The Cecil Language

Specification and Rationale

Version 3.1

Craig Chambers and the Cecil Group

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, Washington 98195-2350 USA

December, 2002

Abstract

Cecil is a purely object-oriented language intended to support rapid construction of high-quality, extensible software. Cecil combines multi-methods with a simple classless object model, a kind of dynamic inheritance, modules, and optional static type checking. Instance variables in Cecil are accessed solely through messages, allowing instance variables to be replaced or overridden by methods and vice versa. Cecil's predicate objects mechanism allows an object to be classified automatically based on its run-time (mutable) state. Cecil's static type system distinguishes between subtyping and code inheritance, but Cecil enables these two graphs to be described with a single set of declarations, streamlining the common case where the two graphs are parallel. Cecil includes a fairly flexible form of parameterization, including explicitly parameterized objects, types, and methods, as well as implicitly parameterized methods related to the polymorphic functions commonly found in functional languages. By making type declarations optional, Cecil aims to allow mixing of and migration between exploratory and production programming styles. Cecil supports a module mechanism that enables independently-developed subsystems to be encapsulated, allowing them to be type-checked and reasoned about in isolation despite the presence of multi-methods and subclassing. Objects can be extended externally with additional methods and instance variables, often encapsulated in separate modules, supporting a kind of role-based or subject-oriented programming style.

This document mixes the specification of the language with discussions of design issues and explanations of the reasoning that led to various design decisions.

Table of Contents

1 Introduction

This document describes the current design of Cecil, an object-oriented language intended to support the rapid construction of high-quality, reusable, extensible software systems [Chambers 92b, Chambers 93b, Chambers & Leavens 94]. Cecil is unusual in combining a pure, classless object model, multiple dispatching (multi-methods), modules, and mixed static and dynamic type checking. Cecil was inspired initially by Self [Ungar & Smith 87, Hölzle *et al.* 91a], CLOS [Bobrow *et al.* 88, Gabriel *et al.* 91], and Trellis [Schaffert *et al.* 85, Schaffert *et al.* 86]. The current version of Cecil extends the earlier version [Chambers 93a] with predicate objects, modules, and efficient typechecking algorithms.

1.1 Design Goals and Major Features

Cecil's design results from several goals:

- *Maximize the programmer's ability to develop software quickly and to reuse and modify existing software easily.*

  In response to this goal, Cecil is based on a pure object model: all data are objects and objects are manipulated solely by passing messages. A pure object model ensures that the power of object-oriented programming is uniformly available for all data and all parts of programs. The run-time performance disadvantage traditionally associated with pure object-oriented languages is diminishing with the advent of advanced implementations.

  Our experience also leads us to develop a classless (prototype-based) object model for Cecil. We feel that a classless object model is simpler and more powerful than traditional class-based object models. Cecil's object model is somewhat more restricted than those in other prototype-based languages [Borning 86, Lieberman 86, LaLonde *et al.* 86, Ungar & Smith 87, Lieberman *et al.* 87], in response to other design goals.

  Since message passing is the cornerstone of the power of object-oriented systems, Cecil includes a fairly general form of dynamic binding based on multiple dispatching. Multi-methods affect many aspects of the rest of the language design, and much of the research on Cecil aims to combine multi-methods with traditional object-oriented language concepts, such as encapsulation and static type checking, not found in other multiple dispatching languages.

  Inheritance also plays a key role in organizing software and factoring out commonalities. Cecil extends traditional inheritance mechanisms with predicate objects to support an automatic form of classification of objects into specialized subclasses based on their run-time state. Since this state can be mutable, an object's classification can change over time. This mechanism enables inheritance and classification to be applied even when modelling time-varying properties of an object. For example, a rectangle can be automatically classified as the predicate subobject square whenever it satisfies the predicate that its length equals its width, even if the rectangle's length and width are mutable.

  Instance variables (called fields in Cecil) are also accessed solely by sending messages, enabling fields to be replaced or even overridden with methods, and vice versa, without affecting clients. Fields can be given default initial values as part of their declaration. An initialization expression is evaluated lazily, when the field is first referenced, acting as a kind of memoized constant function. By allowing the initialization expression to reference the object that it will become a part of, circular data structures can be constructed, and more generally, the value of one field can be computed from the values of other fields of an object.

• *Support production of high-quality, reliable software.*

To help in the construction of high-quality programs, programmers can add statically-checkable declarations and assertions to Cecil programs. One important kind of static declaration specifies the types of (i.e., the interfaces to) objects and methods. Cecil allows programmers to specify the types of method arguments, results, and local variables, and Cecil performs type checking statically when a statically-typed expression is assigned to a statically-typed variable or formal argument. The types specified by programmers describe the minimal *interfaces* required of legal objects, not their *representations* or *implementations*, to support maximum reuse of typed code. In Cecil, the subtype graph is distinguished from the code inheritance graph, since type checking has different goals and requirements than have code reuse and module extension [Snyder 86, Halbert & O'Brien 86, Cook *et al.* 90].

