PROLAC: A LANGUAGE FOR PROTOCOL COMPILATION

by

EDDIE KOHLER

Submitted to the
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
in partial fulfillment of the requirements for the degree of
Master of Science

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September, 1997

PROLAC: A LANGUAGE FOR PROTOCOL COMPILATION

by

EDDIE KOHLER

Submitted to the Department of Electrical Engineering and
Computer Science on August 29, 1997 in partial fulfillment of the
requirements for the degree of Master of Science

ABSTRACT

Prolac is a new statically-typed object-oriented programming language designed for implementing network protocols. Prolac is designed to make protocol specifications *readable* to human beings, and thus more likely to be correct; easily *extensible* to accommodate protocol enhancements; and *efficient* when compiled.

We present an overview of the Prolac language and a discussion of issues and principles in its design, as well as a preliminary language reference manual. The *prolacc* optimizing protocol compiler is also described. A prototype TCP specification is presented that is both readable and extensible; experience with the specification suggests that, even untuned, Prolac overhead is negligible on normal networks.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Computer Science

CONTENTS

ACKNOWLEDGEMENTS

This thesis presents joint work with Frans Kaashoek. In particular, the TCP implementation presented in Chapter 5 was designed and implemented by Frans Kaashoek with design advice from Eddie Kohler. This research was partially supported by an NSF Graduate Research Fellowship.

for Frans and for Rebecca;
because of my family
and the state of Massachusetts;
near David Mazières;
beside Maria and Nummi
and the ever-unique Al Pua;
and despite Count Dorkula
and her lovely sidekick,
Squirrelgirl

1

INTRODUCTION

Designing and implementing network protocols with conventional languages and tools is difficult. The protocols themselves are hard to design; implementing a protocol correctly is another challenge [WS95]. Furthermore, protocol efficiency has become vital with the growing importance of networking, and the occasional need for protocol extensions [Ste97, BOP94] only complicates the issue. Unfortunately, these tensions work against one another. Many optimizations which make protocol code more efficient also tend to make it much harder to understand [MPBO96], and therefore harder to get right. Extensions affect deeply buried snippets of protocol code rarely identifiable a priori. Finally, the clearest organization of protocol code is often among the slowest.

Specialized language tools are a natural area to investigate for a solution to this software engineering problem. Most previous work, however, has focused on only one of the three issues: correctness. Research has been particularly active in formal specification languages amenable to machine verification [BB89, DB89]; while it is possible to use one of these specification languages to generate an implementation semi-automatically, very high performance is often precluded by the languages themselves. Some work has focused on high-performance implementation [AP93, CDO96], but these languages may not be suitable for existing protocols, either by design or due to limitations in the underlying language model.

This thesis presents a language, Prolac, and its compiler, *prolacc*, which address all three issues in protocol implementation: correctness, efficiency, and extensions. Our protocol compiler project, of which this thesis is the first concrete result, has three specific goals: to implement protocol-specific optimizations, thus creating high-performance protocol implementations; to facilitate protocol extensions; and to make protocol implementations more tractable to human readers, and thus easier for people to reason about.

Prolac is a statically-typed, object-oriented language. Unlike many such languages, it focuses on small functions rather than large ones: its syntax encourages the programmer to divide computation into small pieces, and Prolac features, such as namespaces, help a programmer name such small pieces appropriately. A protocol specification divided into small rules is both easier to read and easier to change; a small extension is more likely to affect just a few small functions in Prolac, which can be overridden by a subtype without complicating the base protocol code. Several novel features—specifically, module operators and implicit rules—help make protocol specifications easier to understand.

The *prolacc* compiler compiles a Prolac specification to C code. It applies several protocol-specific optimizations, such as extensive inlining and outlining; these optimizations are specified in the Prolac language as annotations to modules or to individual expressions. Both the language and the compiler have been designed to produce efficient generated code—our goal was to equal or exceed the performance of protocol implementations hand-written in C. In particular, *prolacc* can analyze away almost every dynamic dispatch and structure assignment.

We also describe a prototype implementation of the TCP protocol [Pos81] in the Prolac language, written largely as a proof of concept. We focused on TCP because it is a large, complex, important, and well-documented protocol. TCP is widely recognized as being difficult to implement well; in fact, books have been written about its implementation [WS95].

Our Prolac TCP specification is divided into small, sensibly interlocked pieces. Logical extensions to the base TCP protocol, such as delayed acknowledgement and slow start, are implemented in the specification as extensions to a set of base modules; very simple definitions determine which extensions, if any, are compiled, and even their relative order of execution. The TCP specification is highly extensible while staying highly readable—much of the specification is very similar to language in the standard reference to TCP [Pos81]. Prolac TCP can communicate with other TCPs, and experiments show that Prolac is not a bottleneck under normal networking conditions.

1.1 Related work

The International Organization for Standards (ISO) has defined two formal description techniques originally intended for developing the ISO OSI protocol suite. These techniques are LOTOS and Estelle. LOTOS [BB89], based on Milner's Calculus of Communicating Systems [Mil80], is an algebraic technique with functional properties. Like many functional languages, it is very effective for describing its fundamental abstraction, processes. Unfortunately, also like many functional languages, these abstractions are rigid and may not fit existing protocols without some pain—exactly the pain that protocol languages are supposed to avoid. LOTOS users report some problems with using the language for specification [LMD90], and LOTOS presents very serious challenges to the compiler writer, including valid programs which potentially generate an infinite number of processes [WvB89]. It seems doubtful that LOTOS specifications can be made to run efficiently.

Estelle [DB89] is one of many protocol languages based on a finite state machine model much like that often used in parsers. Estelle, like LOTOS, includes asynchronous parallelism; it is based on Pascal, and includes a module system (bound up with the parallelism structure) where processes (modules) communicate through broadcast signals. Estelle can be used to create semi-automatic implementations of reasonable performance [SCB90]. However, experience with large protocols is not reported, and high performance is not discussed. Furthermore, the state machine model and its cousin the Petri machine model have intrinsic problems for modeling protocols: the division into states often does not correspond to anything real in the protocol, and relationships between states can become very complicated and difficult to change, even in carefully layered protocols [vB86].

Esterel [BdS91] is a version of Estelle without asynchronous parallelism: an Esterel specification has a defined sequential execution. High performance Esterel compilers are being developed [CDO96]; however, full implementation of a large protocol is not reported. In terms of language, Esterel suffers from many of the same problems as Estelle due to their common extended finite state machine model.

RTAG [And88] is based on a context-free attribute grammar. RTAG provides a relatively natural syntax with equivalent modeling power to extended finite state machines. Efficient compilers are reported in the literature [HA89], although "efficient" turns out to mean "arguably efficient enough for research use", i.e., simple protocols suffer a factor of 2 slowdown. This slowdown comes partially from parallelism in the language. RTAG is not always suitable for existing protocols, just as *yacc* is not always suitable for describing computer languages.

The *x*-kernel [HP91] provides an infrastructure and various tools for creating protocol stacks. Prolac is complementary with the *x*-kernel, which focuses on organizing protocol stacks; Prolac is primarily designed for implementing a single protocol.

Morpheus [AP93], an object-oriented protocol language, enforces a large number of constraints on the protocol programmer. These constraints are restrictive enough that "existing protocol specification[s] may not be implementable in Morpheus." While some of the constraints are meant to increase the knowledge available to the compiler to enable domain-specific optimizations, others seem to exist solely to prevent the programmer from making bad design choices. Impressive results are reported for UDP speedup, but we regard Morpheus's inflexibility and inability to implement real protocols as definitive.

1.2 Motivation

This section describes some requirements we formulated in response to the triple goal of tractability, extensibility, and efficiency, and how those requirements guided Prolac's design.

- **The language must be easy to write.** Programs in the language should be generally understandable to most implementors without requiring major grounding in new techniques.

This requirement steered us away from existing models, which are often obscure from the point of view of traditional programming languages, and toward a simple expression model with generally C-like syntax.

- **The language must handle most of the protocol's implementation, including some lower levels.** Semi-automatic protocol implementation generally means that the implementor provides vital pieces of the implementation in an auxiliary file [SB90]. While some code rightfully belongs outside a protocol language, most languages make the exclusion boundary too high. Separation can cause problems when either specification or implementation changes, and acts as a deterrent against reflecting vital implementation concerns in the specification. These problems make high-performance semi-automatic implementation quite difficult.

  Prolac's object-oriented features and namespaces address the requirement for a readable high-level specification. The requirement for low-level implementation led us to treat Prolac as *bilingual;* like *yacc* [Joh75], Prolac allows programmers to escape to C to express complex implementation details. Both high and low levels can easily coexist in a Prolac program through subtyping, which can be used to provide implementation details for a high-level specification.

- **The compiler must generate efficient code for well-written specifications.**

  Prolac's simplicity is a result of this requirement: a simple language is easier to compile into efficient code. Also, Prolac allows the programmer to specify various hints to improve the generated code.

- **The language should support integrated layer processing.** Integrated layer processing, or ILP, where processing for the various protocol layers is performed all at once rather than sequentially, has proven very important for high network performance [CT90, Cla82].

  The inheritance mechanism should apply naturally to ILP, even when layers are specified independently.

A study of the other protocol languages described in §1.1 led to other requirements:

- **Asynchronous parallelism is a mistake.** Every protocol language including asynchronous parallelism has proved difficult or impossible to compile into high-performance code, often demonstrably due to that parallelism.

  Prolac has no asynchronous parallelism—or any parallelism at all, for that matter: a Prolac specification is wholly sequential.

- **Inflexible abstractions cause problems.** While whole protocols (i.e., TCP, IP, Ethernet) can fit well into a fixed abstract framework [HP91], the internals of these protocols are more complex and often don't match up to the idealized state or process abstractions in protocol languages. This forces some uncomfortable twisting to make the protocol fit the language—making the specification harder to understand—or may make it impossible to implement the protocol in the language [AP93]. Thus, any enforced abstraction is also an enforced limitation, restricting the language so that only particular protocols can be naturally implemented.

  This consideration caused us to leave all protocol-specific abstractions out of our protocol language. Prolac is a domain-specific language, in that many of its features, both small and large, were formulated in direct response to how protocols work; however, it is completely without domain-specific abstractions.[1]

Other requirements were inspired by properties common to many protocols.

- **Protocols are often described in informal specifications as decision trees augmented with actions.** See, for example, the TCP specification [Pos81].

  Prolac was designed so that this style of specification easily translates to Prolac.

- **Protocols are not organized around data abstractions.** Consider the large and complex TCP protocol. This protocol is built around two central objects: the transmission control block (TCB) and a segment, or incoming packet. Even considering auxiliary objects like buffers and timers, these objects simply do not determine a reasonable organization of TCP's voluminous code.

  The implicit rule mechanism (§3.2.2) was inspired by this property. It allows a programmer to write maximally concise and understandable specifications, even when code is not organized around data objects.

- **Protocols evolve through accretion of small extensions.** As mentioned above, these extensions involve relatively small changes to deeply buried, seemingly arbitrary pieces of protocol code. The first protocol designer cannot limit such extensions to a few specific places: the right places are impossible to identify a priori. We expect the need for extensions to grow as user-level protocol implementations become more important.

  This inspired our decision to prohibit access control in Prolac: a module can override any function from any of its supertypes. Module operators (§3.2.1) allow module designers to suggest an interface to a module; however, that module's users can still access hidden rules, albeit with a more inconvenient syntax.

[1] Well, almost: see §3.3.1 regarding the seqint type.

- **Protocols behave predictably at run time.** Protocols are generally specified in the form of a state-transition graph; the next step in any protocol processing is always very well-defined.

  This means, in Prolac, that the dynamic dispatch the language provides will often go unused; at run time, most call sites will always dispatch to a single function, not one of a set. We further note that, in specifications we designed, this unique function would often be the most overridden function available. *Prolacc*, therefore, optimizes this case to a static dispatch when it is legal to do so.

1.3 Contributions

This thesis makes the following contributions: The design of the Prolac language, including the novel concepts of module operators and implicit rules; *prolacc*, a working Prolac compiler generating reasonably efficient and high-level C code; and a prototype TCP specification in Prolac which is readable, extensible, and can communicate with other TCPs.

1.4 Thesis organization

Chapters 2 and 3 describe the Prolac language; Chapter 2 gives a general overview, while Chapter 3 provides a more detailed discussion. Chapter 4 describes the internal structure of the *prolacc* compiler and describes some of the algorithms it uses. Chapter 5 gives a brief description of our prototype TCP implementation; Chapter 6 provides a brief summary and directions for future work. Finally, Appendix A is a preliminary version of the Prolac language reference manual.

2
LANGUAGE OVERVIEW

This chapter introduces the Prolac language, providing a quick flavor of its syntax and how it is used by developing a trivial example. Chapter 3 will discuss Prolac's design and some of its important features.

Some familiarity with conventional statically-typed object-oriented languages (e.g., C++ or Java) is assumed. This chapter will not describe any features in detail; the curious reader is referred to Appendix A, which contains a preliminary version of the Prolac language reference manual. The text contains references to the relevant sections of the manual.

2.1 Basics

Prolac is an object-oriented language, by which we mean that it has *data abstraction* (i.e., the user can define new data types and operations acting on them), *subtyping* (i.e., a user-defined type can extend the definition of other user-defined types), and *dynamic dispatch* (i.e., for some function calls, one of a set of actual function bodies may be executed at run time, depending on the run-time type of a special argument).

Prolac is also *statically typed:* like C++ and Java, but unlike Smalltalk, Lisp and Self, the compiler knows the type of every object in a Prolac program. The type the compiler knows for an object, called its *static type,* may be different from the run-time type of the object, called its *dynamic type;* specifically, an object's dynamic type may be a subtype of its static type.

Like ML and Lisp, Prolac is an *expression language:* everything returns a value, and there is no concept of a statement which is not an expression. This contributes to Prolac's readable but very concise syntax. Prolac has even fewer complicated control constructs than ML and Lisp—it has no looping constructs, for example. Looping must either be expressed through recursion or in C. This is not generally a problem when specifying protocols.

Prolac is designed for relatively small protocol specifications (i.e., less than 5,000 lines of code). The Prolac compiler can therefore read the source for an entire program at once, enabling some important global optimizations. For example, the Prolac compiler can discover that a rule[1] is never overridden, in which case any call of that rule can use slightly faster static dispatch or—even better—be inlined.

A Prolac compiler generates C code from a Prolac specification. Prolac itself is bilingual; a Prolac program can specify arbitrary C code to be executed inside a rule. This allows tight integration with C without either complicating Prolac or forcing implementation concerns into a separate file.

Prolac is call-by-value rather than call-by-object. C-style explicit pointer types must be used to get true dynamic dispatch.

2.2 Modules and rules

The basic unit of program organization in Prolac is the *module;* a module in Prolac is like a class in many other object-oriented languages (§A.3). Modules have associated data, called*fields* (§A.6), and code, called *rules* (§A.5). Here is a prototype for a module implementing rational numbers:

```
module Rational {
  field num :> int;
  field den :> int;
  constructor(n :> int, d :> int) ::=
    num = n, den = d;
  negative ::=  // Is it negative?
    (num < 0) != (den < 0);
}
```

The fields num and den have separate values for each object of type Rational. A field is often called a slot or class member in other languages. The ‘:>’ operator declares a type; it should be read “is a”.

The *constructor,* named constructor, is used to create objects of type Rational (§A.5.3); here, it initializes the num and den fields to values the user must pass as constructor parameters.

Finally, negative is a *rule* taking no parameters (§A.5). The expression following ‘::=’ is the *body* of the rule, which is executed whenever the rule is called. The negative rule actually returns type bool (the Boolean type, with values true and false; §A.8.4); because bool return types are so common in Prolac protocol specifications, we allow them to be elided. negative is a *dynamic rule* (often called a method), so it has access to a “current object” of Rational type. Prolac also supports *static rules* which can be called without reference to any object (§A.5.1).

[1] Prolac’s blanket term for “function” and “method”; see §3.1.1 for discussion.

2.3 Imports and supertypes

Here is how a module M would use the Rational module we've created:

```
module M has Rational {
  field ir :> Rational;
}
```

Rational is listed in M's *module header* (§A.3.1), in the *has clause.* A module must explicitly declare every module it uses by *importing* it—that is, by listing it in the has clause (§A.3.3). If we had left Rational out of the has clause, the field definition would have caused an undefined name error.

To demonstrate supertypes, we extend Rational to enforce an additional invariant (that the denominator must be positive):

```
module Pos-Denom-Rational :> Rational {
  constructor(n :> int, d :> int) ::=
    Rational(n, d),  // first, call parent's constructor
    normalize-sign;
  normalize-sign ::=
    den < 0 ==> (num = -num, den = -den);
  negative ::= num < 0;
}
```

Supertypes—more specifically, *parents,* which are direct supertypes—are also declared in the module header, after a type declaration operator ':>' (§A.3.2). While eventually Prolac will have multiple inheritance (i.e., allow multiple parents for a single module), the language and compiler described in this thesis only support single inheritance.

The *arrow operator* '==>' from normalize-sign is the usual way to express conditional execution in Prolac; 'X ==> Y' essentially means 'if X, then Y' (§A.9.4.3).

The call of normalize-sign demonstrates that if a rule has no parameters, parentheses can be omitted when it is called (§A.9.2).

Pos-Denom-Rational adds one new rule, normalize-sign, and *overrides* an old one from Rational's collection, negative (§A.5.2). Consider:

```
module Test has Rational, Pos-Denom-Rational { ...
  field pos-denom :> Pos-Denom-Rational;
  test ::=
    let p :> *Rational in
      p = &pos-denom,
      p->negative  // actually calls Pos-Denom-Rational's negative—
      // even though p's type points to a Rational object—because the
      // run-time type of *p is Pos-Denom-Rational.
    // This is dynamic dispatch.
  end;
}
```

In Test, also notice the let expression, which resembles let expressions in many functional languages (§A.9.5); the different syntax for pointer types (§A.8.6); and the syntax for calling a specific object's version of a rule, which is the same as C++'s.

2.4 Namespaces

Prolac allows the user to create explicit *namespaces*, both at file scope to group modules and within modules to group rules (§A.4). To illustrate, we extend Rational again:

```
module Reduced-Rational :> Pos-Den-Rational {
  constructor(n :> int, d :> int) ::=
    Pos-Den-Rational(n, d), reduce;
  reduce {  // reduce is a namespace
    reduce ::=
      (den == 0  ==>  { assert(0 && "Bad denominator!"); })
      ||| recurse(den);
    recurse(try :> int) ::=
      (try <= 1  ==>  true)
      ||| (num % try == 0  &&  den % try == 0  ==>
            (num /= try, den /= try, recurse(den - 1)))
      ||| recurse(try - 1);
  }
}
```

Here, we avoided polluting the module's top-level namespace with recurse by placing it in a nested namespace, reduce. (This also meant that we could give it a short name, rather than reduce-recurse or some such.) Note that, in the constructor, we treated the namespace name 'reduce' as a rule call; this is legal, and abbreviates calling 'reduce.reduce' (§A.9.2.1).

Other things to notice: recursive rule calls to implement a looping construct; the assert expression in braces { ... }, which is a *C block* specifying C code to be executed (§A.9.6); and the *case bars* '|||', which, together with the arrow operator '==>', express a case expression analogous to Lisp's cond (§A.9.4.7).

2.5 Advanced features

In our final example, we use some relatively advanced Prolac features. First, let's make it harder to change the values of the num and den fields without going through accessor methods:

```
module Rational–Interface :> Reduced–Rational {
 // accessor methods:
 numerator :> int ::= num;
 denominator :> int ::= den;
} hide (num, den);
```

Here, hide is a *module operator,* or an operator which acts on modules instead of values (§A.3.4). Placed in this position, it is an *after-module operator* which affects what users of Rational–Interface will see by default (§A.3.4.2); in particular, it removes the names num and den from Rational–Interface's namespace. (However, in keeping with Prolac's anti-access-control philosophy, num and den can still be accessed, either following a show operator or through Rational–Interface's complete namespace, Rational–Interface.all.) Module operators are Prolac's main linguistic contribution; they are discussed in greater detail in the next chapter.

