end
    end
  end general_unify
```

Figure 8.7: Optimized Call Sites of `substitution$store`

Chapter 9

Summary and Conclusion

Specifications have been advocated because they make it easier for people to reason about programs. The primary conclusion of this thesis is that specifications can make programs run faster because they make it easier for *compilers* to reason about programs.

The key ideas leading to this conclusion are:

- **Enhancement of Conventional Optimizations.** In most conventional compilers, optimizations like common subexpression elimination and code motion are restricted to expressions that don't contain procedure calls. In this thesis, I used specifications to generalize such optimizations to handle procedure calls as well. This eliminated one of the disparities between operations that happen to be primitive to a source language from operations that are defined by the user. I also used specifications to improve side effect analysis, which is needed to detect many kinds of optimizations.

  The primary advantage of using specifications is that specifications are simpler than code. For example, code analysis is impractical for deducing the axioms of a data type or for distinguishing between visible and invisible (benevolent) side effects.

  Another advantage is that specifications contain information not found in code, such as the fact that an implementation need not be deterministic. However, while this information enables optimizations that are impossible without specifications, it is unclear how often such optimizations would apply in practice.

- **Specialized Procedure Implementations.** SPIs allow the programmer to hide several related procedures behind a single interface. This reduces the burden on clients, who would otherwise have had to choose which version of the procedure to call, because the compiler chooses the appropriate version. SPIs also improve modularity, since the programmer can add or remove a specialized implementation and let the compiler worry about updating client code.

User-defined optimizations like SPIs were previously studied in [25]. Because the approach there was based on transformation rules rather than specifications, it lacked modularity. Furthermore, this thesis appears to be novel in that it addresses the problem of propagating optimizations across levels of data abstraction.

- **Partial Specifications.** Partial specifications allow users to write specifications incrementally. This makes it easier to use specifications to improve performance because not all of the specifications have to be written—some specifications can be omitted altogether, while others are written in part or in full. It also allows users to focus on common library routines, where the investment of writing specifications is amortized over many callers. Although writing partial specifications has been proposed before, this work is novel in distinguishing partial specifications from other specifications to prevent unsound optimizations.

  To make partial specifications work well, the compiler must be able to estimate `modifies` clauses when they are omitted. This thesis presented a way of computing estimates from specifications of reachability. Code analysis would be another possibility.

To evaluate the potential utility of these ideas, I designed a programming language that incorporated them and built PSC, a prototype implementation. PSC, which identified but did not apply optimizations, was then used on several small programs and one large one. These experiments demonstrated that several issues need to be addressed before these ideas can be put to practical use.

The first is compiler running time: PSC is too slow. There are several ways in which attempts to prove the soundness of optimizations can be sped up. First, the functions and axioms used to model program states could be implemented in a more efficient way than by relying on general-purpose inference mechanisms. Second, the strategies for proofs by cases and induction could cache previous proof attempts to avoid repeating proofs. Finally, the compiler could improve sharing of common data between the logical systems of the different edges.

A different way to make PSC faster would be to focus its attention on only the potentially most useful optimizations. For example, one could integrate the optimizer with profiling tools.

Because the compiler relies on specifications, bugs in specifications can lead to unsafe optimizations. Therefore, a pragmatic issue is how to identify bugs in specifications.

One way to locate bugs is to verify that the specifications and code are consistent, but there are other possibilities. A specification checker 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 a problem disappears.

A final issue is that of "tuning" specifications to enhance the compiler's ability to find opportunities for optimization. PSC is particularly sensitive to the way specifications are formulated. Attention should be devoted to characterizing the impact of different formalizations of the same specification, so that useful advice can be given to specifiers.

Despite the above concerns, I remain optimistic about the utility of the ideas contained in this thesis. The experiments with PSC were encouraging, demonstrating that significant performance improvements could be obtained. In the AC-Unify case-study, just four SPIs improved performance by 14%. Furthermore, these experiments made it clear that partial specifications can be used productively. They allow one to obtain performance enhancements that are large relative to the effort required of the programmer. In time, specifications will greatly reduce the need for programmers to compromise code readability, safety, and modularity when tuning programs for performance.

Bibliography

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, 1987.

[2] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 1979 Conference on Principles of Programming Languages*, pages 29–41. ACM, January 1979.

[4] Michel Bidoit. *Pluss, un langage pour le développement de spécifications algébriques modulaires.* PhD thesis, Université Paris-Sud, Orsay, May 1989. Thèse d'Etat.

[5] Michel Bidoit. Development of modular specifications by stepwise refinements using the PLUSS specification language. In *Proc. of the* IMA *Unified Computation Laboratory Conference, Stirling, Scotland, July 1990*, pages 171–192. Oxford University Press, 1992.