To support the independent construction of subsystems, Cecil includes a module system. A module encapsulates its internal implementation details and presents an interface to external clients. This encapsulation mechanism is specially designed to work in the presence of multi-methods and inheritance/subtyping across module boundaries. Modules can be used to encapsulate "roles" [Andersen & Reenskaug 92] or "subjects" [Harrison & Ossher 93], programming idioms where pieces of the total interface of an object are split apart into application-specific facets. A given module can include method and field declarations that extend one or more previously-defined objects with additional specialized state and behavior.

Cecil includes other kinds of static declarations. An object can be annotated as an abstract object (providing shared behavior but not manipulable by programs), as a template object (providing behavior suitable for direct instantiation but otherwise not manipulable by the program), or as a concrete object (fully manipulable and instantiated as is). Object annotations inform the type checker how the programmer intends to use objects, enabling the type checker to be more flexible for objects used in only a limited fashion.

Cecil encourages a functional programming style by default, as this is likely to be easier to understand and more robust in the face of programming changes. By default, both local variables and fields are initialize-only; an explicit `var` keyword is required to assert that a variable or field can be mutated. An object can be created and its fields initialized to desired values in a single atomic operation; there are no partially-initialized states as are found during execution of a constructor in C++.

Finally, Cecil omits certain complex language features that can have the effect of masking programming errors. For example, in Cecil, multiple dispatching and multiple inheritance are both *unbiased* with respect to argument order and parent order; any resulting ambiguities are reported back to the programmer as potential errors. This design decision is squarely at odds with the decision in CLOS and related languages. Additionally, subtyping in Cecil is explicit rather than implicit, so that the behavioral specification information implied by types can be incorporated into the decision about whether one type is a behavioral subtype of another.

- *Support both exploratory programming and production programming, and enable smooth migration of parts of programs from one style to the other.*

  Central to achieving this goal in Cecil is the ability to omit type declarations and other annotations in initial exploratory versions of a subsystem and incrementally add annotations as the subsystem matures to production quality. Cecil's type system is intended to be flexible and expressive, so that type declarations can be added to an existing dynamically-typed program and achieve static type correctness without major reorganization of the program. In particular, objects, types, and methods may be explicitly parameterized by types, method argument and result types may be declared as or parameterized by implicitly-bound type variables to achieve polymorphic function definitions, and (as mentioned above) the subtype graph can differ from the inheritance graph. The presence of multiple dispatching relieves some of the type system's burden, since multiple dispatching supports in a type-safe manner what would be considered unsafe covariant method redefinition in a single-dispatching language.

  Additionally, an environment for Cecil could infer on demand some parts of programs that otherwise must be explicitly declared, such as the list of supertypes of an object or the set of legal abstract methods of an object, so that one language can support both exploratory programmers (who use the inferencer) and production programmers (who explicitly specify what they want). This approach resolves some of the tension between language features in support of exploratory programming and features in support of production programming. In some cases, the language supports the more explicit production-oriented feature directly, with an environment expected to provide additional support for the exploratory-oriented feature.

- *Avoid unnecessary redundancy in programs.*

  To avoid requiring the programmer to repeat specifying the interface of an object or method, Cecil allows a single object declaration to define both an implementation and a type (an interface). Similarly, where the subtype hierarchy coincides with the code inheritance hierarchy, a single declaration will establish both relations. This approach greatly reduces the amount of code that otherwise would be required in a system that distinguished subtyping and code inheritance. Without this degree of conciseness, we believe separating subtyping from code inheritance would be impracticably verbose.

  Similarly, Cecil's classless object model is designed so that a single object declaration can define an entire data type. This contrasts with the situation in Self, where two objects are needed to define most data types [Ungar *et al.* 91]. Similarly, Cecil's object model supports both concise inheritance of representation and concise overriding of representation, unlike most class-based object-oriented languages which only support the former and most classless object-oriented languages which only conveniently support the latter.

  Finally, Cecil avoids requiring annotations for exploratory programming. Annotations such as type declarations and privacy declarations are simply omitted when programming in exploratory mode. If this were not the case, the language would likely be too verbose for rapid exploratory programming.

- *Be "as simple as possible but no simpler."*

  Cecil attempts to provide the smallest set of features that meet its design goals. For example, the object model is pure and classless, thus simplifying the language without sacrificing expressive power. However, some features are included in Cecil that make it more complex, such as supporting multiple dispatching or distinguishing between subtyping and implementation inheritance. Given no other alternative, our preference is for a more powerful language which is more complex over a simpler but less powerful language. Simplicity is important but should not override other language goals.

Cecil's design includes a number of other features that have proven their worth in other systems. These include multiple inheritance of both implementation and interface, closures to implement user-defined control structures and exceptions, and, of course, automatic storage reclamation.

1.2 Overview

This document attempts to provide a fairly detailed specification of the Cecil language, together with discussion of the various design decisions. The next section of this document describes the basic object and message passing model in Cecil. Section 3 extends this dynamically-typed core language with a static type system and describes a type checking algorithm, and section 4 discusses parameterization. Section 5 describes Cecil's module system. Section 6 discusses some related work. Appendix A summarizes the complete syntax for Cecil.