Unfortunately, implicit rules, which are most useful in larger programs, are very difficult to justify with a microexample. A larger justifying example is therefore provided along with the discussion of implicit rules; see §3.2.2.

3
LANGUAGE DESIGN

This chapter discusses the Prolac language: its design; its linguistic contributions, especially module operators; and how it is used to write protocols, including a discussion of optimizations it supports.

3.1 Design

3.1.1 History

The predecessor to the Prolac language, designed by Frans Kaashoek with input from Eddie Kohler, was inspired by the *yacc* parser generator [Joh75]. The PC language had named rules and an expression-based syntax. Like *yacc*, it was also bilingual, allowing escapes to C code to express some parts of a computation. Prolac has inherited all of these features. However, PC rules could not take parameters or return results, and there were no modules or namespaces; a source file was a flat collection of rules, and all communication between rules was through global variables.

The goals of the protocol compiler project are to implement protocol-specific optimizations, thus creating high-performance protocol implementations; to facilitate protocol extensions; and to make protocol implementations more tractable to human readers. Experience with PC convinced us that while, with some work, we could create a reasonably high-performance protocol implementation using the language, the second and third goals would be essentially impossible to achieve.

Prolac is a completely new design that addresses those problems. Simplistic restrictions from the earlier language were removed: thus, rules can take parameters and return results. Prolac's new object-oriented module system and flexible namespaces address the problems of extensions and tractability; the Prolac programmer splits computation into many small, sensibly-named rules which may be overridden later. Our experience with Prolac has been positive in all three areas: even before extensive tuning, our new TCP specification is about as fast as the old, but also more extensible (the TCP specification we present consists largely of extensions to a small base) and much more readable.

A word on rule terminology: The term “rule”—borrowed from *yacc*—was perfectly appropriate for PC; in Prolac, rules are much more like functions or methods in other programming languages. We find, however, that the unorthodox terminology highlights salient differences between Prolac rules and most other languages' methods. In particular, Prolac rules tend to be smaller than methods; rule bodies are expressions, not statements; and Prolac has no complicated flow-control statements like for or while, so rules tend to be simpler and appear more “functional” than most methods.

3.1.2 Goals and principles

This section describes some goals and principles which guided the design of Prolac. These goals and principles grew out of our experience with Prolac itself and are more language-specific than those described in §1.2.

Encourage short rules. We want Prolac programs to consist of many sensibly-named rules with small bodies; such code naturally lends itself to extension and to quick top-down comprehension. However, such a style can rapidly become unreadable unless rules are given appropriate names (neither too long nor too short) and organized into useful groups. The namespace system was developed to facilitate this.

Eliminate syntax. Appropriate rule naming is necessary, but not sufficient, to make a program with many small rules comprehensible. If there are many small rules, a user will often forget what one does; the language must allow the user to quickly find the rule in question and take in its purpose at a glance. Prolac tries to facilitate this by eliminating nonessential syntax, so that a user doesn't have to wade through syntactic commonplaces to find what a rule really does. This (perhaps religious) point is elaborated further in §3.2.3.

Abbreviate routine code. The language should allow a user to abbreviate or eliminate routine code by implicitly generating it. Namespace call, implicit constructors, and especially implicit rules are examples of how this principle was put into practice. Implicit code generation is limited, and hopefully made less surprising, by a simple restriction: implicit code generation only uses mechanisms also available to the user.

Abbreviations degrade safely. Any implicit code the compiler generates should degrade safely: when the user makes a change that would significantly change the meaning of implicitly generated code, an error or warning should be produced rather than a silent change in meaning. This goal has been only partially achieved; adding a rule to a supertype may silently change the meaning of a (previously) implicit rule in a subtype, for example.

3.2 Contributions

This section describes novel aspects of the Prolac language, especially module operators and implicit rules. A list of smaller contributions, including the rationale behind Prolac's minimal syntax, closes the section.

3.2.1 Module operators

Module operators are Prolac's simple and powerful mechanism for manipulating modules. A module operator is simply an operator which takes a module as a value and returns a module as a result. A module operator expression is acceptable wherever a module can be used in Prolac—as a supertype or import, for example, or as the type of a field. The module operators we have implemented in Prolac affect only a module's extra-type information (a module's namespace, which names are accessible to implicit rule search, and which rules are inlined); however, the module operator concept is fully general. We do not know of another language which provides an equivalent to the module operator concept.

Module operators in Prolac

Prolac currently provides six module operators: hide, show, rename, using, notusing, and inline. The first three control a module's namespace and provide a form of access control; using and notusing control how implicit rules are found; and inline controls which rules are inlined, and by how much. The reference manual describes these operators in more detail (in §A.3.4, to start with); this section provides an introduction.

A module operator expression looks like 'M *operator* arguments', where M is a module expression and *operator* is one of the six operators listed above. The arguments are generally a list of feature names or the all keyword; rename and, optionally, show take name assignments '*newname* = *oldname*' instead of individual names.

Module operators do not interact with Prolac's type system. Thus, if M is a module, M and 'M hide all show f' represent the same type. Module operators are purely compile-time constructs and have no run-time representation.

Module operators are useful both for a module's creator and for its users. Creators can use *after-module operators* to provide a suggested interface to the module; for example:

```
module M {
  ...
  implementation-detail ::= ...;
} hide implementation-detail;
```

Users can apply additional operators in their module header, specifying necessary changes to the module's interface, suggested inlining, and so on.

Individual module operators

The namespace module operators hide, show, and rename control module namespaces. The rename operator is relatively straightforward, and resembles the renaming facilities provided by several other object-oriented languages:

```
module X {
  f ::= ...;
}
module Y {
  f ::= ...;
}
// resolve the multiple inheritance conflict with rename
// (except that Prolac doesn't yet support multiple inheritance)
module Z :> X rename (f–x = f), Y {
  // use f–x for X's f
}
```

Hide makes some of a module's names inaccessible and show makes inaccessible names accessible again. While hide can also be used to resolve multiple-inheritance conflicts, the primary use for hide and show is to provide suggested interfaces to a module. We saw above how hide can be used to hide a module's implementation details; because Prolac intentionally avoids ironclad access control, however, another module can show the hidden rules if it wants. See below for an example of how one might provide real access control with module operators.

Using and notusing provide an interface to Prolac's implicit rule search mechanism. (Implicit rule search is the process in which the Prolac compiler automatically writes forwarding rules for frequently-used method calls; see below.) Using makes some of a module's names available for implicit rule search, while notusing makes them unavailable. Implicit rule search would be so complex as to be unworkable were it not for the user control the module operators provide.

Finally, the inline operator provides control over how a module's rules are inlined. (No notinline operator is necessary because the inline operator takes an optional argument, a "level" between 0 and 10, where 0 means "never inline".)

Discussion

When defining a module, the only way to hide a feature or to suggest that a rule be inlined, etc., is to use an after-module operator. This has the advantage of factoring subsidiary information out of the module definition, thus making the definition itself smaller and cleaner. Unfortunately, this can also be a disadvantage: while some information (i.e., inlining and implicit rule search) seems to belong out of the module definition, other information (i.e., whether or not a rule is in the interface) seems to belong inside it, with the relevant rules. Judicious use of comments and/or namespaces can mitigate this.

It is not always clear which module operators the compiler should use. Consider:

```
module D {
 rule ::= ...;
}
module S has D {
 field d :> *D;
}
module M :> S has D inline all {
 test ::= d->rule;
}
```

The question is, should the call to d->rule be inlined? According to S, which defined d, it should not; but M seems to be implying, with its 'D inline all' declaration, that it should. The current *prolacc* compiler will inline d->rule in this example. In effect, when the compiler refers to a field of module type or calls a rule returning a module, it checks the current module to see if it should use a new set of module operators for that module type. This algorithm has proved adequate for our current purposes.

One way of thinking about module operators is that they provide a very clean way to give information to the compiler (e.g., using, notusing, and inline).

Hypothetical operators: Access control

Module operators were originally invented as a clean way to provide access control after C++-style access declarations were rejected as too obtrusive and inflexible. The Prolac module operator solution is much more powerful; for example, a module can export multiple interfaces. Say we want to provide two interfaces for M, one containing only f1 and the other only f2. This is simple:

```
module M {
 f1 ::= ...;
 f2 ::= ...;
}
module Interface-1 ::= M hide f2;
module Interface-2 ::= M hide f1;
```

Of course, in keeping with Prolac's anti-access-control philosophy, Interface-1 and Interface-2 do not prevent hidden methods from being accessed; they are essentially suggestions.

The module operator concept is easily flexible enough to handle true access control, however, and it is useful to consider how true access control might be provided using module operators. One solution would be to introduce a private operator, which would leave a feature in the namespace but mark its use as an access control violation. To illustrate:

```
module M {
  ...
}
module Interface–1 ::= M private f2;
module Interface–2 ::= M private f1;
```

Unfortunately, a module could circumvent private simply by using M directly. A solution for this problem involves two new module operators, allow and disallow; a module can refer to another module only if it has been explicitly allowed. Say that module Alice should only use Interface–1, and Bob should only use Interface–2. This could be coded as follows:

```
module M {
  ...
} disallow all allow (Interface–1, Interface–2);
module Interface–1 ::= M private f2 allow Alice;
module Interface–2 ::= M private f1 allow Bob;
```

Other hypothetical operators

This section lists some other module operators we have considered.

A warn module operator would provide a flexible, user-defined method to implement both generalized access control and compile-time usage hints. The warn operator would tell the compiler to generate a particular warning or error when a feature is used; for example:

```
module M {
  armageddon ::= ...;
} warn (armageddon = “Are you sure you should be causing armageddon?”);
```

Extensions to warn could cause warnings on overriding a rule, on referring to a class, on declaring a field, etc. This provides a generalization of obsolete routines in Eiffel [Mey92].

All module operators we’ve described only affect a module’s extra-type information. By contrast, a redefine operator affects only a module’s type. Redefine could change any type into a type generator by allowing its parents or imports to be redefined. A redefine operator has been planned for some time; it remains unimplemented for reasons we describe below. To demonstrate its usage, assume we have a Prolac specification for two related modules, A and B:

```
module A { ... }
module B has A {
  field f :> *A;
  ...
}
```

Now say we want to create an implementation of this specification. The first step is natural enough:

```
module A–Impl :> A {
  // overrides A's rules and provides some new ones
}
module B–Impl :> B has A–Impl {
  // initialize f with a pointer to A–Impl
}
```

However, we're in trouble if we want to refer to some rules specific to A–Impl from the B–Impl module; the field we have points to an A, not an A–Impl, so we have to cast. As a further affront, *prolacc* will not be able to inline as many rule calls on f without performing complicated data flow analysis, since it can only assume f is an A and not the more specialized A–Impl.

Here is a potential solution to this problem using redefinition:

```
module B–Impl :> B redefine (A ==> A–Impl) {
  // can refer to A–Impl rules from f
}
```

While this solution is attractive in that it literally embodies what the programmer wants to do, it is difficult to implement. In particular, the compiler must re-check all of B's rule bodies after the redefine; assigning a pointer to an A object to the f field must cause a compile-time type error in the redefined version! Redefine also becomes clumsier to use in larger examples.

Currently, our TCP specification addresses this problem using module equations and namespaces. Here is a simplified example:

```
Base {   // the base protocol
  module X–Tcb { ... }
  module X–Code has Tcb {
    // Note: use Tcb, the TCB we've chosen globally, not X–Tcb!
    ...
  }
}
Delay–Ack {   // extended for delayed acknowledgements
  // which version are we extending?
  module Parent–Tcb ::= Base.X–Tcb;
  module Parent–Code ::= Base.X–Code;
  module X–Tcb :> Parent–Tcb { ...add fields... }
  module X–Code :> Parent–Code has Tcb { ...refer to new fields... }
}
// which version are we using globally?
module Tcb ::= Delay–Ack.X–Tcb;
module Code ::= Delay–Ack.X–Code;
export Tcb.all, ...;
```

If we globally choose a TCB which is Delay–Ack's or a subclass of it, all is well; if we choose a different TCB, say Base.X–Tcb, the compiler will generate errors for the references to Delay–Ack fields in Delay–Ack.X–Code—but that code should not be used in this situation anyway.

Redefine is not the only possible module operator that would affect a module's type. In particular, note that instantiation of templates or parameterized types can be considered a specialized form of module operator!

Related work

Eiffel [Mey92] provided the initial inspiration for module operators; it allows a subtype to rename, undefine, and redefine its supertypes' features through clauses in its parent list. Many other object-oriented languages offer these operations, or a subset. Only supertypes can be so manipulated, however, and the manipulations are restricted to those useful during inheritance. Of Eiffel's transformations, Prolac implements only rename as a module operator; without a concept of "deferred feature", undefinition is meaningless, and redefinition—overriding—requires no special syntax.

3.2.2 Implicit rules

Many object-oriented languages make it easier to refer to features in the current class by allowing the programmer to elide the class or object name. Prolac goes further: it allows the programmer to elide *other* class or object names through the flexible mechanism of implicit rule search. The following motivating example is copied with modifications from the Prolac reference manual, §A.5.4.

Consider a module Segment–Arrives implementing part of the TCP protocol. This module will frequently refer to the current transmission control block, tcb, which has type *Tcb; we make tcb a field so we don't have to constantly pass it as a parameter. This code uses Tcb's listen, syn–sent, etc. rules to determine the current protocol state:

```
// Example 1
module Segment–Arrives has Tcb {
  field tcb :> *Tcb;
  check–segment ::=
    (tcb->listen ==> do-listen)
    || (tcb->syn-sent ==> do-syn-sent)
    || (tcb->syn-received ==> do-syn-received)
    || (tcb->established ==> do-established)
    ...;  // and much more!
}
```

The repetition of 'tcb->' is tedious and hinders quick comprehension of the code. We know Segment-Arrives deals with only one tcb; why should we have to tell the compiler which tcb we mean again and again?

One solution is to generate forwarding rules in Segment-Arrives. We hide these forwarding rules using after-module operators (§A.3.4.2), since they are artifacts of the implementation.

```
// Example 2
module Segment-Arrives has Tcb {
  field tcb :> *Tcb;
  check-segment ::=
    (listen ==> do-listen)
    || (syn-sent ==> do-syn-sent)
    ...;
  listen ::= tcb->listen;
  syn-sent ::= tcb->syn-sent;
  ...
} hide (listen, syn-sent, ...);
```

This is better; however, the forwarding rules clutter the module definition and, again, are tedious and error-prone to write.

The solution in Prolac is to use implicit rules. We use the using module operator (§A.5.4.4) to open tcb for implicit rule search. When the compiler creates Segment-Arrives's complete namespace, it searches its rules for undefined names, entering them in Segment-Arrives's top-level namespace as undefined implicit rules. Later, it creates their definitions through a search process. It marks the implicit rules as highly inlineable and hides them in the default namespace. Thus, the compiler transforms the following code into something like Example 2:

```
// Example 3
module Segment-Arrives has Tcb {
  field tcb :> *Tcb using all;
  check-segment ::=
    (listen ==> do-listen)
    || (syn-sent ==> do-syn-sent)
    ...;
}
```

Example 3 is, in a sense, optimal: nothing distracts the reader from exactly what the module is doing.

This example has demonstrated that implicit rules can make code more readable rather than less. Overuse of implicit rules can make code very difficult to understand, however; moderation is required, as with any powerful tool. The compiler will generate warnings rather than implement arbitrary choices when implicit rule use gets ambiguous and complex.

Note that only rules can be found implicitly: referring to a field requires explicit syntax.

Discussion

The implicit rule mechanism was inspired by Prolac's intended domain, protocols. We noted, after examining several protocol specifications, that protocols tend to handle a relatively small number of data objects, each of distinct type—a control block and a packet, for example. Unfortunately, arranging protocol code into hierarchies based on those data objects feels unnatural and can result in convoluted code. We wanted the syntactic convenience object-oriented languages provide without forcing programmers to organize their code strictly according to those objects.

Implicit rules are most useful in similar situations, when a large part of a program deals with one object of a given type at a time: if there is more than one object, which one should be used to define the implicit rules? Of course, if there are multiple objects of a given type, the using and notusing module operators give the programmer enough control over implicit rule search to allow any desired result.

Prolac allows implicit rules to be found in imports (for static rules); in fields with module type; in fields with pointer-to-module type; and in supertypes' imports and fields. The reference manual, in §A.5.4, describes the search process which finds the relevant definition for an implicit rule.

The implicit rule mechanism is a good example of how our limitations on implicit code generation (discussed above) work in practice. The compiler generates forwarding rules—rather than, say, changing rule bodies to refer to the relevant rules directly—because the user could write forwarding rules explicitly within the language.

Related work

Implicit rules are similar to package- or namespace-importing directives in languages like C++, Java and Modula, and especially the open directive in ML. All these directives affect functions (and classes and values), not methods, however: none of them can make an object implicit in a method call. The implementation of implicit rules as forwarding rules generated by the compiler—thus allowing subtypes to override an implicit rule—also appears to be new.

3.2.3 Other contributions

Some smaller novel Prolac features include the use of namespaces within modules to organize rules; namespace call (§A.9.2.1), which facilitates working with namespaces within modules by making them act somewhat like functions with local function definitions; and the use of inline and outline levels for flexible control over optimization.

A more subjective contribution—which we nonetheless feel is quite important—is Prolac's minimal syntax. The remainder of this section explains the reasoning behind Prolac's syntax and provides some commentary.

Syntax does matter. A mathematical concept is almost impossible to work with without decent notation to describe it; the better the notation, the easier it is to work with the concept. Prolac was designed with careful attention to syntax, the overriding goal being to make the syntax as minimal as possible without sacrificing readability.

Consider Prolac rule syntax for an illustration of the result. No keyword introduces a rule definition; the rule's name is the first thing in the definition; rules returning bool—the most common return type in our protocol definitions—may elide the return type; parentheses may be elided when calling a rule that takes no arguments. Because Prolac is an expression language, a rule's body is simply an expression representing its result.

Taken together, this means that Prolac rule definitions are very small and easy to both write and read. This removes any subconscious pressure on the programmer to keep rule bodies large, since small rules actually look like small objects. (As a programmer, I often fell victim to a visceral reaction that could be verbalized as "I'm writing all this text to make a function that does *this?!* Forget it!") Prolac syntax actually becomes less readable as rule bodies get large; while this concerns us, it also provides a convenient back-pressure to keep rules small.

A comparison with other object-oriented languages may provide some perspective. Here are six functions which return true iff a TCP connection is in the closed state; in each case, the code implements a "method" of the hypothetical TCB "class":

```
closed ::= state == 0;  // Prolac
bool closed() { return state == 0; }  // C++, Java
fun closed({state,...}: TCB) = state=0;  (* ML *)
closed: BOOLEAN is do Result := state = 0 end  -- Eiffel
closed() returns (bool) return (state = 0) end closed  % Theta
(defmethod tcb-closed ((t TCB)) (= (tcb-state t) 0))  ; CLOS
```

Obviously, Prolac's syntax is the easiest to write,[1] but because the function being defined is so small, Prolac's syntax is the easiest to read and understand as well. For small functions, *conciseness is readability.* This example highlights Prolac's relatively

[1] Compared with some languages, like C++, the difference is admittedly minimal; however, if we repeated the example 11 times for TCP's 11 states—and then hundreds more times throughout the program—even that 12-character difference would begin to become significant.

unusual focus on small functions: most object-oriented languages are focused on helping the programmer write large functions, rather than helping her divide a large function into small ones. A similar example using a larger function would no doubt show that Prolac's syntax became the hardest to read.

We feel that, in combination with namespaces and other facilities for naming rules appropriately, Prolac's minimal syntax has succeeded: our programs are very readable, and we naturally divide computation into small rules.