[6] W. W. Bledsoe and M. Tyson. The UT Interactive Prover. ATP 17, University of Texas Mathematics Dept., May 1975.

[7] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the 1990 Conference on Programming Language Design and Implementation*, pages 296–310. ACM, June 1990.

[8] Jolly Chen. The Larch/Generic interface language. S. B. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., May 1989.

[9] John Cocke and Jacob T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes.* Courant Institute of Mathematical Sciences, New York University, 1970.

[10] Keith D. Cooper. Analyzing aliases of reference formal parameters. In *Proceedings of the 1985 Conference on Principles of Programming Languages*, pages 281–290. ACM, January 1985.

[11] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 1977 Conference on Principles of Programming Languages.* ACM, 1977.

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

[13] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.

[14] Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, January 1983.

[15] S. J. Garland, J. V. Guttag, and J. J. Horning. Debugging Larch Shared Language specifications. *IEEE Transactions on Software Engineering*, 16(9):1044–1075, September 1990.

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

[17] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging Larch Shared Language specifications. Report 60, DEC Systems Research Center, Palo Alto, CA, July 4, 1990.

[18] Steven M. German. Verifying the absence of common runtime errors in computer programs. Technical Report CS-81-866, Stanford, June 1981.

[19] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[20] Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the 1990 Conference on Programming Language Design and Implementation*, pages 272–282. ACM, June 1990.

[21] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(5):24–36, 1985.

[22] J. V. Guttag, J. J. Horning, with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[23] John V. Guttag, James J. Horning, and Andrés Modet. Report on the Larch Shared Language: Version 2.3. Report 58, DEC Systems Research Center, Palo Alto, CA, April 14, 1990.

[24] David Hinman. On the design of Larch interface languages. Master's thesis, Department of Electrical Engineering and Computer Science, M.I.T., January 1987.

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

[26] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[27] Cliff B. Jones. *Software Development: A Rigorous Approach.* Series in Computer Science. Prentice-Hall International, 1980.

[28] Cliff B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice-Hall International, 1986.

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

[30] Donald Knuth. An Empirical Study of FORTRAN Programs. *Software–Practice and Experience*, 1:105–133, 1971.

[31] Butler L. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the programming language euclid. Technical Report CSL-81-12, Xerox PARC, October 1981.

[32] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 33–38. ACM, October 1983. Also available in IEEE Software, Vol. 1, No. 1.

[33] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the 1988 Conference on Programming Language Design and Implementation*, pages 21–34. ACM, June 1988.

[34] Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. In Ursula Martin and Jeannette M. Wing, editors, *First International Workshop on Larch*. Springer-Verlag, July 1992.

[35] Richard Allen Lerner. *Specifying Objects of Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991. TR CMU–CS–91–131.

[36] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Lecture Notes in Computer Science No. 114. Springer-Verlag, 1981. Also available in [37].

[37] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The M.I.T. Electrical Engineering and Computer Science Series. M.I.T. Press, Cambridge, Ma, 1986.

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

[39] David Luckham. *Program Verification and Verification-Oriented Programming*. Springer-Verlag, 1977.

[40] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Bruckner, and Olaf Owe. *ANNA Reference Manual*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[41] John McHugh. Towards the generation of efficient code from verified programs. Technical Report 40, University of Texas at Austin, March 1984.

[42] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[43] A. Neirynck, P. Panangaden, and A. J. Demers. Computation of aliases and support sets. In *Proceedings of the 1987 Conference on Principles of Programming Languages*. ACM, 1987.

[44] Greg Nelson and David Detlefs. Extended static checking. Work in progress at DEC Systems Research Center, May 1993.

[45] D. L. Parnas. Information distribution aspects of design methodology. In *Proceedings of the 1971 IFIP Congress*, pages 339–344, May 1972.

[46] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 1988 Conference on Principles of Programming Languages*. ACM, 1988.

[47] Richard Sites. Proving that computer programs terminate cleanly. Technical Report CS-74-418, Stanford, 1974.

[48] J.M. Spivey. *Understanding Z*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[49] Richard M. Stallman. Using GNU CC. For GCC version 2.3. Available by anonymous ftp from prep.ai.mit.edu, December 1992.

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

[51] Mark T. Vandevoorde. Specifications can make programs run faster. In *Proceedings of TAPSOFT '93*. Springer Verlag, April 1993.

[52] Jeannette M. Wing. A two-tiered approach to specifying programs. Technical Report MIT/LCS/TR-299, M.I.T., 1983.

[53] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[54] Katherine Yelick. A generalized approach to equational unification. Technical Report MIT/LCS/TR-344, M.I.T., August 1985.