2 Dynamically-Typed Core

Cecil is a pure object-oriented language. All data are objects, and message passing is the only way to manipulate objects. Even instance variables are accessed solely using message passing. This purity offers the maximum benefit of object-oriented programming, allowing code to manipulate an object with no knowledge of (and hence no dependence on) its underlying representation or implementation.

Each Cecil implementation defines how programs are put together. The UW Cecil implementation defines a program to be a sequence of declaration blocks and statements:

```
program        ::=  file_body
file_body      ::=  { top_decl_block | stmt }
```

Declaration blocks are comprised of a set of declarations that are introduced simultaneously; names introduced as part of the declarations in the declaration block are visible throughout the declaration block and also for the remainder of the scope containing the declaration block; the names go out of scope once the scope exits. Because the name of an object is visible throughout its declaration block, objects can inherit from objects defined later within the declaration block and methods can be specialized on objects defined later in the declaration block. In environments where the top-level declaration comprising the program is spread across multiple files, the ability to attach methods to objects defined in some other file is important.

The syntax of declarations is as follows:*

```
top_decl_block ::=  { decl | pragma }
decl           ::=  object_decl
                |   obj_ext_decl
                |   predicate_decl
                |   method_decl
                |   field_decl
                |   let_decl
                |   precedence_decl
                |   include_decl
```

The next four subsections describe objects, methods, fields, and predicate objects. Subsection 2.5 describes variables, statements, and expressions, and subsection 2.6 explains precedence declarations. Subsections 2.7 and 2.8 detail the semantics of message passing in Cecil. Subsection 2.9 describes include declarations and file structure in the UW Cecil implementation, and subsection 2.10 discusses pragmas.

2.1 Objects and Inheritance

The basic features of objects in Cecil are illustrated by the following declarations, which define a simple shape hierarchy. Comments in Cecil either begin with "`--`" and extend to the end of the line or are bracketed between "`(--`" and "`--)`" and can be nested.

* Ignoring type and signature declarations (section 3) and module declarations (section 5).

```
object shape;
object circle isa shape;
object rectangle isa shape;
object rhombus isa shape;
object square isa rectangle, rhombus;
```

The syntax of an object declaration, excluding features relating to static type checking and modules, is as follows:*

```
object_decl    ::= "object" name {relation} [field_inits] ";"
relation       ::= "isa" parents
parents        ::= named_object { "," named_object }
named_object   ::= name
```

(`name` is the token for regular identifiers beginning with a letter; see appendix A.2 for more details on the lexical rules of Cecil.)

Cecil has a classless (prototype-based) object model: self-sufficient objects implement data abstractions, and objects inherit directly from other objects to share code. Cecil uses a classless model primarily because of its simplicity, but also because this avoids problems relating to first-class classes and metaclasses and because it makes defining unique named objects with specialized behavior easy. Section 2.2 shows how treating "instance" objects and "class" objects uniformly enables CLOS-style `eql` specializers to be supported with no extra mechanism.

Section 2.3 describes field initializers.

2.1.1 Inheritance

Objects can inherit from other objects. Informally, this means that the operations defined for parent objects will also apply to child objects. Inheritance in Cecil may be multiple, simply by listing more than one parent object; any ambiguities among methods and/or fields defined on these parents will be reported to the programmer. Inheriting from the same ancestor more than once, either directly or indirectly, has no effect other than to place the ancestor in relation to other ancestors; Cecil has no repeated inheritance as in Eiffel [Meyer 88, Meyer 92]. An object need not have any (explicit) parents; all objects are considered to inherit from the predefined `any` object (see section 2.1.4). The inheritance graph must be acyclic.

Inheritance in Cecil requires a child to accept all of the fields and methods defined in the parents. These fields and methods may be overridden in the child, but facilities such as excluding fields or methods from the parents or renaming them as part of the inheritance, as found in Eiffel, are not present in Cecil. We have deliberately chosen to experiment with a simpler inheritance semantics.

Finally, it is important to note that inheritance of code is distinct from *subtyping* ("inheritance" of interface or of specification). Section 3 explains Cecil's support for subtyping and static type checking.

* Appendix A gives the complete syntax of the language and explains the notation.

2.1.2 Object Instantiation

Rather than introduce a distinct instantiation concept into the language, new "instances" of some object are created solely by inheriting from the object. Object declarations allow statically-known, named "instances" to be defined, while *object constructor expressions* allow new anonymous "instances" to be created at run-time. An object constructor expression is syntactically and semantically similar to an object declaration, except that there is no name for the object being created. For example:

```
let s1 := object isa square;   -- create a fresh "instance" of square when executed
```

Section 2.5.7 describes object constructor expressions in more detail. Note that the parent of an object must be statically known; Cecil does not allow objects to be created whose parents are run-time computed expressions. This is a restriction over some other prototype-based languages.