3.3 Usage

3.3.1 Prolac and protocols

With exactly one exception (the seqint type, representing sequence numbers; comparison on seqints is circular §A.9.11.4), Prolac contains no primitive data or control structures targeted at network protocols. As mentioned above (§1.2), this is because of the dangers of inflexible abstractions: it would be impossible to design a "connection" abstraction serving all protocols and situations equally well, and we did not want programmers to have to work against the language. We note, however, that Prolac would be a very good language in which to implement a *library* of domain-specific abstractions. A library is the right place for such code: a user can use it if it is helpful, but can work around it or change it when it is not. (We cannot expect a user to change the compiler.)

Prolac was nevertheless specifically designed for protocol implementation. We list a number of Prolac features below and describe how they were inspired by, and how they help implement, network protocols.

Single source file. Protocol implementations are often not particularly large, just complex. The Prolac compiler takes advantage of this by compiling the entire source at once. This allows inlining across the entire program and global analysis for optimization. For example, with the entire source on hand, the compiler can easily see if a rule is ever overridden; if it is not, dynamic dispatch can be avoided. This optimization is actually a very important one, since it allows the vast majority of rules to be inlined.

Implicit rules. As discussed above, examination of several protocol specifications directly led to the implicit rule mechanism.

No access control. Inspection of several TCP extensions, like slow start and fast retransmit [Ste97], was what convinced us that protocol extensions do not affect code identifiable ahead of time. Again, they tend to change code that the protocol designer probably had not identified as meriting future extension. This led us to decide that hard access control was inappropriate for Prolac.

We are now reconsidering this decision. The intuition was right, as far as it went, but we did not fully consider the documentary value of access control declarations. It is actually helpful to a module's creator and to its users to have interface functions clearly defined. While the hide and show module operators allow module creators to define an interface, there may be too many ways in Prolac for a user to get around them. A tool that reads a Prolac specification and outputs a reduced version of a module, with only interface functions actually visible, would be helpful as well.

3.3.2 Optimization

Prolac has direct linguistic support for two optimizations, *inlining* and *outlining.* Inlining is directly including a function's entire body at a call site. Outlining is when code for an infrequently executed branch is moved to the end of the function body; this will improve a program's i-cache and instruction pipeline behavior when the common path is taken.

These optimizations are particularly important for network protocols [MPBO96]. Protocol code is too important to suffer frequent function call overhead and loss of intraprocedural optimization opportunities; Prolac's focus on many small rules makes inlining even more important. Secondly, much protocol code contains a lot of error-handling code (or, more generally, infrequent-situation-handling code) along the critical path. Moving this code out of the way, and thus allowing the common case to be executed in a straight line, is another important optimization.

In Prolac, inlining and outlining are specified with two operators, inline and outline; there is also an inline module operator which allows a programmer to say, for example, "inline all rules from this module".[2] Both of these optimizations take an optional *level argument,* which must evaluate to an integer between zero and 10. Zero means "do not optimize at all", while 10 means "optimize as much as possible"; for example, inline[10] means "always inline this function call", while outline[0] means "this code is the most common path". Intermediate levels can be interpreted appropriately by the compiler. *Prolacc*, for example, takes a command-line argument which specifies which inline levels should actually result in inlined code.

Prolac therefore supports two of the three optimization techniques discussed in [MPBO96]. Outlining we have already described; cloning, or making more than one copy of a function body, is not supported. *Path inlining* inlines a function call, as well as any functions *it* calls, and so on, recursively; the *prolacc* compiler simply treats a very high inline level—9 or 10—as a suggestion to perform this recursive inlining.

Automated optimization

Note that the optimization techniques supported in Prolac are entirely manually specified. We have carefully designed Prolac to make this manual specification as painless as possible (consider particularly the inline module operator), but an automated system

[2] An outline module operator would be slightly less useful, but perhaps still worthwhile; for example, a module creator could specify that a rule will practically never be called, so any code path which calls that rule should be outlined.

for collecting statistics at run-time and applying them to the specification might be more painless still.

We plan to eventually implement automated statistics collection, but not to automatically apply this information to the compilation process. The user will have to analyze the statistics and modify the Prolac source accordingly. We believe that the source code should completely determine the run-time behavior of a Prolac program, and are convinced that the programmer will be better at specifying optimizations[3] than any automated tool could be. For example, if profiling reveals that a certain path is frequently executed, the programmer can rewrite the source to improve that path's behavior, while an automated tool could only inline and outline code more aggressively along that path. Automation does keep system-specific optimizations out of the normal source code, but if this is desired, subtyping and module operators can do the same thing even more flexibly. We feel that if an optimization is difficult to specify as an annotation on the source, this is a failure of the Prolac language, not an invitation to automate the optimization.

Castelluccia [Cas96] describes a system for automating header prediction in a compiler for the Esterel language. Basically, the system outlines all cases except the common path which header prediction would take. This would be possible in Prolac; in our TCP specification, however, we implement header prediction the old-fashioned way—as additional code executing before normal processing begins. While this suffers some of the problems Castelluccia describes (specifically, code duplication in the generated C code), complexity is not one of them: header prediction requires only 33 lines of code, which is factored out of the base protocol by subtyping. Automating this does not seem immediately worthwhile. Castelluccia also performs header prediction across layered protocols; we expect subtyping and other Prolac mechanisms to continue to be effective there.

[3] Meaning the protocol-specific optimizations we have been discussing.

4

THE prolacc COMPILER

This section gives an overview of *prolacc*, the Prolac compiler implemented for this thesis. After a general description of its structure, we present three of its more interesting subsystems: the representations of modules and namespaces; the name resolution process, which creates the intermediate representation (IR); and the back end, which compiles that IR to efficient C.

4.1 Overview

The *prolacc* compiler consists of about 16,000 lines of object-oriented C++ source code, not counting comments. It has been tested on several different Unixes and should be very portable, even to other operating systems; to enhance portability, it uses only a subset of the C++ language and does not use any C++ library.

The compiler executes in three stages. The first, *parsing,* translates the source text to parse trees without looking up any identifiers. The second stage, *resolution,* reports semantic errors, looks up identifiers, and translates the parse trees to an intermediate representation; the third stage, *optimization and compilation,* translates that IR to the output language, C. Each of these stages runs to completion before the next is begun.

Difficult issues in compiling Prolac include supporting its order independence; efficiently handling often-manipulated structures like modules and namespaces; and generating efficient, high-level C.

This chapter will often discuss C++ classes forming a part of the compiler; when mentioned by name, these are written in *italics.* Table 4.1, on page 34, provides an overview of the more important classes we discuss.

Parsing

First, the source file is lexically analyzed and divided into tokens. Parser feedback is necessary to read C blocks and support C code, which follow different parsing rules; the contents of C blocks and support C code are treated simply as arbitrary text.

Class	**Generated by**	**Description**
Yuck	*main()*	Recursive descent parser
Namespace	*Yuck*, etc.	A Prolac namespace
Prototype	*Yuck*	A Prolac module, module equation, or module operator expression before resolution
Protomodule	*Yuck*	Subclass of *Prototype*. An actual Prolac module, not an equation or module operator expression; stores *Expr*s for the module's features
Expr	*Yuck*	Parse tree
Module	*Protomodule*	Type- and rule-related portion of a Prolac module
ModuleNames	*Protomodule*	Namespace- and module operator-related portion of a Prolac module
Node	*Expr*	Tree-structured intermediate representation
Target	*Node*	*Node*s plus evaluation order
Location	*Target*	*Node*s arranged in basic blocks

Table 4.1: Important *prolacc* classes

The tokens are passed off to *Yuck*, a recursive-descent parser.[1] The *Yuck* class builds three important sets of structures: *Namespace*s, including module namespaces as well as file-level and nested namespaces; *Prototype*s, which represent modules before they are resolved; and *Expr*s, which implement Prolac's parse tree.

Parsing is complicated by Prolac's order independence: the parser does not know which names are types, for example, and subtypes may appear in the file before their supertypes. Therefore, *prolacc* performs no analysis on the structures it creates, except to complain if a name is multiply defined in a single namespace; specifically, it does not resolve identifiers.

After parsing is complete, the compilation is in this state:

- The file-level namespace and its subnamespaces are *prepared,* meaning that they map identifiers to the right features and that they will not be changed later. *Prototype*s that represent the modules defined in the Prolac program are stored in these namespaces.

- Each module's internal namespace is prepared. This namespace contains any rules, fields, and nested namespaces defined by the module.

[1] *Prolacc* does not use a parser generator—specifically, *yacc*—both because *yacc* interacts badly with C++, and because we prefer recursive descent parsers anyway. Recursive descent parsers can provide better error recovery and better error messages, as well as being able to parse more languages. Also, our experience with parser generators for larger languages has been generally negative: they are too inflexible, too demanding, and tend to force an unnatural structure on the rest of the parser. We are not alone in this preference; see [Str94, pp68–9].

- A complete *Prototype* exists for each module and module equation. *Protomodules*, which are *Prototypes* for actual module definitions, store parse trees for each of a module's features.

Resolution

Once the entire source has been parsed, *prolacc* transforms the objects created by the parser (*Prototypes* and *Exprs*) into the corresponding objects in the intermediate representation (*Modules* and *Nodes*). The goal of resolution is twofold: first, to prepare a *Module* class for every Prolac module M, which involves merging M's parents with M's internally defined features; and second, to use the complete namespaces so generated to resolve rule bodies and implicit rules, producing the *Node* intermediate representation.

The *Node* intermediate representation is tree-based. Later stages of the compiler—optimization, for example—generally treat *Nodes* functionally: the compiler creates new *Node* trees (which may share partial state with existing trees) rather than modifying ones that already exist.

The resolution process has three substages, which are described in more detail in §4.3.

After resolution is complete, the compilation is in this state:

- Each module's complete namespace (§A.4.4) is prepared. This contains its parents' namespaces as well as its internal namespace and its implicit rules.

- Each module is prepared: all of its ancestors are known; the compiler knows which of its rules are overrides (and what they are overrides of) and which ones originate in the module itself; and all of its fields, including those defined in ancestors, have been collected.

- Every declared type—e.g., the types of fields, parameters, return types, and let-bound variables—is known.

- Each rule is prepared: its body expression has been translated into unoptimized IR; its signature has been checked against the rule it overrides, if there is such a rule; and the compiler knows whether or not it has ever been overridden.

- All user errors have been reported.

Prototypes and *Exprs* can be thrown away once resolution has completed.

Optimization and compilation

The final stage is code generation, which creates a C header file (containing structure definitions for Prolac structures) and a C source file (containing C code implementing exported rules as well as any rules they call). The *Node* IR is optimized before generation by passing it through a *NodeOptimizer*; there are several kinds of *NodeOptimizer*, corresponding to different optimizations and to several code transformations necessary for generating correct C. A *NodeOptimizer* class, assisted by the *Node* hierarchy, makes a bottom-up pass over a *Node* tree while simultaneously creating a new, optimized result tree.

During compilation, *prolacc* arranges a *Node* tree into a traditional basic block representation. This process is described in more detail in §4.4.

4.2 Representation issues

4.2.1 Modules

The internal representation of Prolac modules is complicated by the requirement for reasonably efficient module operator support. The solution adopted for *prolacc* has been to split a module into two parts. The first, *Module*, contains type- and rule-related information; for example, a module's ancestors, fields, and rules are all stored in the *Module*. In the compiler's class hierarchy, *Module*s stand alone: *Module* is not a subclass of *Type*, for example.

A *ModuleNames*, in contrast, contains a module's namespace, as well as any inline and using operators the user has applied. Each *ModuleNames* object corresponds to a *Module* object; there can be many *ModuleNames* for each *Module*. A new *ModuleNames* is created, also during the resolution phase, each time a collection of module operators is applied to a module. The *ModuleNames* class is a subclass of *Type*; different *ModuleNames* with the same *Module* are equivalent to the type system except for feature lookup. For example:

```
module M {
  field f :> int;
  test ::=
    let m :> M, m–hidden :> M hide f in
      m = m–hidden,  // OK
      m–hidden = m,  // OK
      m.f,  // OK
      m–hidden.f   // error: no such feature
    end;
}
```

Here, the two assignment statements are legal because assignment depends only on type: ‘M’ and ‘M hide f’ are, again, equivalent to the type system except for feature lookup. The two member expressions demonstrate how feature lookup differs between ‘M’ and ‘M hide f’.

A third class, *Protomodule*, represents modules before the resolution stage; where *ModuleNames* and *Module*s contain features, *Protomodule*s contain parse trees for those features. The parsing stage creates *Protomodule*s, and the resolution stage—largely controlled by the *Protomodule* class—builds *ModuleNames* and *Module*s from those *Protomodule*s.

4.2.2 Namespaces

A Prolac compiler will expend a lot of effort manipulating namespaces—for example, see the namespace resolution algorithm described below. This requires an efficient representation for namespaces. *Prolacc*, therefore, allows namespaces to be shared when possible, even potentially between different modules. In the following example, *prolacc* will only create nested-namespace once, since it can be shared between modules M and N:

```
module M {
 nested-namespace {
  ... many definitions ...
 }
}
module N :> M {
}
```

This is facilitated by making each namespace an indirection. Most of the compiler deals only with *Namespace* objects; *Namespace* is largely a forwarding class that sends messages to the *ConcreteNamespace* class. *ConcreteNamespace*s are reference-counted and, when necessary, a *Namespace* is changed to point to a unique copy of its *ConcreteNamespace*. This needs to happen here, for example:

```
module M {
 nested-namespace { ... }
}
module N :> M {
 nested-namespace { ... }
}
```

After its parents' namespaces are merged, but before its internal definitions are added, N will be sharing M's nested-namespace; when its internal definitions are added, N's version of nested-namespace will be changed to a unique copy.

4.3 Resolution

The problem of resolution boils down to resolving names. How and when does *prolacc* decide what a particular name means in a particular situation? Prolac's order independence partially determines the method we choose. The goal is simple: The Prolac language was designed so that a human being can unambiguously discover what a name refers to; the *prolacc* compiler should unambiguously discover the same result.

4.3.1 Overview

Prolac disallows only impossible circular dependencies—that is, circular dependencies in subtype relations, module equations, and object composition. Thus:

```
module M :> N { ... }  // error: circular dependency
module N :> M { ... }
module X ::= Y  // error: circular dependency
module Y ::= X
```

Circular dependencies through imports are allowed:

```
module M has N { ... }  // OK
module N has M { ... }
```

As of this writing, *prolacc* does not detect circular dependencies of object composition; however, the C compiler does:

```
module M has N { field f :> N; ... }
module N has M { field f :> M; ... }
// OK to Prolac, C will complain
```

The resolution process works one module at a time. Each of the three stages is run to completion on all modules defined in the program before the next stage is begun. The stages can be characterized as follows:

1. **Namespace.** Resolve parents and create the completed namespace (§A.4.4).

2. **Types.** Resolve imports and feature types.

3. **Rules.** Find all implicit rules; check that overrides are legal; resolve each rule's body.

Again, resolution is mostly directed by methods in the *Protomodule* class.

4.3.2 First stage: Namespace resolution

After the first stage of the resolution process has completed on a module M, the following invariants are true of M:

- M's complete namespace has been prepared. It will not be changed in any later stage of the process.

- All of M's rules have been identified, either as overrides or not. Note that this includes any implicit rules, although definitions for implicit rules are not found until the third stage.

- For each rule defined in M or any of its supertypes, *prolacc* knows which rule definition is effective in M.

- All of M's slots and imports, including slots and imports inherited from M's supertypes, have been created; however, their types have not yet been resolved.

To complete the first stage on a *Protomodule* M, the compiler first looks up the names of M's parents and imports, simply to check that they are actually modules (or, to be precise, *Prototype*s). This can never result in a circular dependency.

The process begins in earnest when it executes the first stage on each parent P. An error is reported here on a circular dependency.

Once M's parents satisfy the invariants above, *prolacc* creates the actual *Module* A and its corresponding *ModuleNames*. First, it merges all parents' supertypes, rules, imports, and fields into A;[2] then, it adds to A new features originating in M.

M's completed namespace is created using the algorithm described in the reference manual (§A.4.4). This algorithm, which also finds all of M's implicit rules, will not be repeated here. Rule overrides are discovered during this process; when a rule RI defined by M has the same name as a rule RP defined by some supertype, RI is marked as an override of RP.

Note that M's imports are not resolved whatsoever by this first stage.

4.3.3 Second stage: Type resolution

After the second stage of the resolution process has completed on a module M, the following invariants are true of M in addition to those listed above for the first stage:

- All of M's types have been resolved. This includes the type of each field and the signature of each explicit rule.

To complete the second stage on a *Protomodule* M, *prolacc* first executes the second stage on every parent P and the *first* stage on every import I. This ensures that I's namespace- and type-related information is available. The compiler then goes through each field and explicit rule and resolves the simple *Expr*s which represent their types and signatures.

[2] Again, the current Prolac language definition only supports single inheritance, so there can be no conflict here.

4.3.4 Third stage: Rule resolution

After the third stage of the resolution process has completed on a module M, the following invariants are true of M in addition to those listed above for the second stage:

- Definitions have been found for all of M's implicit rules (except those which had no definition). Implicit rules have the correct types, subject to the caveat described below (§4.3.5).
- Overriding rules defined in M have been checked for correctness by the usual contravariance rule, which is described in §A.5.2.1.
- The body of each rule has been translated into the intermediate representation, *Node*s.
- All user errors have been reported.

To complete the third stage on a *Protomodule* M, *prolacc* first executes the third stage on every parent P and the second stage on every import I. The implicit rule search process described in the reference manual (§A.5.4.1) is used to find definitions for implicit rules. The contravariance check is straightforward; it is left until the third stage in case an implicit rule is overridden—an implicit rule's signature is not known until its definition is found. The final step, transforming the *Expr* parse trees for rule bodies into *Node* intermediate representations, is also more or less straightforward, now that all signatures, types, and namespaces have been resolved by this and earlier stages.

In future, we will solve the problems described below by performing the third stage one rule at a time, rather than one *Protomodule* at a time.

4.3.5 Circularity issues

The current *prolacc* compiler meets the goal of resolving all unambiguous names with only two exceptions. The more important of these is *symbolic constants,* used in Prolac inline and outline levels. Rather than introduce a new syntax, we would like symbolic constants to be calls of static rules with constant bodies:

```
module Constants {
 static inline–Bob :> int ::= 9;
}
module Bob.M {
 ...
} inline[Constants.inline–Bob] all;
```

This may seem simple enough; however, consider this example:

```
module A has B {
  static level :> int ::= B.level;
}
module B has A {
  f ::= inline[A.level] ...;
  static level :> int ::= 9;
}
```

The call of A.level is unambiguous to a human, who can easily see that A.level eventually evaluates to 9. However, implementing this correctly would require a more complicated strategy than the one described above. Again, resolution currently works one module at a time; it is an error if a circular dependency is found between modules. The example does have a circular dependency on modules, because resolving the body of rule B.f requires that A.level already be resolved (so we can find the inline level); however, A.level similarly depends on B.level—that is, B depends on A depends on B.

The second problem—much less important in practice—boils down to the same issue: circularity at the module level when no circularity exists at the rule level. This problem involves complicated and unlikely manipulations of implicit rules. For example:

```
module M has N {
  field f1 :> N using all;
  rule1 ::= find-me;
} show (renamed-find-me = find-me);
module N has M {
  field f2 :> M using all;
  rule2 ::= renamed-find-me;
  find-me ::= true;
}
```

First, consider module M. Its rule1 refers to an undefined name, find-me; *prolacc* will therefore define find-me as an implicit rule, and eventually resolve it to f1.find-me—the find-me rule from module N. Now, consider module N. Here, rule2 also references an undefined name, renamed-find-me; this also becomes an implicit rule. This implicit rule is eventually resolved to f2.renamed-find-me—but, due to the show operator on module M, it turns out that this is M's implicit rule find-me!