2.1.3 Extension Declarations

The inheritance structure of a named object may be augmented separately from the object declaration through an object extension declaration:

```
obj_ext_decl   ::= "extend" named_object {relation} [field_inits] ";"
```

In Cecil, object extension declarations, in conjunction with field and method declarations, enable programmers to extend previously-existing objects. This ability can be important when reusing and integrating groups of objects implemented by other programmers. For example, predefined objects such as `int`, `i_vector`, and `m_vector` are given additional behavior and ancestry through separate user code. Similarly, particular applications may need to add application-specific behavior to objects defined as part of other applications. For example, a text-processing application may add specialized tab-to-space conversion behavior to strings and other collections of characters defined in the standard library. Other object-oriented languages such as C++ [Stroustrup 86, Ellis & Stroustrup 90] and Eiffel do not allow programmers to add behavior to existing classes without modifying the source code of the existing classes, and completely disallow adding behavior to built-in classes like strings. Sather is a notable exception, allowing a new class to be defined which is a superclass of some existing classes [Omohundro 93]. Section 3.3.3 explains how object extensions are particularly useful to declare that two objects, provided by two independent vendors, are subtypes of some third abstract type. Section 5 describes how modules can be used to localize extensions to particular regions of code.

2.1.4 Predefined Objects

Several objects are predefined and play special roles.

- The `void` object is used to represent a lack of a value. It is used as the result of methods or expressions that have no useful result. The system will guarantee (statically in the presence of type checking) that `void` is never passed as an argument to a method.
- The `any` object is implicitly the ancestor of all non-`void` objects. It supports behavior that is shared by all objects.

A Cecil implementation provides other predefined objects, such as integers, floats, characters, booleans, and mutable and immutable vectors and strings, as part of its standard library.

2.1.5 Closures

Cecil includes closure objects which represent first-class anonymous functions. Closures are lexically nested in their enclosing scope. As with methods, a closure can have formal arguments. A closure object is "invoked" by sending it the `eval` message, with additional actual arguments for each of its formal arguments. Closures are considered to inherit from the `closure` predefined object.

More details on closures are given throughout the remainder of section 2. In particular, section 2.5.9 describes the syntax and semantics of closure constructor expressions and section 2.2.2 describes the evaluation rules for closure `eval` methods.

2.2 Methods

The following definitions expand the earlier shape hierarchy with some methods:

```
object shape;
  method draw(s, d) { (-- draws s on display d --) }
  method move_to(s, new_center) { (-- move s to new_center --) }

object circle isa shape;
  method area(c@circle) { c.radius * c.radius * pi}
  method circum(c@circle) { c.radius * 2 * pi }

object rectangle isa shape;
  method area(r@rectangle) { r.length * r.width }
  method circum(r@rectangle) { 2 * r.length + 2 * r.width }
  method draw(r@rectangle, d@Xwindow) {
    (-- override draw for the case of drawing rectangles on X windows --) }

object rhombus isa shape;

object square isa rectangle, rhombus; -- inherits area method, but overrides circum
  method circum(s@square) { 4 * s.length }
```

The syntax for method declarations (again, excluding aspects relating to static typing and encapsulation) is as follows:

```
method_decl    ::= "method" method_name "(" [formals] ")" {pragma}
                        "{" (body | prim_body) "}" [";"]
method_name    ::= msg_name | op_name
msg_name       ::= name
formals        ::= formal { "," formal }
formal         ::= [name] specializer        formal names are optional, if never referenced
specializer    ::= "@" named_object          specialized formal
                |  empty                     unspecialized formal
```

(`op_name` is the token for infix and prefix operators beginning with a punctuation symbol; see appendix A.2 for more details.)

As a convention, we indent method declarations under the associated object declaration. This has no semantic implication, but it helps to visually organize a collection of object and method declarations in the absence of a more powerful graphical programming environment [Chambers 92b].

2.2.1 Argument Specializers and Multi-Methods

In Cecil, a method specifies the kinds of arguments for which its code is designed to work. For each formal argument of a method, the programmer may specify that the method is applicable only to actual arguments that are implemented or represented in a particular way, i.e., that are equal to or inherit from a particular object. These specifications are called *argument specializers*, and arguments with such restrictions are called *specialized arguments*. The `r@rectangle` notation specializes the `r` formal argument on the `rectangle` object, implying that the method is intended to work correctly with any actual argument object that is equal to or a descendant of the `rectangle` object as the `r` formal. An unspecialized formal argument (one lacking a `@...` suffix), such as `s` and `new_center` in the `move_to` method above, is treated as being specialized on the predefined object `any` that is implicitly an ancestor of all other objects; consequently an unspecialized formal can accept any argument object.

Methods may be overloaded, i.e., there may be many methods with the same name, as long as the methods with the same name and number of arguments differ in their argument specializers. Methods with different numbers of arguments are independent; the system considers the number of arguments to be part of the method's name. When sending a message of a particular name with a certain number of arguments, the method lookup system (described in section 2.7) will resolve the overloaded methods to a single most-specific applicable method based on the dynamic values of the actual argument objects and the corresponding formal argument specializers of the methods.