The circularity comes when we try to determine the signatures of the implicit rules. To find the signature of N.renamed-find-me, we need the signature of M.find-me; but to find the signature of M.find-me, we need the signature of N.find-me—N depends on M depends on N.

The obvious solution is to work one rule at a time rather than one module at a time during the third stage of the resolution process. Because the problem's scope is so limited, however, we have not yet implemented this.

4.4 Compilation

The compilation process, which translates the *Node* intermediate representation to C, is complicated by several issues:

- We would like to keep the generated C code as high-level as possible. In particular, we do not want an expression like 'x + y * z' to compile down to a straightforward three-address-code version in C:

  ```
  int __T__1;
  int __T__2;
  ...
  __T__1 = y * z;
  __T__2 = x + __T__1;
  /* code uses __T__2 */
  ```

  The main reason for this is to keep the generated C code readable, and therefore debuggable. A secondary reason is that we do not trust C compilers to always optimize such explicit temporaries away; while some preliminary tests supported our caution, we do not feel that enough tests were completed to come to any definite conclusion.

- Some Prolac constructs which are part of a Prolac expression must generate C code which is syntactically a statement; the most prominent example is a C block, but control-flow operators generally compile to *if* statements. Thus, the Prolac compiler must occasionally introduce temporaries. This is required when a single expression is broken up by statements, and also when a rule call is inlined: direct substitution obviously won't work when inlining a rule call like f(x++).

- We want *prolacc*-generated C code for a Prolac program to be potentially as efficient as a version of the program hand-written in C. Several optimizations address this issue. In particular, simple global analysis is done on rules, turning many dynamic dispatches into static ones, and some structure assignments are optimized away.

4.4.1 Nodes and compilation

Each *Node* object can generate two kind of code:

- **State code.** State code must be executed exactly once, and is syntactically a statement. A *Node* representing a C block generates state code, for example. Most nodes do not generate any state code.

- **Value code.** Value code may be executed multiple times, and is syntactically an expression.

The method which generates a *Node*'s value code will often call its subexpressions' value code generation methods. For example, the *Node* representing 'A + B', where A and B are expressions, will generate the value code '*(va)* + *(vb)*', where *va* and *vb* are A and B's value code, respectively. This procedure allows *prolacc* to generate reasonably high-level output C code by copying input Prolac expressions when possible.

If a *Node* tree were translated without optimization, any *Node*'s value code will be generated exactly once. However, optimization—particularly inlining, where a parameter value is substituted for the parameter as many times as necessary—can introduce *Node*s whose value code must be generated multiple times. This is perfectly safe for some *Node*s: integer literals, for example, and variables. For others, however (assignments, increments, rule calls), this is an invalid transformation. To address this, each *Node* can return a *Node* which is its corresponding *simple value*—a value which is safe to reuse—or 0 if it has no simple value. For example, an integer literal's simple value is itself; an assignment 'x = f(94)' has the simple value 'x'; and a rule call 'f(94)' has no simple value. Any code transformation which can cause a *Node* to be used multiple times will ensure that the *Node* is a simple value by creating a new temporary if necessary.

Using simple values, rather than, say, always generating a temporary, allows *prolacc* to avoid many spurious structure assignments. This was an important optimization: *gcc*, the optimizing C compiler we used, does not optimize away spurious structure assignments through copy propagation.

Any *Node* can be *temporarized,* meaning that *prolacc* will generate a new temporary to hold its value. A temporarized *Node* will always generate state code; this has the form '*temporary_variable = (the expression's real value);*'. A temporarized *Node*'s value code is just the name of the temporary variable.

4.4.2 Targets

The *Target* class is the first step the compiler takes on the way to a basic-block representation of a *Node* tree. *Target*, like *Node*, is tree-structured, but with only two-way branching. A *Target* tree represents the evaluation order of a *Node* tree; thus, a *Node* tree representing the simple addition 'x + y' will compile to three *Target*s:

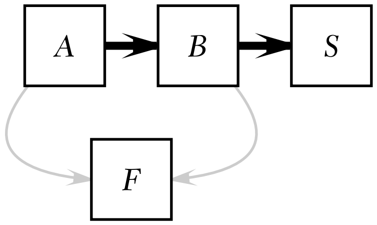
calculate x → calculate y → add x and y.

Each *Target* points back to the *Node* whose computation it represents. The second branch of a *Target* is only used at when the execution path can fork, such as for most control flow operators.

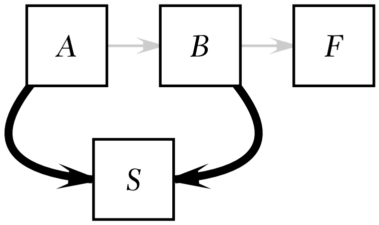
*Target*s are then used to decide whether an execution point must start a basic block. A *Target* starts a new basic block if it reachable through more than one path; it is reachable through one branch of a forking *Target*; or it represents a new occurrence of the outline operator. The next stage makes these basic blocks explicit: the *Location* class and its subclasses, particularly *BlockLocation*, represent basic blocks. Once *Location*s are generated, the *Target*s can be thrown away.

4.4.3 Locations and C generation

The compiler then makes several passes over the generated *Location*s to arrange them in output order. The first pass creates C && and || expressions, where possible, to enhance code readability—and, hopefully, object code performance—relative to nested *if*s; the compiler looks for the following patterns, generating && and || expressions as indicated.



if (A && B) { S; goto L; } F; L:



if (A || B) { S; goto L; } F; L:

(Any basic block with two branches has, as its final action, a test to see if it some expression is zero: one branch is taken on zero, the other if not. In the figure, thick black arrows represent the nonzero branch and thin grey arrows represent the zero branch.) *Prolacc* will apply either transformation only if *B* cannot generate any state code and *B* has not been outlined relative to *A*.

Prolacc then decides which branches will be placed in line and which will be implemented as *goto*s, using any outline information provided by the user.

The penultimate step is assigning temporaries, including any temporaries required because an expression's definition in one basic block has been separated from its use in another. All temporaries are gathered and emitted in one block of declarations at the beginning of the resulting function body.

Finally, the compiler generates C code for all of the basic blocks in the order it previously found. Once the C code is generated, all *Location*s can be freed, as well as the optimized *Node* tree.

4.5 Evaluation and future work

The *prolacc* compiler is stable on correct programs. Unfortunately, some errors can cause a crash; improving error recovery is high on our list of priorities. Other directions for future work include:

- Implementing multiple inheritance.
- More carefully analyzing the C compiler's behavior on our C output. We would like to determine how, and if, our high-level C output affects the C optimizer.
- Implementing more optimizations, both general and protocol-specific.
- Investigating separate compilation.

5
RESULTS

This section presents our prototype Prolac TCP specification. The specification is divided into small, modular pieces; computation is divided into small rules; and some of the protocol is implemented as extensions, overriding some parts of the base protocol. The prototype TCP can communicate with other TCPs, and we show that Prolac is not a performance bottleneck on a normal network.

5.1 TCP in Prolac

The Prolac TCP specification, *tcp.pc*, is one source file containing 1300 lines of Prolac and 700 lines of support C code. The support C code connects the Prolac specification with operating system code, specifically timers and mechanisms for sending and receiving messages.[1]

Figures 5.1 and 5.2 on pages 48 and 49 provide an overview of *tcp.pc*'s internal organization. The figures show namespaces and modules, but no module internals; for each module, supertypes are listed, but not imports.

The TCP which *tcp.pc* implements includes delayed acknowledgements and slow start, which are structured in the *tcp.pc* file as extensions to a base protocol. The base protocol and the two extensions are localized in file-level namespaces in *tcp.pc*. Each namespace contains definitions for several modules; the Simple namespace defines the base protocol's versions of the modules, while the extension namespaces define subtypes of those modules which override some of their behavior. This organization makes the base protocol simpler while grouping changes relevant to an extension. Extensions are easy to write: the Delay-Ack extension takes 30 lines of Prolac code, the Slow-Start extension takes 65.

A block of module equations, shown in Figure 5.2, chooses the global version of each of these extended modules. Elsewhere in the program, other modules referring to

[1] A single *tcp.pc* file supports communicating both over Unix pipes and through the Berkeley Packet Filter system, which doubles the size of the support code.

```
// Headers defines simple structures for network headers.
Headers {
  module Ethernet–Header;
  module Ip–Header;
  module Tcp–Header :> Ip–Header;   // single header for TCP/IP
}

// Simple defines the base protocol.
Simple {
  // Segment is Tcp–Header plus data and some helpful rules.
  module Segment :> Headers.Tcp–Header;

  // The TCB, a complicated structure, is built up through accretion.
  Tcb–Build {
    module Tcb–Base;   // basic fields defined in the RFC, allocation, statistics
    module Tcb–Timer :> Tcb–Base;   // ...plus retries
    module Tcb–Flags :> Tcb–Timer;   // ...plus some flags
    module Tcb–Drop :> Tcb–Flags;   // ...optionally drop some packets for testing
    module Tcb–Listen :> Tcb–Drop;   // ...plus listening
    module Tcb–Write :> Tcb–Listen;   // ...plus sending data
    module Tcb–Read :> Tcb–Write;   // ...plus receiving data
    module Tcb–State :> Tcb–Read;   // ...plus rules related to the connection’s state
    module Tcb–Buffering :> Tcb–State;   // ...plus buffering
  }
  // Choose the TCB other people will override.
  module X–Tcb ::= Tcb–Build.Tcb–Buffering;

  // Common pairs a Tcb and a Segment and provides common methods.
  module X–Common;
  // Reassembly supports reassembling data when it arrives out of order.
  module X–Reassembly :> Common;

  // The protocol is separated into modules for readability.
  module Listen, Syn–Sent, Other–States, Fin :> Common;

  // Sender builds up and sends messages with a convenient syntax;
  // Fragmentation fragments them if necessary.
  module X–Sender;
  module X–Fragmentation;

  // Timeout is the interface for timeout events.
  module X–Timeout;
}
```

Figure 5.1: *tcp.pc* overview, part 1—Base protocol

```
// Delay–Ack implements delayed acknowledgements as an extension to
// the base protocol.
Delay–Ack {
 module X–Tcb :> Simple.X–Tcb;
 module X–Sender :> Simple.X–Sender;
 module X–Reassembly :> Simple.X–Reassembly;
}

// Slow–Start, implementing slow start, is also an extension,
// but an extension of Delay–Ack (when they extend the same structure).
Slow–Start {
 module X–Tcb :> Delay–Ack.X–Tcb;
 module X–Common :> Simple.X–Common;
 module X–Fragmentation :> Simple.X–Fragmentation;
 module X–Timeout :> Simple.X–Timeout;
}

// Our simple header prediction module.
module Header–Prediction :> Common;

// Choose the components we'll actually use.
module Tcb ::= Slow–Start.X–Tcb;
module Common ::= Slow–Start.X–Common;
module Reassembly ::= Delay–Ack.X–Reassembly;
module Sender ::= Delay–Ack.X–Sender;
module Fragmentation ::= Slow–Start.X–Fragmentation;
module Timeout ::= Slow–Start.X–Timeout;

// Tcp–Interface has only static rules; C code calls these
// to interface with the Prolac code.
module Tcp–Interface;
```

Figure 5.2: *tcp.pc* overview, part 2—Extensions and interface

a possibly extended module—for example, Tcb—always refer to the global name: Tcb, not Simple.X-Tcb or Slow-Start.X-Tcb. This avoids most casts that would otherwise have been required; see §3.2.1 for more discussion.

In fact, parameterization through module equations goes further than this. The figure shows extension modules referring literally to the modules they are extending; in actuality, another level of indirection allows the programmer to alter the order in which extensions are applied simply by altering one block of module equations. For example:

```
Delay-Ack {
  module X-Tcb :> Parent-Tcb;
  module X-Sender :> Parent-Sender;
  ...
}
...
module Delay-Ack.Parent-Tcb ::= Simple.X-Tcb;
module Delay-Ack.Parent-Sender ::= Simple.X-Sender;
```

5.1.1 Status

The TCP specification is a prototype; we plan to rework it substantially for readability, correctness, and performance. Many aspects of TCP remain unimplemented, including TCP options, the urgent pointer, window scaling, and estimated round-trip time. *tcp.pc* also do not implement several conventional TCP optimizations, such as fast retransmit and caching TCBs and response messages. It only sends out segments of the smallest possible size, 512 bytes, which severely restricts maximum bandwidth.

None of this prevents *tcp.pc* from communicating with other TCPs; the omitted features are secondary or exist to improve network behavior. We do not expect any of these features to cause performance problems when implemented. In fact, several features, such as caching TCBs and response messages, should make the specification faster.

5.1.2 Sample code

This Prolac code, from the Simple.Other-States module, illustrates the general flavor of our TCP specification. It implements the initial part of processing for an incoming segment when the relevant TCB is in one of the SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, and TIME-WAIT states.

```
module Other-States :> Common has Tcb, Segment, Fin, Sender, Reassembly {
  // set up fields seg :> *Segment and tcb :> *Tcb so we don't have
  // to constantly pass parameters
  constructor(s :> *Segment, t :> *Tcb) ::= Common(s, t);

  // do-other is the main interface rule
  do-other ::= check-seq, check-rst, check-syn, do-ack, do-states,
        after-ack-and-syn;

  // "first check sequence number" (RFC793)
  // check-seq will throw an exception if all is not well.
  check-seq {
   check-seq ::= is-next-ack  // valid cases (see RFC793 p69):
        || within-wnd-begin  // if any succeed, the sequence number is OK
        || within-wnd-end
        || (ack ==> send-ack-fail)  // invalid cases: fail throws exception
        || send-reset-fail;
   is-next-ack ::=
        seg->len == 0 && tcb->rcv_wnd == 0
        && seg->seq == tcb->rcv_next;
   ... // define within-wnd-begin and within-wnd-end
  }
  ...
}
```

Note how close this code is to a literal translation of the TCP specification [Pos81]. The code for do-other, for example, is essentially the same as this language from the TCP specification [Pos81, pp69–76], except for portions *tcp.pc* does not implement:

tcp.pc

```
do-other ::= check-seq, check-rst, check-syn, do-ack,
     do-states, after-ack-and-syn;
do-states ::= state-specific processing;
after-ack-and-syn ::= check-urg, Reassembly(seg, tcb).process-text,
     Fin(seg, tcb).check-fin;
```

[Pos81]

> Otherwise, first check sequence number…second check the RST bit,…third check security and precedence…fourth, check the SYN bit,…fifth check the ACK field,…sixth, check the URG bit,…seventh, process the segment text,…eighth, check the FIN bit,…and return.

While not all of *tcp.pc* corresponds so clearly to the specification, Prolac's namespace features allow us to get pretty close. This congruence between implementation code and specification language was one of the goals of Prolac; we believe it makes Prolac specifications both more readable and easier to get correct.

5.2 Generated code

To get a feel for what *prolacc*'s generated C code looks like, consider the Sender module, which provides a convenient interface for sending packets. A Sender is meant to be used something like this:

```
Sender(sequence number).ack(acknowledgement sequence number)
   .rst.send(tcb)
```

Sender(*sequence number*) creates a new Sender object representing a packet about to be sent. The other rules—ack and rst here, but syn and fin are also possible—affect that packet: by setting the ACK bit and the acknowledgement sequence number, for example. The rules then return the modified Sender object so that other rules can be applied. The final send rule sends the packet.

In *tcp.pc*, Sender is overridden once, by Delay-Ack (when a packet is sent, any delayed acknowledgements do not need to be sent).[2] Here are simplified portions of the code:

```
// the basic sender
module Simple.X-Sender has Tcb, Segment, Ethernet-Header {
 field e :> *Ethernet-Header;
 field s :> *Segment;

 // set the sequence number
 constructor(seqno :> seqint) ::= e = Ethernet-Header.new,
    s = Segment.new, s->seq = seqno, s->len = 0;

 // rules for specific kinds of messages
 ack(ackno :> seqint) :> X-Sender ::= s->ackno = ackno, s->set-ack-flag, self;
 rst :> X-Sender ::= s->set-rst-flag, self;
 syn :> X-Sender ::= s->set-syn-flag, self;
 fin :> X-Sender ::= s->set-fin-flag, self;

 // internal rule, called by send and send-data
 set-default-values(tcb :> *Tcb, size :> int) ::= ...;

 // send with no data
 send(tcb :> *Tcb) ::= set-default-values(tcb, 0), ...;
 ...
}
```

[2] It is difficult to write this kind of code without reference types. Consider the Simple.X-Sender.ack rule: it is declared to return an X-Sender. Strictly speaking, this means that *prolacc* should force the return type to be Simple.X-Sender even if self has type Delay-Ack.X-Sender! Because of a problem mentioned in the reference manual—*prolacc* does not correctly handle the assignment of subtypes to supertypes—this code will actually work. Reference types are the solution to this problem; the return type of ack should be "reference to X-Sender", which would allow a Delay-Ack.X-Sender to be returned.

```
// Delay-Ack's sender
module Delay-Ack.X-Sender :> Simple.X-Sender has Tcb {
  // override set-default-values to clear any pending delayed acknowledgements
  set-default-values(tcb :> *Tcb, size :> int) ::=
        tcb->reset-delay-ack,
        super.set-default-values(tcb, size);   // call supertype's rule
}

// the global Sender is Delay-Ack's:
module Sender ::= Delay-Ack.X-Sender;
```

We now present a small portion of the C source code generated by the following Prolac expression:

```
inline[9] (Sender(97).ack(43).rst.send(tcb))
```

Most of the generated C has to do with setting up the Ethernet header, sending the packet, etc. We focus instead on Sender's rules, and particularly on *prolacc*'s optimization features. The expression above, if naively implemented, would generate at least three structure assignments and a dynamic dispatch on the overridden set-default-values rule; *prolacc* generates no structure assignments and no dynamic dispatches. The comments are not in *prolacc*'s output.

```
Simple__Sender_Base _ctor_T_153;   // temporary Sender
...
// Implementing Sender(97):
// set up Simple.X-Sender's virtual function table
(_ctor_T_153).__pcvt__Simple__Sender_Base =
    &Simple__Sender_Base__pcvt;
... code to initialize Ethernet header and segment omitted ...
((*((_ctor_T_153).s)).seq) = ((seqint)(97));   // 's->seq = seqno'
((*((_ctor_T_153).s)).len) = (0);   // 's->len = 0'
// now we're back in Delay-Ack.X-Sender's constructor; install its virtual table
(_ctor_T_153).__pcvt__Simple__Sender_Base =
    &Simple__Sender_Base__9Delay_Ack__Sender_Delay_Ack__pcvt;

// Implementing .ack(43):
((*((_ctor_T_153).s)).ackno) = ((seqint)(43));   // 's->ackno = ackno'
((*((_ctor_T_153).s)).flags) |= ((uchar)(16));   // 's->set-ack-flag'

// Implementing .rst:
((*((_ctor_T_153).s)).flags) |= ((uchar)(4));   // 's->set-rst-flag'

// Implementing .send(tcb):
// prolacc statically determines that Delay-Ack.X-Sender.set-default-values is
```

TCP implementation	Round-trip time (ms)	Bandwidth (KB/s)
Prolac – inlining	1.68	690
Prolac + inlining	1.68	684
OpenBSD	1.76	875

Table 5.1: Round-trip time and bandwidth over a public 10Mb/s Ethernet for three TCP implementations. Round-trip times are for 4-data-byte packets; bandwidth was measured by sending roughly 1MB of data.

```
// the relevant set-default-values definition, and inlines it.
((*(tcb)).flags) &= (~(2));  // ‘tcb->reset-delay-ack’
set_default_values__Simple__Sender_Base
   ((Simple__Sender_Base*)&(_ctor_T_153),
    (tcb), (0));  // ‘super.set-default-values(tcb, 0)’

... rest of code omitted ...
```

5.3 Performance

This section presents some preliminary results concerning our Prolac TCP implementation. Our measurements attempt to answer two questions. First, does Prolac impose unacceptable overhead? Second, do Prolac optimizations, particularly inlining, affect TCP performance? Future work, including refining *tcp.pc* and taking more careful measurements, will answer other interesting questions, such as: Can a readable Prolac TCP specification meet or exceed the performance of existing high-performance TCPs?