Zero, one, or several of a method's arguments may be specialized, thus enabling Cecil methods to emulate normal undispatched functions (by leaving all formals unspecialized, as in `move_to` above) and singly-dispatched methods (by specializing only the first argument, as in the `area` methods) as well as true multi-methods (as in the specialized version of `draw` for rectangles on X windows). Statically-overloaded functions and functions declared via certain kinds of pattern-matching also are subsumed by multi-methods. Callers which send a particular message to a group of arguments are not aware of the collection of methods that might handle the message or which arguments of the methods are specialized, if any; these are internal implementation decisions that should not affect callers. In particular, a given message can initially be implemented with a single unspecialized procedure and then later extended or replaced with several specialized implementations, without affecting clients of the original method, as occurs with the `draw` methods in the previous example. In contrast, CLOS has a "congruent lambda list" rule that requires all methods in a particular generic function to specialize on the same argument positions.

Argument specializers are distinct from type declarations. Argument specializers restrict the allowed implementations of actual arguments and are used as part of method lookup to locate a suitable method to handle a message send. Type declarations require that certain operations be supported by argument objects, but place no constraints on how those operations are implemented. Type declarations have no effect on method lookup.

The name of a formal may be omitted if it is not needed in the method's body. Unlike singly-dispatched languages, there is no implicit `self` formal in Cecil; all formals are listed explicitly.

Cecil's classless object model combines with its definition of argument specializers to support something similar to CLOS's `eql` specializers. In CLOS, an argument to a multi-method in a generic function may be restricted to apply only to a particular *object* by annotating the argument specializer with the `eql` keyword. Cecil needs no extra language mechanism to achieve a similar effect, since methods already are specialized on particular objects. Cecil's mechanism differs from CLOS's in that in Cecil such a method also will apply to any children of the specializing object, while in CLOS the method will apply only for that object. Dylan, a descendant of CLOS, has a `singleton` specializer that is analogous to CLOS's `eql` specializer [Apple 92].

As mentioned in subsection 2.1.3, methods can be added to existing objects without needing to modify those existing objects. This facility, lacking in most object-oriented languages, can make reusing existing components easier since they can be adapted to new uses by adding methods, fields, and even parents to them.

The names of methods and fields are in a name space separate from the name space of objects and variables. A method or field can have the same name as a variable or object without confusion.

2.2.2 Method Bodies

The syntax of the body of a method, closure, or parenthetical subexpression is as follows:

`body`	`::=`	`{stmt} result`	
	|	*empty*	*return* `void`
`result`	`::=`	`normal_return`	*return an expression*
	|	`non_local_rtn`	*return from the lexically-enclosing method*
`normal_return`	`::=`	`decl_block [";"]`	*return* `void`
	|	`assignment [";"]`	*return* `void`
	|	`expr [";"]`	*return result of expression*
`non_local_rtn`	`::=`	`"^" [";"]`	*do a non-local return, returning* `void`
	|	`"^" expr [";"]`	*do a non-local return, returning a result*

(The syntax and semantics of statements and expressions is described in section 2.5.)

When invoked, a method evaluates its statements in a new environment containing bindings for the method's formal parameters and nested in the method's lexically-enclosing environment. (The interactions among nested scopes, method lookup, and other language features is described in more detail in section 2.7.7.)

The result of the message invoking the method is the result of the last statement in the method's body. If the method's body is empty, then the method returns the special `void` value. Alternatively, a method returns `void` if the last statement is a declaration block, an assignment statement, or an expression that itself returns `void`. The `void` value is used to indicate that the method returns no useful result. The system ensures that `void` is not accidentally used in later computation by reporting an error (statically in the presence of type checking) if `void` is passed as an argument to a message.

When a closure's `eval` method is invoked, evaluation proceeds much like the evaluation of any other method. One difference is that a closure `eval` method may force a *non-local return* by prefixing the result expression with the ^ symbol; if the result expression is omitted, then `void` is returned non-locally. A non-local return returns to the caller of the closest lexically-enclosing non-closure method rather than to the caller of the `eval` method, just like a non-local return in Smalltalk-80* [Goldberg & Robson 83] and Self and similar to a `return` statement in C. The language currently prohibits invoking a non-local return after the lexically-enclosing scope of a closure has returned; first-class continuations are not supported.

2.2.3 Primitive Methods

```
prim_body        ::=  "prim" { language_binding }

language_binding ::=  language ":" code_string
                 |    language "{" code_chars "}"

language         ::=  name

code_string      ::=  string

code_chars       ::=  brace_balanced_chars
```
any characters, with balanced use of "{" and "}"

Low-level operations, such as integer arithmetic, vector indexing, looping, and file I/O, are implemented through the use of primitive methods. A primitive method's body is a list of (language name, implementation source code) pairs. The details of the protocol for writing code in another language inside a Cecil primitive method are implementation-specific. The UW Cecil implementation recognizes the `c_++` and `rtl` language names, for primitives written in C++ and the compiler's internal intermediate language, respectively. It is fairly straightforward to make calls to routines written in C++ from Cecil by defining a primitive method whose body is written in C++.