Table 5.1 shows measurements for three TCPs running over a public Ethernet. We measured our Prolac specification, *tcp.pc*, both with path inlining and with no inlining whatsoever; the baseline implementation is the mature TCP from OpenBSD, a free, BSD-derived Unix operating system. For these tests, Prolac's TCP was run in a user space program on an OpenBSD machine. The Berkeley Packet Filter system [MJ93] was used to forward the packets to Prolac.

Note that, in terms of round-trip time, Prolac is actually faster than OpenBSD. This surprising result might be due to several factors: OpenBSD is a complete TCP, while *tcp.pc* is not, and the OpenBSD version may be slowed down by the kernel's *mbuf* buffering scheme, for example. In terms of bandwidth, Prolac performs poorly because it only sends segments of the minimum length (512 bytes); once we implement TCP options, we expect to achieve comparable bandwidth.

Table 5.1 shows that, for normal networks, any Prolac overhead is negligible compared with network delay; Table 5.2 investigates Prolac overhead on a simulated fast network by timing communication over a Unix pipe. The raw pipe's round-trip

TCP implementation	**Round-trip time (μs)**	**TCP receive path (cycles)**
Prolac – inlining	241	2790
Prolac + inlining	214	1550
Hand-coded	161	1100
Raw pipe	33	N/A

Table 5.2: Round-trip time and TCP receive path latency over a Unix pipe. Round-trip times are for 4-data-byte TCP packets; TCP receive path latency measures time spent within the TCP protocol processing the arrival of a single segment containing 4 data bytes.

latency—i.e., without any protocol processing—is 34μs. The "hand-coded" TCP is a high-performance TCP hand-written in C, originally described in [WEK96]. Like *tcp.pc*, the hand-coded version is not a fully compliant TCP: it implements options, but all other functionality not present in *tcp.pc* is also not present in the hand-coded version.

The difference between Prolac with and without inlining is clear in Table 5.2: path inlining cuts 40% of the cycles spent in the TCP receive path. The remaining gap between Prolac and the hand-coded version is due to the fact that *tcp.pc* is untuned; an arriving segment will cause five *malloc*s in the Prolac version due to its simplistic buffering strategy, while the hand-coded TCP will usually not *malloc* at all.

These preliminary experiments show that, even untuned and with relatively few optimizations, Prolac does not impose unacceptable overhead for a normal TCP implementation. We have also shown that one of Prolac's existing optimizations, path inlining, is extremely effective at reducing protocol processing relative to unoptimized Prolac. For future work, we will tune our specification and perform more detailed measurements, hopefully showing that Prolac's overhead can be negligible even on a high-performance network.

6
SUMMARY

This thesis has described Prolac, a new statically-typed, object-oriented language designed for protocol implementation; *prolacc*, a compiler for Prolac generating efficient C code from a Prolac specification; and *tcp.pc*, a preliminary TCP specification in Prolac that is readable, extensible, and can communicate with other TCPs.

Contributions of this thesis include:

- The design of the Prolac language, including the novel concepts of module operators and implicit rules.
- Implementation of *prolacc*, the Prolac compiler, including algorithms necessary for compiling the Prolac language and methods for generating high-quality, high-level C code.
- A TCP specification which demonstrates that a Prolac program can be both readable and extensible without serious overhead. We are optimistic that future work will show that our Prolac TCP can be made as fast as most TCP implementations hand-written in C.

Future work on this project will include:

- Designing and implementing multiple inheritance.
- Enhancing *prolacc*'s error recovery.
- Implementing further optimizations.
- Writing specifications for other protocols.
- Completing and refining the Prolac TCP specification.
- Making a detailed performance analysis of the TCP specification.

APPENDIX A
PROLAC LANGUAGE REFERENCE MANUAL
PRELIMINARY VERSION OF AUGUST 26, 1997

1 Introduction

This is a draft of the Prolac language reference manual. Prolac is an object-oriented language designed for creating efficient but readable protocol implementations. This draft does not always precisely define Prolac's versions of conventional concepts; we will address this issue in a later version of the manual.

After a brief section on lexical analysis, the manual is structured in a top-down fashion: the largest Prolac structures, modules, are discussed first, while types and expressions are saved for last.

A Prolac specification is stored in a single file; the Prolac compiler reads the file, analyzes it, and produces two output files in the C language. The first output file is a header file containing C structure definitions corresponding to Prolac structures; the second is a C source file containing definitions for any exported Prolac rules (§5.5).

Prolac constructs are completely order-independent: anything can be used before it is declared or defined. Thus, Prolac input files can be structured top-down, bottom-up, or however you like.

1.1 Terminology

A *name* is either a *simple name*—that is, a name with one component, or, equivalently, an identifier—or a *member name*, which is a member expression 'X.n' (§9.7) where 'X' is a name. (Note that a pointer-to-member expression 'X->n' is not a name.) A name which is not simple is also called a *qualified name*.

A *feature* is simply something that has a name. Modules (§3), namespaces (§4), rules (§5), and fields (§6) are features.

2 Lexical analysis

Prolac programs are stored as a sequence of ASCII characters, which are interpreted as the following sections describe.

2.1 Whitespace and comments

As in C, whitespace—comments as well as spaces, horizontal and vertical tabs, formfeeds, carriage returns, and newlines—is ignored except as it separates other tokens. Prolac supports both C's comment syntax '/* ... */' and C++'s '// ... *newline*'.

2.2 Identifiers

An *identifier* is an arbitrarily long sequence of letters, digits, underscores, and hyphens '-', which tend to enhance readability. Identifiers must start with a letter or underscore; an identifier cannot contain two consecutive hyphens or end in a hyphen. For example:

```
a-pretty-long-identifier  ≡  one identifier
-23x_x-23                 ≡  '- 23 x_x-23'
x--                       ≡  'x --'
```

Identifiers which differ only in substituting hyphens for underscores and vice versa are considered identical; thus, 'one-thing' and 'one_thing' are the same identifier. (In generated C code, Prolac substitutes underscores for all hyphens.)

Identifiers containing double underscores '__' or an equivalent ('-_', '_-') are reserved for the implementation.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

all
allstatic
bool
char
constructor
else
elseif
end
export
false
field
has
hide
if
in
inline
int
let
long
module
notusing
outline
rename
self
seqint
short
show
static
super
then
true
uchar
uint
ulong
ushort
using
void

These single characters serve as operators or punctuation:

```
!   %   ^   &   *   (   )   -   +   =   {   }   |   ~
[   ]   \   ;   '   :   "   <   >   ?   ,   .   /
```

These multi-character sequences are also single tokens:

```
->    ++    --    :>    <<    >>    <=    >=    ==
!=    &&    ||    +=    -=    *=    /=    %=    ^=
&=    |=    <<=   >>=   ==>   |||   ::=   %{    %}
```

2.4 Numbers and literals

Prolac's definitions for number, string, and character literals are the same as C's. However, Prolac does not support floating-point types or values, so any floating-point literal encountered is an error. The current *prolacc* compiler does not really support string or character literals either.

2.5 Preprocessing

Although the Prolac language is preprocessing-neutral—it does not require or encourage any preprocessing phase—the *prolacc* compiler does have some features to facilitate preprocessing Prolac files with *cpp*, the C preprocessor. In particular, the lexer understands '# *line*' directives: any error messages will have appropriate line numbers. *Prolacc* also generates '# *line*' directives in its C output files.

2.6 Including C code

C code may be included in a Prolac file in two ways. First, *C blocks* (§9.6) occur within rule bodies, where they specify code to run during the rule's execution. An open brace '{' within an expression introduces a C block; to read the C block, the lexer copies characters without interpretation, respecting nested braces, until the next unbalanced '}' character not in a string or character literal or a comment.[1] C blocks can refer to some Prolac objects using Prolac names; see §9.6 for details.

The '%{' and '%}' operators specify *support C code* not relating to any rule; support C code is passed unchanged to the output file at file scope. Support C code cannot refer to any Prolac objects using Prolac names.

'%{' can occur wherever a definition is expected; the lexer then copies characters without interpretation until the next '%}' sequence not in a string or character literal or a comment. Note that support C code can therefore contain unbalanced braces.

Support C code occurring in the input file before any actual Prolac code is collected, in order of definition, and placed in the output C source file before any Prolac-generated code. All other support C code is collected in order of definition and placed at the very end of the output source file.

3 Modules

Modules are Prolac's basic means of organizing programs. Modules contain rules (§5), which represent computation, and fields (§6), which represent data. Each module is also a namespace (§4). A module is wholly self-contained; it must explicitly import other modules if it wants to refer to them (§3.3). Modules can be subtypes of other modules (§3.2). A module can have a special rule which initializes objects of its type (§5.3). Module definitions cannot be nested.

A module definition looks like this:

[1] As of this writing, the *prolacc* compiler will be confused by '{' and '}' characters in string or character literals or comments.

```
module name [:> parents...] [has imports...] {
  ...features...
}
```

3.1 The module header

The *module header* defines the interface between a module and the rest of the program. The interior of a module can only refer to other modules if they have been explicitly specified in the module header.

The module header has two parts, *parents* and *imports*. Each part is a comma-separated list of module expressions, where a module expression is a module name possibly modified by module operators (§3.4).

The names of parents and imports must be distinct; specifically, the rightmost *components* of all parent and import names must be distinct. These name components can be used within the module to refer to the parents and imports. Thus, this example is illegal:

```
Alice {
  module M { ... }
}
Bob {
  module M { ... }
}
module N has Alice.M, Bob.M {
  // error: two definitions for 'M'
}
```

Use module equations (§3.4.1) to get around this restriction.

3.2 Supertypes

A module may have any number of *supertypes*, which must be other modules. The supertype relation is transitive: if A is a supertype of B and B is a supertype of C, then A is a supertype of C. A module's immediate supertypes, which are explicitly listed in its module header, are also called its *parents*.

[Implementation note: We have not yet completed the definition of multiple inheritance in Prolac. The *prolacc* compiler works only with single inheritance as of this writing, and some other parts of this manual (e.g., §5.2) also assume single inheritance.]

A module inherits its supertypes' features—i.e., their imports, fields, namespaces, and rules. Each parent's features are generally available without qualification under their own names, as if the module's definition was inserted into the parent's definition; see §4.5.1 for a detailed description of how parents' namespaces are combined into a module's namespace.

Supertypes interact with the Prolac type system (§8). Specifically, if P is a supertype of M, then M may be used wherever P is; or, in notation, M :> P (§8.7).

A module can *override* some of its supertypes' rules; on an object of that module type, the overriding definitions will be used whenever the parents' overridden rules are called. This process, called *dynamic dispatch*, is described in §5.2.

It is not an error to explicitly mention another parent's supertype as a parent. This is not actually multiple inheritance; only one copy of the supertype in question is inherited. For example, this is legal:

```
module M { ... }
module N :> M { ... }
module O :> M, N { ... }
```

This can be useful to redefine some of M's module operators—for example, its inline levels (§9.9.1).

3.3 Imports

To gain access to a module M which is not one of its supertypes, a module must list M as an *import* in its module header's has clause. Imports' features are generally not available under their own names—the importing module must qualify them through the import's name (but see §5.4). Imports do not interact with the Prolac type system; if M imports I, it is not true that M :> I.

3.4 Module operators

Prolac has a powerful collection of *module operators*, which are simply operators that act on modules instead of values. There are operators that control a module's namespace (hide, show, and rename; §4.6), operators that control how implicit rules (§5.4) are found (using and notusing; §5.4.3), and an operator controlling how rules are inlined (inline; §9.9.1).

A module operator expression has a module value; thus, module operators may be used anywhere a module is expected. A module's creator can suggest how the module should be used with module operators; the module's user can use the same syntax to define how it actually will be used. For example, a module's creator might suggest which rules should be inlined and hide any rules not in the normal interface; its user might force a different set of rules to be inlined, and specify which of the module's rules are to be used during implicit rule lookup.

Each module operator described in this manual changes how a module is perceived—its namespace, its exported rules, its preferred inline levels—without changing the module itself. In particular, this means that two module expressions differing only in module operators have the same type.

The individual module operators are described elsewhere in the manual, closer to the features they affect.

3.4.1 Module equations

A *module equation* defines a module as an abbreviation for applying module operators to another module. The syntax is:

```
module new-module ::=
  old-module [module operators...] ;
```

Other modules can then refer to *new-module* as an abbreviation for '*old-module module operators...*'. A module equation does not define a new type.

3.4.2 After-module operators

A module's creator can apply module operators to the module by placing them directly after the module definition. For example:

```
module M {
 interface-rule ::= ...;
 impl-rule-1 ::= ...;
 impl-rule-2 ::= ...;
} hide all show interface-rule;
```

A module definition with after-module operators 'module M { ... } *operators*' is effectively equivalent to this definition with a module equation:

```
module "M" { ... }
module M ::= "M" operators;
```

4 Namespaces

A *namespace* is a container mapping names to features. The most common examples of namespaces in Prolac are modules—each module defines a namespace. The programmer can also create *explicit namespaces,* either outside all modules (to organize modules into groups) or within a module (to organize rules into groups).

Namespaces may be nested; thus, a namespace may have a *parent namespace* which is used during name lookup. Most namespaces are open; in other words, you can define a namespace in parts through several definitions. (Module namespaces are not open: each module namespace is defined in exactly one place.)

4.1 Explicit namespaces

Namespaces are introduced simply by following the name of the namespace by an open brace '{':

```
namespace-name {
 ...features...
}
```

4.2 Name lookup

When a simple name *n* is used, Prolac searches for *n* in the current namespace, then in its parent, then its grandparent, etc.; the first definition found is used. If no definition is found, the lookup fails.

When a member name '*X.n*' is used, Prolac first looks up the name *X* in the current namespace; the result's type must be a namespace. After this, Prolac looks for *n* in *X*'s namespace: no recursive search is performed. If no definition is found, the lookup fails. This algorithm is described in more detail in §9.7.

A name which defines a Prolac feature is treated differently; see §4.2.2 below.

4.2.1 Global names

The direct member operator '.' can also be used as a prefix; for example, '.M' is a name. To look up a name '.*n*', Prolac first finds the current *most global namespace, GS*. Within a module, the most global namespace is the module's top-level namespace; outside any module, it is the file namespace. Once it has found *GS*, the name lookup proceeds as if the expression was '*GS.n*'. This syntax allows an expression in an inner namespace to refer to features in an outer namespace, whether or not their names have been reused in the inner namespace.

4.2.2 Defining names

A name used to define a Prolac feature is treated differently than a name used to refer to a Prolac feature. This example illustrates where *defining names* occur:

```
module DN1 {  // module definition
 DN2 {  // namespace definition
  DN3 ::= ...;  // rule definition
  field DN4 :> ...;  // field definition
 }
} rename (DN5 = ...);
// module operators that create names
```

A definition '*n1...nk* ≡ *F*' is an abbreviation for a definition within nested namespaces, '*n1* { ... { *nk* ≡ *F* } ... }'. Thus, these two examples are completely equivalent:

```
module Package.M {
 internal.rule ::= ...;
}
Package {
 module M {
  internal {
   rule ::= ...;
  }
 }
}
```

Because namespaces are open, the intermediate namespaces created by such a definition can be extended by other namespace definitions.

Note that the usual search algorithm for simple names (i.e., search the current namespace, then its parent, etc.) does not apply when a name is being defined. A definition ‘`n1` ≡ *F*’ always means “bind `n1` to *F* in the current namespace”.

4.2.3 Explicit supertype overrides

Consider a feature definition at the top level of a module M. If the first component of the defining name is the name of one of the module’s supertypes, S, this definition is an *explicit supertype override* and is treated specially by Prolac. The remainder of the defining name is looked up in S’s complete namespace (§4.4); thus, any name defined by S can be overridden, regardless of whether or not it is visible in M.

An explicit supertype override cannot introduce new names into S’s namespace: it can only override existing names in S. Since fields cannot be overridden, this means that explicit supertype overrides must refer to rules.

The body of a rule defined as an explicit supertype override is resolved in a special namespace. This namespace layers S’s complete namespace, with any intermediate namespaces referred to in the defining name, over M’s top-level namespace. Thus, names are looked up as if the rule had really been defined in S, except that if a name cannot be found, M’s top-level namespace is checked before an error is reported.

Here is an example of an explicit supertype override.

```
module S {
 override-me ::= ...;
 x ::= 97;
} hide all;
module M :> S {
 m-rule ::= ...;
 x ::= ...;
 S.override-me ::= x,  // finds S.x
    m-rule;  // finds M.m-rule
}
```

4.2.4 Notes

Prolac allows rules which take no arguments to be called without parentheses (§9.2). This means that an expression which looks like a name may actually contain rule calls; for example:

```
module M1 {
 a.b.c.d.e ::= ...;
 test ::= a.b.c.d.e;  // really just a name
}
module D {
 d.e ::= ...;
}
module M2 has D {
 a :> M2 ::= ...;
 b.c :> D ::= ...;
 test ::= a.b.c.d.e;  // not just a name!
 // same as:
 test-2 ::= let temp1 :> M2 = a() in
   let temp2 :> D = temp1.b.c() in
     temp3.d.e()
   end end;
}
```

These name-like expressions are not allowed where a name is required; for example, ‘M2.a.b.c.d.e’ cannot be hidden by the hide module operator.

4.3 Module namespaces

Each module is a namespace. A module namespace is sealed off from surrounding namespaces; in other words, it has no parent. To illustrate:

```
module Find-Me { ... }
N {
 ... Find-Me ...  // finds Find-Me
}
module M {
 find-me-2 ::= ...;
 inner {
   ... find-me-2 ...  // finds M.find-me-2
 }
 ... Find-Me ...  // does not find Find-Me
}
```

The outside world—specifically, other modules—can only be reached through a module’s parents and imports, which the module explicitly declares in its module header (§3.1). This means that a module interior is completely self-contained and relatively insulated from the effects of other modules’ name changes: only the module header needs to be changed.

4.4 Complete and default namespaces

Each module M has a *complete namespace,* which is the namespace seen inside the definition of M. It contains a union of M’s parents’ complete namespaces, as well as every rule, field, and nested namespace defined in M, and every implicit rule (§5.4) used by M. While module operators may rename or hide some features from a module’s complete namespace, the complete namespace is always accessible through the special syntax ‘*module-name*.all’. Thus:

```
module M {
  x ::= ...;
} hide all;
module N :> M {
  y ::= x,   // error: no definition for implicit rule 'x'
     M.x,   // error: 'M' has no 'x' feature
     M.all.x;   // OK
}
```

Most clients of M see a different namespace, M's *default namespace.* M's default namespace is equal to the its complete namespace with all the implicit rules hidden and any after-module operators (§3.4.2) applied. (Note that the after-module operators might show some of the implicit rules again.) The default namespace is the namespace found when you refer to a module without further qualification.

4.5 Creating the complete namespace

A module M's complete namespace R is created by merging all of M's parents' complete namespaces, and then merging M's internal namespace into that. Finally, any implicit rules referred to in M are stored in M's top-level namespace.

Imports coming from M's parents are not merged into the complete namespace. Name conflicts during the merging process are generally an error, unless the two names intuitively refer to the same feature (the same supertype or the same rule).

The remainder of this section describes this process in more precise detail.

4.5.1 Parents

If M has parents P1,..., Pk, we first recursively create their complete namespaces, then combine these into a new result namespace R.

We do not actually combine all of the parents' features into the result namespace. Specifically, we omit all imports and all constructors.