Looping primitive behavior is provided by the `loop` primitive method specialized on the `closure` predefined object. This method repeatedly invokes its argument closure until some closure performs a non-local return to break out of the loop. Other languages such as Scheme [Rees & Clinger 86] avoid the need for such a primitive by relying instead on user-level tail recursion and implementation-provided tail-recursion elimination. However, tail-recursion elimination precludes complete source-level debugging [Chambers 92a, Hölzle *et al.* 92] and consequently is undesirable in general. The primitive `loop` method may be viewed as a simple tail-recursive method for which the implementation has been instructed to perform tail-recursion elimination.

A primitive body may be included at the top-level using a primitive body declaration:

```
prim_decl        ::=  prim_body ";"
```

This construct allows code from other languages to be included outside of any compiled routines. Primitive declarations can be used to include global declarations used by primitive methods. Again, the detailed semantics of this construct are implementation-specific.

---

* Smalltalk-80 is a trademark of ParcPlace Systems.

2.3 Fields

Object state, such as instance variables and class variables, is supported in Cecil through *fields* and associated *accessor methods*. To define a mutable instance variable `x` for a particular object `obj`, the programmer can declare a `field` of the following form:

```
var field x(@obj);
```

This declaration allocates space for an object reference in the `obj` object and constructs two real methods attached to the `obj` object that provide the only access to the variable:

```
method x(v@obj) { prim rtl { <v.x> } } -- the get accessor method
method set_x(v@obj, value) { prim rtl { <v.x> := value; } } -- the set accessor method
```

The *get accessor method* returns the contents of the hidden variable. The *set accessor method* mutates the contents of the hidden variable to refer to a new object, and returns `void`. Accessor methods are specialized on the object containing the variable, thus establishing the link between the accessor methods and the object. For example, sending the `x` message to the `obj` object will find and invoke the get accessor method and return the contents of the hidden variable, thus acting like a reference to `obj`'s `x` instance variable. (Section 5 describes how these accessor methods can be encapsulated within the data abstraction implementation and protected from external manipulation.)

To illustrate, the following declarations define a standard list inheritance hierarchy:

```
object list isa ordered_collection;
   method is_empty(l@list) { l.length = 0 }
   method prepend(x, l@list) { -- dispatch on second argument
      object isa cons { head := x, tail := l } }

object nil isa list; -- empty list
   method length(@nil) { 0 }
   method do(@nil, ) {} -- iterating over all elements of the empty list: do nothing
   method pair_do(@nil, , ) {}
   method pair_do(, @nil, ) {}
   method pair_do(@nil, @nil, ) {}

object cons isa list; -- non-empty lists
   var field head(@cons); -- defines head(@cons) and set_head(@cons, ) accessor methods
   var field tail(@cons); -- defines tail(@cons) and set_tail(@cons, ) accessor methods
   method length(c@cons) { 1 + c.tail.length }
   method do(c@cons, block) {
      eval(block, c.head); -- call block on head of list
      do(c.tail, block); } -- recur down tail of list
   method pair_do(c1@cons, c2@cons, block) {
      eval(block, c1.head, c2.head);
      pair_do(c1.tail, c2.tail, block); }
```

The `cons` object has two fields, only accessible through the automatically-generated accessor methods.

The syntax of field declarations, excluding static typing aspects and encapsulation, is as follows:

```
field_decl      ::=  ["shared"] ["var"] "field" method_name "(" formal ")"
                        {pragma} [":=" expr] ";"
```

2.3.1 Read-Only vs. Mutable Fields

By default, a field is immutable: only the get accessor method is generated for it. To support updating the value of a field, the `var` prefix must be used with the field declaration. The presence of the `var` annotation triggers generation of the set accessor method. Immutable fields receive their values either as part of object creation or by an initializing expression associated with the field declaration; see section 2.3.4. Note that the contents of an immutable field can itself be mutable, but the binding of the field to its contents cannot change. (Global and local variables in Cecil similarly default to initialize-only semantics, with an explicit `var` annotation required to allow updating of the variable's value, as described in section 2.5.2.)

In general, we believe that it is beneficial to explicitly indicate when a field is mutable; to encourage this indication, immutable fields are the default. Programmers looking at code can more easily reason about the behavior of programs if they know that certain parts of the state of an object cannot be side-effected. Similarly, immutable fields support the construction of immutable "value" objects, such as complex numbers and points, that are easier to reason about.

Many languages, including Self and Eiffel, support distinguishing between assignable and constant variables, but few imperative languages support initialize-only instance variables. CLOS can define initialize-only variables in the sense that a slot can be initialized at object-creation time without a set accessor method being defined, but in CLOS the `slot-value` primitive function can always modify a slot even if the set accessor is not generated.