Of course, there may be name conflicts when combining namespaces (two features with the same name coming from different parents). We now consider how a conflict is resolved, considering only a two-feature conflict—specifically, a conflict where a feature F1 from parent P1 and a feature F2 from parent P2 are inherited under the same name. (Three or more conflicting features are handled in essentially the same way.) There are four cases:

1. If F1 and F2 refer to the same feature, no error is reported. This can happen when F1 and F2 both inherit a feature from a mutual supertype.

2. If F1 and F2 are supertypes of P1 and P2, respectively, and they refer to the same module, no error is reported.

3. If F1 and F2 are both namespaces, then R inherits one namespace with the combined contents of F1 and F2. Conflicts found when combining these namespaces are resolved using this algorithm.

4. Otherwise, an error is reported.

4.5.2 Imports and parents

After creating the merged namespace R, we combine it with features representing M's parents and imports.

Each parent or import is imported under a simple name, N, which is the rightmost component of its module name. The complete namespace for M will have a feature named N representing that parent or import.

Any conflict between a feature FP, inherited in the last stage from some parent P, and FM, a parent or import being added, is resolved silently in favor of FM (i.e., FM replaces FP in R).

4.5.3 The internal namespace

After creating R, we merge M's *internal namespace,* I, into it. A module's internal namespace contains only the definitions the user explicitly provided when defining the module; in particular, changing a module's module header does not change its internal namespace. Again, there may be conflicts; conflicts between two features FR, from R, and FI, from I, are resolved as follows:

1. If FR and FI are both rules, then the rule FI *overrides* the rule FR; there is no conflict. See §5.2 for details on overriding.

2. If FR and FI are both namespaces, the contents of FI are merged into FR. Conflicts found during the merge are resolved using this algorithm.

3. If FR is an supertype and FI is a namespace, then the user has given an explicit supertype override for supertype FR (§4.2.3). We look up FR's complete namespace C, and merge FI into C. The merging process uses this algorithm, except that all names in FI must already exist in C: that is, it is an error for there *not* to be a name conflict.

4. Otherwise, an error is reported. If FR was inherited from some parent (i.e., it is not one of M's parents or imports introduced in §4.5.2), FI replaces FR.

4.5.4 Implicit rules

Finally, all rules that were defined in M have their body expressions scanned for implicit rules. If any simple name in a rule body cannot be found through the usual namespace search, it means that name is an implicit rule (§5.4) attached to some field or import; its definition will be found later. The name in question is defined in R as a new implicit rule.

The resulting namespace R is M's complete namespace.

4.6 Namespace module operators

This section describes the three module operators which affect module namespaces: hide, show, and rename.

4.6.1 'hide'

The hide operator hides some of a module's names. The left operand to hide is a module specification; the right operand is a name, a comma-separated list of names,[2] or 'all', which hides all of the module's names. Some examples:

```
M hide internal-operation
M hide (evil, kill, horrible.death, maim)
M hide all
```

It is an error to attempt to hide a name through a supertype or import; for example, this is an error:

```
module A {
 x ::= ...;
}
module B :> A {
}
... B hide A.x ...  // illegal!
```

Note that hidden names are still available by explicit qualification through the module's complete namespace (§4.4).

4.6.2 'show'

Show is the converse of hide. Hide makes names from the complete module namespace inaccessible; show makes hidden names accessible again.

The left argument to show is a module specification; the right argument must be a list containing any number of names 'n' and name assignments '(new = old)'.[3] A name 'n' is essentially equivalent to the name assignment '(n = n)'.

To evaluate a show operator applied to a module M, Prolac first looks up the old name, 'old', in M's complete namespace (i.e., as if through 'M.all'). This name may be further qualified through M's supertypes. This must result in a feature F.

Prolac then evaluates the new name 'new' in M's *current* namespace. This name must *not* be qualified through M's supertypes and imports.[4] It is an error if a feature with this name already exists; otherwise, Prolac binds F to this name in the resulting module.

It is an error to show a module's constructor in a nested namespace or under a name other than constructor.

Note that show can be used to make a single feature available under multiple names; for example:

```
module M {
 bad ::= ...;
}
module M2 ::= M show (good = bad);
```

Since M2.bad and M2.good are the *same* feature—not two copies of a feature—overriding either one will effectively override them both.

Because old names are looked up in the module's complete namespace, renamed features cannot be hidden and then shown; this code will not work:

```
module M { r ::= ...; }
module N :>
  M rename (r = weird)
    hide weird  // OK
    show weird  // error: no weird in M
    show r  // OK
{}
```

The 'show all' operation is not yet implemented.

4.6.3 'rename'

Rename changes the name you use to access a feature. The left operand must be a module specification; the right operand must be a name assignment '(new = old)' or a list of name assignments.

'M rename (new = old)' is equivalent to 'M show (new = old) hide old'. It is an error for either old or new to be qualified through supertypes or imports. It is an error to attempt to rename a module's constructor, or to give a new name which conflicts with an existing name.

4.6.4 Usage commentary

The namespace module operators are powerful tools—perhaps too powerful: I am worried about their misuse. Particularly dangerous is the ability to change a module's namespace arbitrarily with show and rename. While this is useful when fitting one module to another, incompatible interface and when resolving multiple inheritance conflicts, other solutions might be cleaner in the long run.

A reasonably careful discipline for using the namespace module operators might be as follows:

1. Never hide or show names in a nested namespace. Instead, hide or show the whole nested namespace.[5]

[2] Such a list must be enclosed in parentheses because of module operators' high precedence.

[3] Again, the parentheses are necessary.

[4] A show operation 'M show A.n'—that is, where there is only one name, and it is qualified through an supertype or import—is not an error; it is equivalent to 'M show (n = A.n)'.

[5] The original definition of Prolac enforced this restriction, which

2. Never use show to change a feature's name.

3. Never use rename. Instead, use hide and explicit supertype overrides (§4.2.3), or forwarding methods. For example, here is a multiple inheritance conflict solved using all three methods:

```
module A {
  x ::= ...;
}
module B {
  x ::= ...;
}
module C :> A, B { }   // conflict on 'x'

// rename solution:
module C2 :> A rename (x = a–x),
    B rename (x = b–x) { }

// Explicit supertype override solution:
module C3 :> A hide x, B hide x {
  // submodules use A.all.x and B.all.x
}

// Forwarding solution:
module C4 :> A hide x, B hide x {
  a–x ::= A.all.x;
  b–x ::= B.all.x;
  // submodules override A.all.x and B.all.x
  // but refer to a–x and b–x
}
```

Another forwarding scheme allows submodules to override a–x and b–x as well as refer to them, but at the cost of one more dynamic dispatch per call of A.x or B.x in a supertype:

```
module C5 :> A hide x, B hide x {
  A.x ::= a–x;
  B.x ::= b–x;
  a–x ::= super.A.all.x;
  b–x ::= super.B.all.x;
}
```

5 Rules

A Prolac program organizes code by dividing it into *rules.*[6] Like functions in most programming languages, rules can take parameters and can return a value. Rules can call one another, possibly recursively.

Rules can be *static* or *dynamic.* Static rules are equivalent to normal functions. Dynamic rules are called with an implicit reference to some object of module type; thus, dynamic rules are like most object-oriented languages' methods.

A module may provide new definitions for some of its supertypes' dynamic rules. This process is called *overriding* the supertypes' rules. When an overridden rule is called on an object with that module's type, the call will be handled by the new, overriding definition instead; this is called *dynamic dispatch.*

Each rule has an *origin,* which is the module that provided the first, non-overriding definition for the rule. Each rule definition comes from some module, which is called its *actual.* For example:

```
module M {
  r ::= ...;   // origin = M, actual = M
}
module N :> M {
  r ::= ...;   // override: origin = M, actual = N
}
```

Rule definitions look like this:

rule–name(*parameters...*) :> *return–type* ::= *body* ;

Each *parameter* has the form *name* :> *type*. If there are no parameters, the parentheses may be omitted. If the return type is bool (§8.4), ':> *return–type*' may be omitted. *Body* is an expression (§9); it may be omitted, in which case any call of the rule will result in a run-time error.[7] Here is the shortest rule definition possible:

```
x::=;
```

Static rule definitions have a static keyword before the rule name.

5.1 Static and dynamic rules

A dynamic rule defined in module M is called with reference to some object whose type is either M or one of M's subtypes. Within the rule body, this object is called self. Any dynamic rule call must provide a value for self; this is done with member operator syntax (§9.7). The intuition is that the rule is also a member of the module. For example:

```
module M {
  r ::= ...;
}
module N has M {
  ... let m :> M in
    m.r
  end ...
}
```

we may reenforce.

[6] This unorthodox terminology is mostly due to Prolac's prehistory as a *yacc*-like language.

[7] Such a rule can still be overridden, however.

It is an error to refer to a dynamic rule without reference to an object. This is a special case of the rules for static and dynamic context described in §9.7.

Within a dynamic rule, 'self.' can be elided in field and rule references. For example, these two rule definitions are identical:

```
module M {
  rule ::= ...;
  field f :> ...;
  d1 ::= self.rule, self.f;
  d2 ::= rule, f;
}
```

It is an error to refer to a dynamic rule or field in a static rule, unless the rule or field is accessed through an object.

5.2 Overriding and dynamic dispatch

A rule declared with the same name as a supertype's rule is an *overriding rule.* The exact algorithm for determining whether a rule is an override is described in §4.4 (specifically, §4.5.3); this section describes the semantics of overriding.

Only dynamic rules may be overridden. It is an error to attempt to override a static rule.

5.2.1 Correctness

An override of rule RA by rule RB is correct only if RB's signature[8]—i.e., the number and types of its parameters and its return type—agree with RA's by the usual *contravariance rule.* Specifically:

1. RA and RB must take the same number of parameters.

2. For corresponding parameter types PA and PB, we must have PB :> PA; that is, PA and PB are equal, or PB is a supertype of PA. Thus, the overriding parameters are the same as, or more general than, the overridden parameters. This ensures that any value passed as a parameter to RA is also valid as a parameter to RB.

3. For the rules' return types TA and TB, we must have TA :> TB; that is, TA and TB are equal, or TB is a subtype of TA. Thus, the overriding return type is the same as, or more specific than, the overridden return type.

It is an error to define an incorrect override.

5.2.2 Rule selection

The process of deciding which actual rule definition to use for a given rule call is called *rule selection.* Rule selection depends on only one factor: the *run-time type* of the object which will become the rule's self.

In Prolac, an object of type M, where M is a module, can be used in place of any of M's supertypes. Therefore, an object's run-time type, or the actual type of the object used at run time, can differ from its static type, or the type used to declare the object.

For objects of simple module type, the static type is always identical to the dynamic type; Prolac's semantics are call-by-value, like C, rather than call-by-object, like Clu. For example:[9]

```
module M { ... }
module N :> M { ... }
module U has M, N { ...
  let m :> M, n :> N in
    m = n  // This assignment actually copies
    // the M part of n's state into the m object!
    ...
}
```

An object's static and dynamic types can differ only if the object is referenced through a pointer (§8.6), or the object is self.[10] For example:

```
module M {
  r ::= ...;
}
module N :> M { ... }
module U has M, N { ...
  let m :> *M = ..., n :> *N = ... in
    m = n,  // *m has static type M but dynamic type N,
    // as the m pointer actually points to the n object.
    m−>r  // Within r, self will have dynamic type N.
    ...
}
```

The rule selected for a rule call O.r depends then on the run-time type of O. Let the run-time type of O be T; then we select the most specific definition for r existing in T and its supertypes. A more precise definition follows:

Consider all possible definitions for r coming from T and all of its supertypes. Let these definitions be d1,..., dk, coming from modules M1,..., Mk. We only consider single inheritance here; therefore, the modules M1,..., Mk must form a total order under the supertype relation. Let

[8] When and if Prolac supports rule types, we will say "RB's type" here.

[9] As of this writing, the *prolacc* compiler cannot handle object assignment of subtypes to supertypes. This example and any similar example (e.g., parameter passing) will result in bad C code being generated.

[10] Unlike any other value, self is actually a reference. We are considering adding reference types to Prolac, however.

Ms ∈ {M1,…, Mk} be the most specific module in this order—that is, we have Ms :> Mi for all Mi ∈ {M1,…, Mk}. The definition selected for the rule call is then ds, the rule definition from Ms.

5.2.3 'super'

A module can specify that its immediate supertypes' definition for a rule be used by calling the rule through the special object super. For example:

```
module One {
  f :> int ::= 1;
}
module Two :> One {
  f :> int ::= 1 + super.f;   // returns 2
}
module Three :> Two {
  f :> int ::= 1 + super.f;   // returns 3
}
```

Except for its behavior in relation to dynamic rules, super acts exactly like self.

Any dynamic rule in the module can use super to call any inherited rule; it is not limited to calling the supertypes' version of the current rule. For example:

```
module One ...   // same as above
module Two :> One {
  f :> int ::= 2;
  old-f :> int ::= super.f;   // returns 1
}
```

Note that super is very different from calling a rule through the module's parent references. Calling a rule through a parent reference still refers to the most specific definition of the rule; for example:

```
module One ...   // same as above
module Two :> One {
  f :> int ::= 2;
  test ::= f,   // calls Two.f
    One.f,   // also calls Two.f
    super.f,   // calls One.f
    super.One.f;   // also calls One.f
}
```

It is not immediately clear how to generalize super for multiple inheritance.

5.3 Constructors

Each module may contain a special rule, called its *constructor,* which is called when objects of the module type are created. (See §9.2.2 for more information on when constructors are called.) The constructor is distinguished by its name, which is the keyword 'constructor'. A constructor must appear in the module's top-level namespace; it must not be static and must not define a return type, but it can take parameters.

The body of a constructor rule is parsed slightly differently than those of normal rules. It consists of zero or more *parent constructor expressions* separated by commas, followed by a normal expression. A parent constructor expression is just a constructor call (§9.2.2) for one of the module's parents; it allows the subtype to give any necessary arguments for the parents' constructors. For example:

```
module Friends {
  field num-friends :> int;
  constructor(f :> int) ::= num-friends = f;
}
module Friends-and-Enemies :> Friends {
  field num-enemies :> int;
  constructor(f :> int, e :> int) ::=
    Friends(f),   // parent constructor expression
    num-enemies = e;
}
```

In a normal context, the subexpression 'Friends(f)' would have no visible effect; it'd create a new Friends object, then throw away the result. As a parent constructor expression, however, it does have a visible effect—specifically, initializing self.num-friends.

If a parent is not mentioned in the parent constructor expressions, its constructor is called without arguments. It is an error to omit a parent constructor expression for a parent that requires arguments.[11]

If no constructor is provided for a module, Prolac will generate a default constructor which calls any necessary parent constructors, but does nothing else. Exactly one of the module's parents' constructors may take arguments in this case; if so, the generated constructor will take the same number and types of arguments and pass them to that supertype's constructor.

The *prolacc* compiler does not currently generate any necessary constructor calls for a module's fields. In future, parent constructor expressions will be generalized to support field initialization as well as parent initialization.

5.4 Implicit rules

Explicit rules are rules the user explicitly defines. *Implicit rules,* on the other hand, are created automatically when a Prolac expression refers to an undefined name (§4.5.4). The compiler will fill in the implicit rule's definition by looking through the module's fields and imports, subject to

[11] As of this writing, *prolacc* implements neither default parent constructor call, nor the necessary checks on omitted constructors.

any using and notusing module operators, until it finds a rule with the same name. Implicit rules can considerably simplify the text of a module by eliding frequently-used object or module names.

A motivating example seems in order. Consider a module Segment-Arrives implementing part of the TCP protocol. This module will frequently refer to the current transmission control block, tcb, which has type *Tcb. Here is a partial definition for a hypothetical Tcb module:

```
module Tcb {
 field state :> int;
 field flags :> int; ...
 // Which state are we in?
 listen ::= state == 0;
 syn-sent ::= state == 1;
 syn-received ::= state == 2;
 ...
}
```

Now, how should we implement Segment-Arrives? We want to divide computation into many small rules, so we could make tcb a parameter to each; however, passing the parameter would quickly become tiresome. Therefore, we make tcb a field in Segment-Arrives. Here is a sample of what our code might look like, considerably simplified for didactic purposes:

```
// Example 1
module Segment-Arrives has Tcb {
 field tcb :> *Tcb;
 // constructor sets tcb
 check-segment ::=
  (tcb->listen ==> do-listen)
  || (tcb->syn-sent ==> do-syn-sent)
  || (tcb->syn-received ==> do-syn-received)
  || (tcb->established ==> do-established)
  ...;  // and much more!
}
```

The repetition of 'tcb->' is tedious and hinders quick comprehension of the code. We know Segment-Arrives deals with only one tcb; why should we have to tell the compiler which tcb we mean again and again?

One solution is to generate forwarding rules in Segment-Arrives. We hide these forwarding rules using after-module operators (§3.4.2), since they are artifacts of the implementation.

```
// Example 2
module Segment-Arrives has Tcb {
 field tcb :> *Tcb;
 check-segment ::=
  (listen ==> do-listen)
  || (syn-sent ==> do-syn-sent)
  ...;
 listen ::= tcb->listen;
 syn-sent ::= tcb->syn-sent;
 ...
} hide (listen, syn-sent, ...);
```

This is better; however, the forwarding rules clutter the module definition and, again, are tedious and error-prone to write.

The solution in Prolac is to use implicit rules. We use the using module operator (§5.4.4) to open tcb for implicit rule search. When the compiler creates Segment-Arrives's complete namespace, it searches its rules for undefined names, entering them in Segment-Arrives's top-level namespace as undefined implicit rules. Later, it creates their definitions through a search process. It marks the implicit rules as highly inlineable and hides them in the default namespace. Thus, the compiler transforms the following code into something like Example 2:

```
// Example 3
module Segment-Arrives has Tcb {
 field tcb :> *Tcb using all;
 check-segment ::=
  (listen ==> do-listen)
  || (syn-sent ==> do-syn-sent)
  ...;
}
```

Example 3 is, in a sense, optimal: nothing distracts the reader from exactly what the module is doing.

This example has demonstrated that implicit rules can make code more readable rather than less. Overuse of implicit rules can make code very difficult to understand, however; moderation is required, as with any powerful tool.

Note that only rules can be found implicitly: referring to a field requires explicit syntax.

The remainder of this section describes the various mechanisms supporting implicit rules, specifically the implicit rule search algorithm and the using and notusing module operators.

5.4.1 Implicit rule search

Prolac allows implicit rules to be found in a module's supertypes. Therefore, this example will work:

```
module I {
 implicit ::= ...;
}
module S has I using all { }
module M :> S {
 ... implicit ...  // finds I.implicit
}
```

This is necessary behavior; consider, for example, inheriting from a module in order to extend it—not being able to refer to implicit names which the parent module used would be very counterintuitive. Unfortunately, this seriously complicates implicit rule search. The following algorithm nevertheless provides relatively few surprises.

To find an implicit rule named *n* in module M, a breadth-first search is performed. First all of M's imports and fields are checked for a top-level rule named *n*; then all M's parents' imports and fields; then all M's grandparents' imports and fields; and so on.

The algorithm finds a definition for *n* in some import or field iff:

1. That import or field has a module or pointer-to-module type;
2. That module has a top-level visible rule or namespace call (§9.2.1) named *n*; and
3. A 'using *n*' or 'using all' directive (§5.4.4) is in effect on that module.

A warning is given if, at any point during the search, the algorithm finds a field or a namespace that cannot be called (§9.2.1) instead of a rule.

If two or more rule definitions for *n* are found in the same generation of the search, the implicit rule is ambiguous and an error is reported. For example, if two of M's fields define *n*, *n* is ambiguous; but also, if M has two parents P1 and P2 which both have a field defining *n*, then *n* is ambiguous, and so on.

If a unique definition for *n* is found in any generation of the search, that definition is used. A warning is given if no definition is found; calling an undefined implicit rule will result in a run-time error.

There are some caveats. First, if any parent (grandparent, etc.) is closed off to implicit rule search by an explicit 'notusing all', neither that parent nor its supertypes are searched.

Second, only static rules (§5.1) are considered in imports, and only dynamic rules are considered in fields. Thus, there is no ambiguity in this example:

```
module I {
  dyn ::= ...;
  static stat ::= ...;
}
module M has I using all {
  field f :> I using all;
  test ::=
    dyn,  // unambiguously f.dyn (I.dyn not considered)
    stat; // unambiguously I.stat (f.stat not considered)
}
```

5.4.2 Implicit rule definitions

Once the compiler finds an unambiguous definition *D* for an implicit rule named *n*, it writes a forwarding definition for *n* which simply calls *D*. The forwarding definition depends on the type of *D*: specifically, the new definition takes the same number parameters with the same respective types as *D*, and returns the same type.

If *D* was found in an import *I*, the definition will look like this:

```
static n(parameters) ::= I.n(parameters);
```

If *D* was found in a field *f*, there are two possibilities, depending on whether *f* has pointer-to-module type:

```
n(parameters) ::= f.n(parameters);
n(parameters) ::= f->n(parameters);
```

5.4.3 Implicit rule module operators

The user controls implicit rule search through the using and notusing module operators.

5.4.4 'using'

The using operator makes a module's names available for implicit rule search. Its left operand must be a module expression; its right operand must be a list of simple names (qualified names can never be implicit rules, anyway), or either 'all' or 'allstatic'.

Here is a simple example:

```
module M {
  static r ::= ...;
}
module U has M {
  ... r ...  // error: undefined implicit rule
}
module U2 has M using r {
  ... r ...  // OK
}
```

'using all' makes all of a module's top-level names available for implicit rule search, while 'using allstatic' makes all of a module's *static* top-level names available for implicit rule search.

Note that any field module types are actually references to a module's import list. This can lead to more effective using directives than you want:

```
module U1 has M using all {
  field m1 :> M;
  field m2 :> M;
}
// is equivalent to...
```

```
module U2 has M using all {
  field m1 :> M using all;
  field m2 :> M using all;
}
```

Any reference to a dynamic implicit rule from M will result in an ambiguity between m1's definition and m2's. To fix this situation, use notusing, or say 'using allstatic' in the module header instead of 'using all'.

5.4.5 'notusing'

The notusing operator hides a module's names from implicit rule search. Its left operand must be a module expression; its right operand must be a list of simple names or 'all'. (You can't say 'notusing allstatic'.)

5.4.6 Notes

Once they are defined through the implicit rule search defined above, implicit rules are treated identically to normal rules by the language. In particular, this means that *implicit rules can be overridden* in a module's subtypes. This will not normally happen because implicit rules are hidden by default, but does enhance Prolac's flexibility.

5.5 Export specifications

Prolac does not, by default, generate a C function definition for every rule in the Prolac program; rather, *export specifications* tell Prolac which rules to generate. Export specifications are placed outside of any module in the Prolac input file. Here is the syntax for an export specification:

> export *module.rule* [, *module.rule* ...]

Module must be a module name (possibly preceded by namespace qualifiers); *rule* can be a rule from that module or 'all', which means "export all rules defined in *module*".

Prolac collects all export specifications and generates code for the rules they mention in arbitrary order. It then recursively generates code for all the rules they call, the rules those rules call, and so on, until it reaches closure.

6 Fields

Fields are essentially module-specific variables. Fields, like rules (§5.1), can be static or dynamic; dynamic fields are like instance variables or slots in other object-oriented languages. Each object of a module type has its own copy of each of the module's dynamic fields, while only one copy exists of each of a module's static fields. Any field, static or dynamic, must be part of some module.

Fields are declared with the following syntax:

> [static] field *name* :> *type*;

Remember that, if *type* is a module type, it must be a visible supertype of the current module or it must have been explicitly imported (§3.3).

A dynamic field can be referred to only in a dynamic context (§9.7); in a dynamic rule, 'self.' can be omitted when referring to self's dynamic fields. Note that, unlike rules, fields cannot be overridden[12] and they cannot be found with any kind of implicit search.[13]

7 Exceptions

Prolac exceptions are not yet implemented, although they are perhaps more necessary in Prolac than in many other languages. Prolac protocol specifications often involve deeply-nested calls of many small rules; a rule far down in the call stack may detect an error, in which case it would like to terminate processing immediately. It is tedious and error-prone to require any intermediate rules to detect and act on such a result.

Currently, our TCP specification uses the C *setjmp/longjmp* facility to fake exception handling; this makes our current TCP specification unsuitable for use as an in-kernel TCP implementation. Prolac exception handling, when designed and implemented, will not use *setjmp* or *longjmp*.

8 Types

This section describes the Prolac type system, including Prolac's built-in types and their values and allowable conversions between types.

We write that a value V has type T with the notation 'V :> T'; the type declaration operator ':>' (§9.8) is used in Prolac to declare the types of fields, parameters, rules, and supertypes. The expression 'V :> T' should be read "V is a T".

[12] One way to see why not is to think of a field as an abbreviation for two rules, a *setter* and a *getter*. For a field 'f :> M', these have signatures set-f(M) :> void and get-f :> M. Now, say we hypothetically overrode f to have type N. The overriding setter and getter have signatures set-f(N) :> void and get-f :> N; but these must be correct relative to the overridden signatures. If we apply the contravariance rules (§5.2.1), we see that this implies both M :> N and N :> M—therefore, M must equal N!

Interestingly, module operators—which create usefully different modules with the same effective type—imply that it might still be useful to override a field, *even if the effective type of the field is constrained to be the same!* We are considering this as an extension.

[13] The reason why not is mostly philosophical: Implicit rule search abbreviates something the user can already do (write forwarding rules). However, the user cannot create anything resembling a "forwarding field". If we introduce such a concept, we may add implicit field search as well.

8.1 Converting and casting

The two processes of *conversion* (or, equivalently, *implicit conversion*) and *casting* convert a value from one type to another. Casting is strictly more powerful than conversion.

Prolac automatically invokes implicit conversion in any context where a value is expected to be of some type—for example, as test argument to a choice operator '?:' (§9.4.5) must be of type bool. Integral values and pointer values both implicitly convert to bool, as defined below; therefore, these values are also acceptable as test arguments. The phrase "V is converted to T" means "V is implicitly converted to T if this is possible; if not, an error is reported."

Prolac never automatically casts a value; the user explicitly invokes a cast by using the type cast operator (§9.8). Whenever a value V can be implicitly converted to a type T, the explicit cast '(T)V' is also possible and has the same result.

8.2 Common types

Two types T1 and T2 always have a *common type*, which is used when T1 and T2 are combined in an expression; for example, the choice expression 'test ? V1 : V2', whose value may be either V1 :> T1 or V2 :> T2, has the common type of T1 and T2 as its type. Implicit conversions (never casts) are used to convert each operand to the common type.

The common type for two types T1 and T2 is found as follows:

1. If T1 and T2 are the same type T, the common type is T.
2. If either T1 or T2 is bool and the other can be implicitly converted to bool (i.e., it is bool or an integral or pointer type), the common type is bool.
3. If both T1 and T2 are integral types, the common type is the larger of them (see §8.5).
4. If both T1 and T2 are pointer types, then:
   (a) If either T1 or T2 is *void, the common type is *void.
   (b) If T1 and T2 are pointers to module types M1 and M2, then if either module is an supertype of the other, the pointer type to the supertype is returned.
5. Otherwise, the common type is void.

8.3 void

The void type signifies the absence of any value. Any expression can be implicitly converted to void. A void expression cannot be cast to any other type.

Because void implies the absence of a value, it is an error to declare a value (object, parameter, field, etc.) of type void. void is most useful as a rule return type and as the base for the generic pointer type *void.

8.4 bool

bool is the Boolean type. It has two values, true and false.

bool values may be implicitly converted from integral values; specifically, 0 converts to false, and any non-zero value converts to true. Pointer types also implicitly convert to bool: the null pointer converts to false, any non-null pointer to true.

bool values may be explicitly cast to integral types; false casts to 0 and true casts to 1.

8.5 Integral types

Prolac has nine integral types: four signed types, char, short, int, and long; four unsigned types, uchar, ushort, uint, and ulong; and one unsigned type with circular comparison (§9.11.4), seqint. Their properties are summarized in this table; **size** is in bits:

Size	**Signed**	**Unsigned**	**Circular**
8	char	uchar	
16	short	ushort	
32	int	uint	seqint
64	long	ulong	

The common type of two integral types is the larger of the two types, as defined below.

- If either type is ulong, the common type is ulong.
- Otherwise, if either type is long, the common type is long.
- Otherwise, if either type is seqint, the common type is seqint.
- Otherwise, if either type is uint, the common type is uint.
- Otherwise, the common type is int.

8.6 Pointer types

If T is a type, then *T is also a type, representing a pointer to a value of type T. (Note that Prolac differs syntactically from C, where the '*' operator attaches to the declared name, not the type.)

As in C, *void is the generic pointer type: any pointer can be implicitly converted to type *void, and an object of type *void can be implicitly converted to any pointer type.

A pointer to a module M can be implicitly converted to a pointer to a module S, where S is an supertype of M. Furthermore, a pointer to S can be explicitly cast to type *M.

The integer constant 0 can be implicitly converted to any pointer type, resulting in a *null pointer* of the given type. The semantics of such a null pointer are the same as in C. Note that, unlike C, any integer constant expression evaluating to 0 will not do.

8.7 Module types

Each defined module M is a distinct type. Note that two versions of the same module with different module operators (§3.4) do not define two different types; in terms of type, M is equivalent to 'M hide all inline x'.

An object of module type M may be implicitly converted to module type S, where S is one of M's supertypes.[14]

9 Expressions

This section describes the Prolac operators. Prolac is an expression-based language; unlike C, but like Lisp and ML, Prolac has no concept of a statement (i.e., a control structure which is not an expression). This property means that Prolac has even more operators and precedence levels than C (which, some might argue, already had too many); it also means that once you understand the Prolac operators, you can understand any computation expressed in Prolac.

Prolac operators fall into several categories. *Rule call* is the first to be discussed (§9.2), followed by *control flow operators*, which control Prolac's order of computation. The *let operator* (§9.5) and *C blocks* (§9.6), two special operators, come next, followed by *member operators* (§9.7), *typing operators* (§9.8), *code motion operators* (i.e., inline and outline) (§9.9), and, finally, *C operators*, whose meanings are the same in Prolac as in C (§9.11).

9.1 Operator precedence

Table 9.1 lists all of Prolac's operators and their precedences. Some operators do not have precedence—they textually contain all their subexpressions, and thus are never ambiguous; these are listed at the bottom of the table. Of course, grouping parentheses can be used to override any precedence or associativity.

Operators with precedence

23.	(t)x	type cast §9.8
22.	x.n .n	member §9.7
	x−>n	pointer to member §9.7
21.	f(x, y, ...)	rule call §9.2
	x[y]	array reference
	x++ x−−	postincrement, postdecrement
20.	module operators (§3.4):	
	M hide x M show x M rename (x=y)	namespace control §4.6
	M using x M notusing x	implicit rules §5.4.3
	M inline[n] x	inlining §9.9.1
19.	x :> t	type declaration §9.8
18.	*x	dereference
	&x	address of
	+x −x	unary plus/minus
	~x	bitwise not
	!x	logical not
	++x −−x	preincrement, predecrement
	inline[n] x	inlining §9.9.2
17.	x * y x / y x % y	multiply, divide, remainder
16.	x + y x − y	add, subtract
15.	x << y x >> y	left and right shift
14.	x < y x > y x <= y x >= y	arithmetic compare
13.	x == y x != y	equality tests
12.	x & y	bitwise and
11.	x ^ y	bitwise xor
10.	x | y	bitwise or
9.	x && y	logical and
8.	x || y	logical or
7.	x = y	assignment (R)

	compound assignment (R):
	x += y x −= y x *= y x /= y x %= y
	x &= y x ^= y x |= y x <<= y x >>= y	

6.	x ? y : z	choice (R) §9.4.5
5.	x {C} y	C block §9.6
4.	x ==> y	arrow (R) §9.4.3
3.	x ||| y	case §9.4.7
2.	outline[n] x	outlining §9.9.3
1.	x, y	comma §9.4.1

Operators without precedence

(x)	grouping
if x then y else z end	if-then-else §9.4.6
let decls in body end	let §9.5

Table 9.1: Prolac operators and precedence levels. Operators higher in the table bind more tightly; (R) denotes right-associative operators. If a reference is not given, the operator is described in §9.11.

[14] As of this writing, *prolacc* does not completely implement this conversion; for example, the assignment 'let s :> S, m :> M in s = m end' is not generated correctly.

9.2 Rule calls

A rule call expression '*r*(...)' expresses the execution of the specified rule. Any *actual parameters* to the rule are given inside the parentheses, separated by commas; they must match the rule's declared parameters in number and type. The types need not match exactly: Prolac attempts to convert the actual parameters to the types of the declared parameters.

Calls to rules without parameters need not provide parentheses. For example:

```
module M {
 r ::= true;
 s ::= r(),  // call M.r
    r;  // also call M.r
}
```

9.2.1 Namespace call

To facilitate the use of namespaces, a namespace name may be treated as a rule call. If a namespace originally named *n*[15] is found within an expression where namespaces are not expected, Prolac looks in that namespace for a rule named *n* and uses that if it is found. For example:

```
module M {
 nest { nest ::= ...; }
 r ::= nest.nest,  // call M.nest.nest
    nest;  // also call M.nest.nest
}
```

Note that the search is only performed one level deep. Thus, this is an error:

```
module M {
 nest { nest { nest ::= found-the-prize; } }
 r ::= nest;  // error:
    // namespace 'M.nest' cannot be called
}
```

In a normal Prolac expression (that is, any expression except supertype and import lists and operands to module operators), namespaces are expected in only one context: to the left of a direct member operator '.' (§9.7). Thus, this search is performed everywhere except to the left of '.'.

9.2.2 Constructor calls

Constructors (§5.3) can be called indirectly by Prolac, or directly by the user. Most constructor calls are indirect; direct constructor calls are useful to initialize a block of memory allocated by C code as a Prolac object.

Direct constructor calls. To call a constructor directly, simply treat it as a normal rule. For example:

```
module M {
 constructor(i :> int) ::= ...;
 static new :> *M ::= let ptr :> *M in
   { ptr = malloc(sizeof(M)); }
   ptr->constructor(97),  // direct constructor
   ptr end;
}
```

Indirect constructor calls. A module's constructor is called whenever an object of that module type, or one of its subtypes, is created. This happens within let expressions (§9.5) when the value for a variable is not given, and whenever a module type is used as a value expression (§9.3).

An indirect constructor call for a module M looks like M(*args*). Just as with rule calls, the actual arguments to a constructor call must match the constructor's declared parameters in number and type. To illustrate:

```
module M {
 constructor(n :> int) ::= ...;
 r ::= let m :> M,  // error:
    // 'M's constructor requires 1 argument
    m2 :> M(97)  // OK
 in false end;
}
```

As with rule calls, the parentheses can be omitted from a constructor call when the constructor takes no arguments. This leads to an ambiguity between treating module names as type names and as calls to constructors with omitted parentheses. For example:

```
module N has M {
  ... M.r ...  // calling import M's static rule r,
  // or calling dynamic rule r in a newly constructed M?
}
```

This issue is discussed in more detail below (§9.3). To summarize that discussion, Prolac assumes that such an expression is a type name wherever a type name might be meaningful. Thus, M.r above is resolved to calling M's static rule r. (This resolution does not depend on whether or not M.r is actually static, or whether or not M's constructor takes no arguments.) To force the other interpretation, simply leave in the parentheses: 'M().r'.

It is an error to call a constructor, directly or indirectly, if the constructor has been hidden (§4.6.1). This technique allows a module to suggest[16] that no objects of that module type be created except through visible interface functions.

[15] I.e., before any show or rename operations.

[16] A user of the module can always show the constructor.

9.3 Type expressions and value expressions

Prolac expressions come in two flavors: those that name types and those that name values. Some expressions, especially module names, can be either type or value expressions; which form is used depends on the context in which the expression is found.

Here are some examples of type and value expressions:

```
module M {
  static x ::= ...;
}
module N has M {
  field f :> M;
  test ::=
    int,  // type
    * int,  // type
    75,  // value
    M,  // type or value (here, value)
    M.x,  // value; M is type
    f = M;  // value; M is value
}
```

The expression 'M' can be interpreted in two ways: either as naming the module M, or as creating an object of module M via a call to M's constructor. Given an expression E which can be either type or value, Prolac resolves E as follows:

1. If E is in a context where a type *can be* expected (for example, as the argument to unary '*' (§9.11.1) or one of the module operators (§3.4), or as the second argument to the type declaration operator ':>' (§9.8)), E is a type expression.

2. Otherwise, if E is in the context E.name, E is a type expression.

3. Otherwise, E is a value expression.

9.4 Control flow operators

Control flow operators are semistrict: they do not always evaluate all of their operands. Different control flow operators express sequencing, choice or if-then-else semantics, conjunction, disjunction, conditional execution, and case statements.

9.4.1 Comma: ','

The comma operator ',' expresses sequencing: an expression 'A, B' first evaluates A, then throws the result away and returns the result of B.

A and B can each have any type. The expression's type is the type of B.

9.4.2 Logical and: '&&'

The logical and operator '&&' expresses conjunction: an expression 'A && B' evaluates to true iff both A and B are true. A is evaluated first; if it is false, the whole expression must be false, and B is not evaluated at all.

Both A and B are converted to bool. The expression also has type bool.

9.4.3 Arrow: '==>'

The arrow operator '==>' expresses conditional execution. An expression 'A ==> B' evaluates to true iff A is true—but in that case, B is evaluated before the expression returns.

A is converted to bool, but B can have any type. The expression has type bool (but see below §9.4.7).

This example demonstrates the difference between && and ==>:

```
true && false  // result: false
true ==> false  // result: true
```

This behavior is useful for building "case statements"; for example,

```
(condition-1 ==> case-1)
|| (condition-2 ==> case-2)
|| else-case
```

This code will evaluate else-case only if condition-1 and condition-2 are both false. At most one of case-1, case-2, and else-case will be executed. (For a case statement which in addition returns a useful value, see §9.4.7.)

Contrast this with a version using &&:

```
(condition-1 && case-1)
|| (condition-2 && case-2)
|| else-case
```

This is both more opaque and probably incorrect: if condition-1 is true, *but case-1 returns false,* the remaining two clauses will still be executed.

The expression 'A ==> B' is exactly equivalent to 'A && (B, true)', except for its behavior within case bars (§9.4.7).

9.4.4 Logical or: '||'

The logical or operator '||' expresses disjunction: an expression 'A || B' evaluates to true iff either A is true, B is true, or both. A is evaluated first; if it is true, the whole expression must be true, and B is not evaluated at all.

Both A and B are converted to bool. The expression also has type bool.

The bool return type of || hides information from its subexpressions; this may result in overly convoluted programs. The case bar operator '|||' addresses this issue (§9.4.7).

9.4.5 Choice: '?:'

The question mark–colon operator '?:'—also called the choice operator—expresses choice. An expression 'A ? B : C' first evaluates A. If A is true, it returns B (without evaluating C); otherwise, it returns C (without evaluating B).

A is converted to bool; the type of the expression is the common type of B and C.

9.4.6 'if-then-else'

The if-then-else operator is another way to express choice. The full syntax for if-then-else is as follows:

```
if condition
then case–1
[elseif condition–2 then case–2]...
[else else–case]
end
```

This expression is a synonym for:

```
condition ? case–1
  [: condition–2 ? case–2]...
  : else–case
```

The type of the if-then-else expression is the common type of all *cases*. The *else–case* can be omitted, forming an if-then expression; if it is omitted, the type of the expression is void.

'if–then–else' is provided primarily as an alternative to '?:' for larger expressions—the '?:' syntax becomes difficult to read very quickly when its operands are large.