2.3.2 Fields and Methods

Accessing variables solely through automatically-generated wrapper methods has a number of advantages over the traditional mechanism of direct variable access common in most object-oriented languages. Since instance variables can only be accessed through messages, all code becomes representation-independent to a certain degree. Instance variables can be overridden by methods, and vice versa, allowing code to be reused even if the representation assumed by the parent implementation is different in the child implementation. For example, in the following code, the `rectangle` abstraction can inherit from the `polygon` abstraction but alter the representation to something more appropriate for rectangles:

```
object polygon;
    var field vertices(@polygon);
    method draw(p@polygon, d@output_device) {
        (-- draw the polygon on an output device, accessing vertices --) }


object rectangle isa polygon;
    var field top(@rectangle);
    var field bottom(@rectangle);
    var field left(@rectangle);
    var field right(@rectangle);
```

```
method vectices(r@rectangle) {
    -- ++ is a binary operator, here creating a new point object
    [r.top    ++ r.left,  r.top    ++ r.right,
     r.bottom ++ r.right, r.bottom ++ r.left] }
method set_vertices(r@rectangle, vs) {
    (-- set corners of rectangle from vs list, if possible --) }
```

Even within a single abstraction, programmers can change their minds about what is stored and what is computed without rewriting lots of code. Syntactically, a simple message send that accesses an accessor method is just as concise as would be a variable access (using the `p.x` syntactic sugar, described in section 2.5.6), thus imposing no burden on the programmer for the extra expressiveness. Other object-oriented languages such as Self and Trellis have shown the advantages of accessing instance variables solely through special get and set accessor methods. CLOS enables get and/or set accessor methods to be defined automatically as part of the `defclass` form, but CLOS also provides a lower-level `slot-value` primitive that can read and write any slot directly. Dylan joins Self and Trellis in accessing instance variables solely through accessor methods.

An object may define or inherit several fields with the same name. Just as with overloaded methods, this is legal as long as two methods, accessor or otherwise, do not have the same name, number of arguments, and argument specializers. A method may override a field accessor method without removing the field's memory location from the object, since a resend within the overriding method may invoke the field accessor method. Implementations may optimize away the storage for a field in an object if it cannot be accessed, as with the `vertices` field in the `rectangle` object.

2.3.3 Copy-Down vs. Shared Fields

By default, each object inheriting a field declaration receives its own space to hold its version of the field's contents, and the field's accessor methods access the memory space associated with their first argument. Such a "copy-down" field acts much like an instance variable declaration in a class-based language, since each object gets its own local copy of the field. Alternatively, a field declaration may be prefixed with the `shared` keyword, implying that all inheriting objects should share a single memory location. A shared field thus acts like a class variable.

Supporting both copy-down and shared fields addresses weaknesses in some other prototype-based object-oriented languages relative to class-based languages. In class-based languages, instance variables declared in a superclass are automatically copied down into subclasses; the *declaration* is inherited, not the variable's *contents*. Class variables, on the other hand, are shared among the class, its instances, and its subclasses. In some prototype-based languages, including Self and Actra [Lieberman 86], instance variables of one object are not copied down into inheriting objects; rather, these variables are shared, much like class variables in a class-based language. In Self, to get the effect of object-specific state, most data types are actually defined with two objects: one object, the *prototype*, includes all the instance-specific variables that objects of the data type need, while the other object, the *traits object*, is inherited by the prototype and holds the methods and shared state of the data type [Ungar *et al.* 91]. New Self objects are created by cloning (shallow-copying) a prototype, thus giving new objects their own instance variables while sharing the parent traits object and its methods and state. Defining a data type in two pieces can be awkward, especially since it separates the declarations of instance variables from the definitions of the methods that access them. Furthermore, inheriting the instance variable part of the implementation of one data type into another is more difficult in Self than in class-based languages, relying on complex inheritance rules and dynamic inheritance [Chambers *et al.* 91] or programming environment support [Ungar 95]. Copy-down fields in Cecil solve these problems in Self without sacrificing the simple classless object model. In Cecil, only one object needs to be defined for a given data type, and the field declarations can be in the same place as the method declarations that access them. This design increases both conciseness and readability, at the cost of some additional language mechanism.

Cecil objects are created only through object declarations and object constructor expressions; these two expressions have similar run-time effects, with the former additionally binding statically-known names to the created objects enabling methods and fields to be associated with them and enabling other objects to inherit from them. Cecil needs no other primitive mechanism to create or copy objects as do other languages. Self provides a shallow-copy (clone) primitive in addition to object literal syntax (analogous to Cecil's object constructor expressions), in part because there are no "copy-down" data slots in Self. Class-based languages typically include several mechanisms for creating instances and classes and relations among them. On the other hand, creating an object by inheriting from an existing object may not be as natural as creating an object by copying an existing object.

2.3.4 Field Initialization

Cecil allows a field to be given an initial value when it is declared by suffixing the field declaration with the `:=` symbol and an initializing expression. Additionally, when an object is created, an object-specific initial value may be specified for a non-shared field. The syntax of field initializers for object declarations and object constructor expressions is as follows:

```
field_inits    ::= "{" field_init { "," field_init } "}"
field_init     ::= name [location] ":=" expr
location       ::= "@" named_object
```

For example, the following method produces a new list object with particular values for its inherited fields:

```
method prepend(e, l@list) {
   object isa cons { head := e, tail := l } }
```

For a field initialization of the form `name := expr`, the field to be initialized is found by performing a lookup akin to message lookup to find a field declaration named `name`, starting with the object being created. Method lookup itself cannot be used directly, since the field to be initialized may have been overridden with a method of the same name. Instead, a form of lookup that ignores all methods is used. If this lookup succeeds in finding a single most-specific matching field declaration, then that field is the one given an initial value; the matching field should not be a shared field. If no matching field or more than one matching field is found, then a "field initializer not understood" or an "ambiguous field initializer" error, respectively, is reported. To resolve ambiguities and to initialize fields otherwise overridden by other fields, an extended name for the field of the form `name@obj := expr` may be used instead. For these kind of initializers, lookup for a matching field begins with the object named `obj` rather than the object being created. The `obj` object must be an ancestor of the object being created. Extended field names are analogous to a similar mechanism related to directed resends, described in section 2.8.