9.4.7 Case bars: '|||'

The case bar operator '|||', in conjunction with the arrow operator (§9.4.3), expresses a case statement returning a meaningful value.

A case statement has this general form:

```
condition–1 ==> consequent–1
||| condition–2 ==> consequent–2 ...
||| else–case
```

Exactly one of the *consequents* or *else–case* is executed, depending on which, if any, *condition* is true. The result of that *consequent* or *else–case* is returned as the value of the expression.

The case statement's syntax is based on matching constructs from functional programming languages; here, however, the *conditions* are all converted to bool. The case statement is exactly analogous to Lisp's cond special form.

Case bars are actually syntactic sugar for choice operators (§9.4.5). Given an expression containing '|||', the compiler repeatedly applies the following transformations until no '|||'s remain:

```
1. A ==> B ||| X      ⇒  A ? B : X
2. (A ? B : C) ||| X  ⇒  A ? B : (C ||| X)
3. A ||| X            ⇒  A || X
```

Note that '|||'s used outside the context of a case statement reduce to normal logical ors, '||' (§9.4.4).

Here is a demonstration of the rules:

```
A ==> B ||| C ==> D ||| E
⇒ (A ==> B ||| C ==> D) ||| E  // left-associative
⇒ (A ? B : (C ==> D)) ||| E  // rule 1
⇒ A ? B : ((C ==> D) ||| E)  // rule 2
⇒ A ? B : (C ? D : E)  // rule 1
```

The type of the whole expression is therefore the common type of B, D, and E.

A case statement without a final *else–case* usually has type bool. To see why, consider this expansion:

```
A ==> B ||| C ==> D
⇒ A ? B : (C ==> D)  // rule 1
```

Since there are no case bars remaining, expansion is over. The type of the expression is then the common type of B and 'C ==> D'; but the type of 'C ==> D' is just bool, so B will be converted to bool if possible.

9.5 'let'

The let operator, like the let operator in many functional languages, introduces new statically-bound variables within a subexpression. The syntax of let is as follows:

```
let variable [:> type] [= value]
   [, variable [:> type] [= value] ...]
in body end
```

Variable is just an identifier. *Type* is a type expression and *value* is a value expression; either *type* or *value* may be omitted, but not both. If *value* is omitted, *variable* is constructed implicitly if it has module type (§9.2.2), or left uninitialized otherwise; if *type* is omitted, *variable*'s type is the type of the *value* expression.

To evaluate a let expression, Prolac first evaluates the *value* expressions and any necessary *type* constructors in an arbitrary order. The resulting values are then bound to the *variables*. Finally, *body* is evaluated with these bindings in force. The value of the let expression is the value of *body*; the type of the let expression is the type of *body*. Note that a let expression's variables are not visible to any of its *type* or *value* expressions.

9.6 C blocks: '{...}'

A C block '{...}' is used to call C code at a given point during the execution of a rule. The type of a C block is bool and its value is always true.

C blocks are syntactically rather special: a C block acts like a Prolac value—specifically, 'true'—with "implicit commas" on either side. These implicit commas are equivalent to the comma operator (§9.4.1), but have different precedence—specifically, just above the arrow operator (§9.4.3). Some examples will make things clearer; the implicit true is shown when necessary.

```
{A}             ≡  {A}, true
{B} X           ≡  {B}, X
X {C}           ≡  X, {C}, true
X {D} Y         ≡  X, {D}, Y
X ==> {E}       ≡  X ==> ({E}, true)
X ==> Y {F}     ≡  X ==> (Y, {F}, true)
X = Y {G} Z     ≡  (X = Y), {G}, Z
X ? Y : Z {H}   ≡  (X ? Y : Z), {H}, true
```

Of course, parentheses can be used to override C blocks' precedence.

Some Prolac names can be used in a C block to refer to the C equivalents of those Prolac objects. Specifically, the following objects are available under their Prolac names:

1. In a dynamic rule, self.
2. In a dynamic rule, any of self's fields accessible by simple names.
3. Parameters from the current rule.
4. Variables bound by surrounding let expressions (§9.5).

Note that this list does not include Prolac rules: you cannot call Prolac rules from C blocks using Prolac syntax. Also note that if you need to use a member name to refer to some object in Prolac, you can't refer to it in a C block; only simple names will work.

Finally, C and Prolac treat hyphens differently (§2.2): C does not allow hyphens in identifiers. Prolac follows C's rules while parsing C blocks; thus, this C block will not work as expected:[17]

```
let thing-1 = 0 in { return thing-1; } end
```

To refer to an object with a hyphen in its name, simply change the hyphen to an underscore:

```
let thing-1 = 0 in { return thing_1; } end
```

9.7 Member operators: '.' and '->'

The member operators '.' and '->' express finding a feature in a namespace or object. The right operand of a member operator must be an identifier. The pointer-to-member operator '->' is used on pointer types (§8.6); the expression 'A->x' is exactly equivalent to '(*A).x'. The rest of this section only discusses the direct-member operator, '.'.

The left-hand operand, or "object operand", of a member expression must be a namespace or have a module type. The object operand may be either a type or value expression. If it is a type expression, the member has *static context;* if it is a value expression, it has *dynamic context*. It is an error to refer to a static feature in a dynamic context, or a dynamic feature in a static context. To illustrate:

```
module M {
  field d :> int;
  static s ::= 0;
}
module N has M {
  field m-object :> M;
  test ::=
    M.s,  // OK: static context, static rule
    m-object.d,  // OK: dynamic context, dynamic field
    M.d,  // error: static context, dynamic field
    m-object.s;  // error: dynamic context, static rule
}
```

Fields, parameters, and objects are always value expressions, and thus always have dynamic context. Imports are always type expressions, and thus always have static context. Supertypes are a special case: In a static rule, supertypes are type expressions and have static context. In a dynamic rule, supertypes are value expressions and have dynamic context; however, in this case and this case only, you may refer to an supertype's static feature, even though the supertype has dynamic context.

As discussed above (§9.3), ambiguous object operands are resolved to type expressions, and thus static context. To force such an object operand to dynamic context, use parentheses to tell Prolac to call the operand's constructor:

```
module M {
  field d :> int;
  constructor ::= ...;
}
module N has M { ...
  M.d,  // error: dynamic field in static context
  M().d;  // OK: parentheses make it a constructor call
}
```

The direct member operator '.' also has a prefix version '.x', used to look up names in a global namespace (§4.2.1).

[17] Or maybe it is the Prolac code that does not work as expected!

9.8 Type operators: ':>' and '(cast)'

The type declaration operator ':>' is used elsewhere in Prolac to declare the types of objects; inside a value expression, ':>' expresses a type assertion. The right operand of ':>' should be a type expression, T. The value of an expression 'V :> T' is the value of V converted (§8.1) to type T. Note that the ':>' operator will only use implicit conversions on V; an error is given if V cannot be implicitly converted to T. Thus, ':>' can be used to guarantee that V has type T without invoking a possibly dangerous type cast.

The type casting operator '(*type*)' is used to change the type of its value operand. The value of an expression '(T)V', where T is a type expression, is the value of V cast (§8.1) to type T.

9.9 Code motion operators

Prolac provides two operators to control optimization and code motion, inline and outline. inline is also available as a module operator.

9.9.1 'inline' module operator

The inline module operator controls how a module's rules are inlined. Its left operand must be a module expression; its right operand must be a list whose elements are one of the following kinds of expressions:

1. A simple rule name, or the name of a namespace that can be called (§9.2.1); the corresponding rule will be affected.

2. 'NS.all', where NS is a namespace; all rules defined in NS or any of its nested namespaces will be affected.

3. 'M.all', where M is a supertype of the module expression; all rules defined by M (i.e., whose actual is M) will be affected.

4. 'all'; all rules will be affected.

In addition, inline can take an optional *inline level* argument, which must appear in brackets directly after the inline keyword. An inline level is a static integer constant[18] whose value must be between 0 and 10, where 0 means "never inline" and 10 means "always inline". If no optional argument is given, the corresponding inline level is 5. The default inline level for a rule (if no inline operator has been applied) is 2. A Prolac compiler may provide an option determining which inline levels actually result in inlined rules; normally, the default level of 5 will result in inlined rules.

When a rule call expression (§9.2) is evaluated, Prolac checks the corresponding module for that rule's inline level; if it is high enough and the rule call is unambiguous (i.e., no dynamic dispatch is possible), the rule call will be inlined. For example:

```
module M {
  rule ::= ...;
}
module N has M {
  test(r1 :> M, r2 :> M inline all) ::=
    r1.rule,  // not inlined
    r2.rule;  // inlined
}
```

9.9.2 Expression 'inline'

Expression inline is a prefix unary operator appearing in rule bodies, while the inline module operator is a binary operator appearing in module specifications. Like the inline module operator, expression inline can take an optional inline level argument in brackets.

An expression 'inline[*n*] X' simply evaluates X and returns its value; the type of 'inline[*n*] X' is the type of X. However, any rule calls within X are inlined with inline level *n*. An expression inline overrides all relevant inline module operators.

An expression inline with a high enough inline level (9 or 10 in the current implementation) will also cause nested calls to be inlined; for example:

```
module M {
  a ::= { XXX; };
  b ::= a;
  d ::= inline[10] b;
  // generates code equivalent to 'd ::= { XXX; };'
  // rather than 'd ::= a;'
}
```

This recursive inlining can be stopped by expression inlines in the inlined rules' bodies. Recursive or mutually recursive rules are currently inlined only one level deep; any recursive calls are generated as procedure calls.

9.9.3 'outline'

outline is a prefix unary operator appearing in rule bodies; it controls *code outlining*, or the removal of infrequently-executed code from a computation's critical path. The outline operator tells the compiler that the current branch of control flow is relatively unlikely; the compiler will move

[18] While we intend eventually to allow symbolic constants to be used in inline levels, they have presented implementation difficulties that far outstrip their immediate utility. *Prolacc* currently accepts inline levels which are integer literals or integer literal expressions; rule calls sometimes work, but most often do not due to order dependencies.

that branch to the end of the function body in the code it generates.

Like the inline operators, outline it takes an optional static integer constant argument which must be between 0 and 10. Here, 0 means "never outline", while 10 means "outline as far as possible"; or, equivalently, 0 means "this is the most common branch" and 10 means "this is the least common branch".

An expression 'outline[*n*] X' evaluates X and returns its value; its type is the type of X.

The outline operator is only meaningful to the right of a control flow operator (§9.4).[19] To see why, consider the expression 'A + (outline B)'. This expression suggests that B is less likely to be executed than A; but this is impossible, since (unless A generates an exception) B will be executed whenever A is.[20] Prolac does not warn on such expressions; rather, it floats the outline into the expression until it finds a control flow operator, and attaches the outline onto that operator's right operand. Thus, these pairs of expressions are equivalent:

```
A + (outline B ? C : D)  ≡  A + (B ? C : (outline D))
outline A || B           ≡  A || (outline B)
```

9.10 Lvalues

Some expressions are *lvalues*, meaning that they can appear on the *l*eft side of an assignment expression. Only the following expressions are lvalues:

1. Field (§6), parameter, or let-bound variable (§9.5) references;
2. Dereferences '*X', where X is an object of pointer type (§9.11.1);
3. Assignments 'X = Y' (§9.11.8);
4. Compound assignments 'X @= Y' where @ is a binary arithmetic operator (§9.11.9);
5. Increment or decrement expressions 'X++', 'X--', '++X', '--X' (§9.11.10);
6. '(X)', where X is an lvalue.

9.11 C operators

This section describes the remaining operators, whose definitions are generally borrowed from C.

9.11.1 Dereference: unary '*'

The star or dereference operator '*' acts differently depending on whether its operand X is a type or value expression.

If X is a value expression, then it must have some pointer type *T, but not *void. The expression '*X' then has type T; its value is the value of the object to which X points.

If X is a type expression, then '*X' is also a type expression defining the type "pointer to X" (§8.6).

9.11.2 Address of: unary '&'

The operand in an address expression '&X' must be an lvalue (§9.10). The value of the expression is a pointer to X; it has type '*T', where T is the type of X.

9.11.3 Equality tests: '==', '!='

Any two non-module values may be compared for equality. In an expression 'X == Y', X and Y are both converted to their common type (§8.2), which must not be void; as a special case, any pointer can be compared with the integer constant 0, which is converted to a null pointer. The result has type bool, and is true iff X and Y are equal.

The expression 'X != Y' is a synonym for '!(X == Y)'.

9.11.4 Arithmetic compare: '<', '<=', '>', '>='

The operands to an arithmetic compare operation are converted to their common type, which must be an integral or pointer type. Any integer value can be compared to any other, and two pointers of the same type can be compared. As a special case, any pointer can be compared with the integer constant 0, which is converted to a null pointer (§8.6). An arithmetic compare expression has type bool.

If the operands have type seqint, a special circular compare is performed which provides better results on overflow than normal compare. To illustrate, consider the unsigned integer expression 'L < H' where L = 4294967294 and H = 4294967295: L and H have values close to one another, but also close to 2^{32}. This comparison is true, but all we need to do is add one to H to make it false; H will wrap around to 0, and 4294967294 < 0 is obviously false. Circular comparison does not exhibit this flipping behavior until the difference between L and H is very large ($2^{31}-1$, to be exact), and is therefore more useful in certain contexts, such as TCP sequence numbers.

The circular comparison operators are defined as follows:

```
x < y   ≡  (int)(x – y) < 0
x <= y  ≡  (int)(x – y) <= 0
x > y   ≡  (int)(x – y) > 0
x >= y  ≡  (int)(x – y) >= 0
```

Again, these definitions only come into play when x and y have type seqint.

[19] It is meaningful on any consequent of any of the choice operators (§9.4.5, §9.4.6, §9.4.7), but not on the test.

[20] Strictly speaking, the comma operator (§9.4.1) has the same property.

9.11.5 Logical not: ‘!’

The operand to a logical not expression ‘!X’ is converted to bool; the expression has type bool. If the value of X is true, the expression has value false, and vice versa.

9.11.6 Arithmetic operators

The binary arithmetic operators are addition ‘+’, subtraction ‘–’, multiplication ‘*’, division ‘/’, remainder ‘%’, left ‘<<’ and right ‘>>’ shift, bitwise and ‘&’, bitwise or ‘|’, and bitwise exclusive or ‘^’. The unary arithmetic operators are unary plus ‘+’ and minus ‘–’ and bitwise not ‘~’.

The operands to most arithmetic operators must have integral type. The type of a unary arithmetic expression is the type of its operand; the type of a binary arithmetic expression is the common type of its operands.

Binary addition and subtraction also support some combinations of pointer operands. In an addition expression ‘A + B’:

- Either A or B may be a pointer; the other must have integral type. The result has the type of the pointer operand.

In a subtraction expression ‘A – B’:

- A and B may be pointers of the same type. The result has type int.
- A may have pointer type and B may have integral type. The result’s type is the type of A.

All arithmetic operators behave as they do in C.

9.11.7 Array reference: ‘[]’

The bracket operator ‘[]’ expresses array reference. As in C, an expression ‘X[Y]’ is exactly equivalent to the expression ‘*(X + Y)’. Note that array reference is not yet fully functional in Prolac, in that array *types* do not currently exist; however, the bracket syntax is still useful when one of X and Y is a pointer.

9.11.8 Assignment: ‘=’

The assignment operator ‘=’ expresses variable assignment. In an expression ‘X = Y’, X must be an lvalue (§9.10); Y is converted to the type of X. The expression has the type of X; its value is the value of X after the assignment is performed.

9.11.9 Compound assignment

A compound assignment expression ‘X @= Y’, where @ is a binary arithmetic operator (§9.11.6), is exactly equivalent to the expression ‘X = X @ Y’ except that X is evaluated only once.

9.11.10 Increment and decrement: ‘++’, ‘––’

The increment and decrement operators ‘++’ and ‘––’ are used to increment or decrement an lvalue by 1. Their operand must be an lvalue with arithmetic or pointer type; the result of the expression has the same type. The following table shows equivalent expressions for each increment and decrement operator, except that postfix increment and decrement evaluate their operand only once.

```
++X  ≡  X += 1
––X  ≡  X –= 1
X++  ≡  let temp = X in X += 1, temp end
X––  ≡  let temp = X in X –= 1, temp end
```

BIBLIOGRAPHY

[And88] David P. Anderson. Automated protocol implementation with RTAG. *IEEE Transactions on Software Engineering*, 14(3):291–300, March 1988.

[AP93] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, February 1993.

[BB89] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. In Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz, editors, *The formal description technique LOTOS*, pages 23–73. North-Holland, 1989.

[BdS91] Frédéric Boussinot and Robert de Simone. The ESTEREL language. Technical Report 1487, INRIA Sophia-Antipolis, July 1991.

[BOP94] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM 1994 Conference*, pages 24–35, August 1994.

[Cas96] Claude Castelluccia. Automating header prediction. In *Workshop record of WCSSS'96: The inaugural workshop on compiler support for systems software*, pages 44–53, February 1996.

[CDO96] Claude Castelluccia, Walid Dabbous, and Sean O'Malley. Generating efficient protocol code from an abstract specification. In *Proceedings of the ACM SIGCOMM 1996 Conference*, pages 60–71, August 1996.

[Cla82] David D. Clark. Modularity and efficiency in protocol implementation. RFC 817, IETF, July 1982.

[CT90] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM 1990 Conference*, pages 200–208, September 1990.

[DB89] P. Dembinski and S. Budkowski. Specification language Estelle. In Michel Diaz, Jean-Pierre Ansart, Jean-Pierre Courtiat, Pierre Azema, and Vijaya Chari, editors, *The formal description technique Estelle*, pages 35–75. North-Holland, 1989.

[HA89] Diane Hernek and David P. Anderson. Efficient automated protocol implementation using RTAG. Report UCB/CSD 89/526, University of California at Berkeley, August 1989.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[Joh75] Stephen C. Johnson. Yacc—Yet Another Compiler-Compiler. Comp. Sci. Tech. Rep. #32, Bell Laboratories, July 1975. Reprinted as PS1:15 in *Unix Programmer's Manual*, Usenix Association, 1986.

[LMD90] Luigi Logrippo, Tim Melanchuk, and Robert J. Du Wors. The algebraic specification language LOTOS: an industrial experience. In Mark Moriconi, editor, *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, pages 59–66, September 1990.

[Mey92] Bertrand Meyer. *Eiffel: the language*. Prentice Hall, 1992.

[Mil80] Robin Milner. *A calculus of communicating systems*, volume 92 of *Lecture notes in computer science*. Springer-Verlag, 1980.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pages 259–269, San Diego, Winter 1993. USENIX.

[MPBO96] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of techniques to improve protocol processing latency. In *Proceedings of the ACM SIGCOMM 1996 Conference*, pages 73–84, August 1996.

[Pos81] Jon Postel. Transmission Control Protocol: DARPA Internet Program protocol specification. RFC 793, IETF, September 1981.

[SB90] Deepinder P. Sidhu and Thomas P. Blumer. Semi-automatic implementation of OSI protocols. Technical Report CS-TR-2391, University of Maryland at College Park, January 1990.

[SCB90] Deepinder Sidhu, Anthony Chung, and Thomas P. Blumer. A formal description technique for protocol engineering. Technical Report CS-TR-2505, University of Maryland at College Park, July 1990.

[Ste97] W. Richard Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Rfc, IETF, January 1997.

[Str94] Bjarne Stroustrup. *The design and evolution of C++*. Addison-Wesley, 1994.

[vB86] Gregor v. Bochmann. Methods and tools for the design and validation of protocol specifications and implementations. Publication #596, Université de Montréal, October 1986.

[WEK96] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: application-specific handlers for high-performance messaging. In *Proceedings of the ACM SIGCOMM 1996 Conference*, pages 40–52, August 1996.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

[WvB89] Cheng Wu and Gregor v. Bochmann. An execution model for LOTOS specifications. Publication #701, Université de Montréal, October 1989.