Immutable shared fields must be initialized as part of the field declaration; there is no other way to give them a value. Immutable copy-down fields may be initialized as part of the field declaration, but often they are initialized as part of object constructor expressions for objects that inherit the field, leading to a more functional programming style where data structures are (largely) immutable.

To avoid pesky problems with uninitialized variables, all fields must be initialized before being accessed, either by providing an initial value as part of the field declaration, by providing an object-specific value as part of the object declaration or object constructor expression, or by assigning to the field before reading from it. The static type checker warns when it cannot prove that at least one of the first two options is taken for each field inherited by an object, as described in section 3.7.

In Cecil, the initializing expression for a field declaration is not evaluated until the field is first read. If the field is a shared field, then the initializer is evaluated and the contents of the field is updated to refer to the initial value; subsequent reads of the shared field will simply return the initial value. This supports functionality similar to `once` functions in Eiffel and other languages. If the field is a copy-down field, then the initializing expression will be evaluated separately for each object accessed, and the result cached for that object. The initializing expression may name the formal parameter of the field declaration, allowing the initial value of the field to reference the object of which the field is a part. The default initializer is not evaluated if it is not needed, i.e., if the field has already been given a contents as part of object creation or via invocation of the set accessor.

By evaluating field initializers on demand rather than at declaration time, we avoid the need to specify some arbitrary ordering over field declarations or to resort to an unhelpful "unspecified" or "implementation-dependent" rule. It is illegal to try to read the value of a field during execution of the field's initializer; no cyclic dependencies among field initializers are allowed.

Evaluating a copy-down field's initializer expression repeatedly for each inheriting object seems to support common Cecil programming style. This corresponds to CLOS's `:initform` specifier. An earlier version of Cecil specified caching of the results of field initializer evaluation so that other objects evaluating the same initializer expression would end up sharing the initial value. The initializing expression was viewed as a shared part of the field declaration, not as a separate part copied down to each inheriting object. This earlier semantics corresponded more to CLOS's `:default-initargs` specifier. The difference in the semantics is exposed if the initializing expression evaluates to a new mutable object. In practice, it seems that each object wants its own mutable object rather than sharing the mutable object among all inheriting objects. Moreover, the old semantics can be simulated with a combination of a copy-down field that accesses a shared field to get the field's initial value.

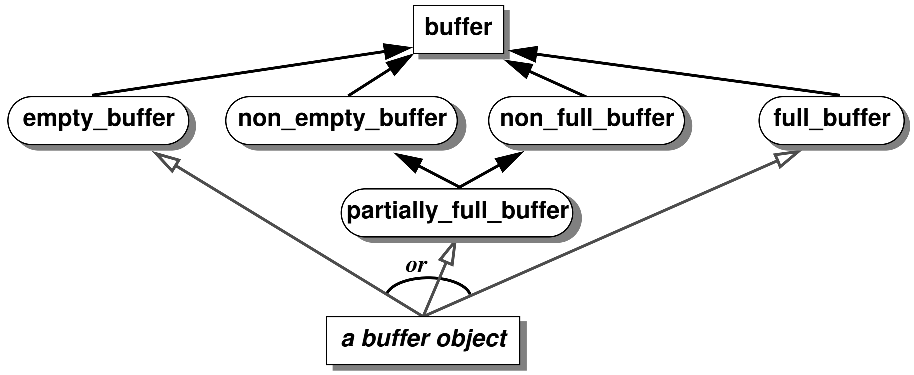
2.4 Predicate Objects

To enable inheritance and classes to be used to capture run-time varying object behavior, Cecil support *predicate objects* [Chambers 93b]. Predicate objects are like normal objects except that they have an associated predicate expression. The semantics of a predicate object is that if an object inherits from the parents of the predicate object and also the predicate expression is true when evaluated on the child object, then the child is considered to also inherit from the predicate object in addition to its explicitly-declared parents. Since methods can be associated with predicate objects, and since predicate expressions can test the value or state of a candidate object, predicate objects allow a form of state-based dynamic classification of objects, enabling better factoring of code. Also, predicate objects and multi-methods allow a pattern-matching style to be used to implement cooperating methods.

For example, predicate objects could be used to implement a bounded buffer abstraction:

```
object buffer isa collection;
   field elements(b@buffer); -- a queue of elements
   field max_size(b@buffer); -- an integer
   method length(b@buffer) { b.elements.length }
   method is_empty(b@buffer) { b.length = 0 }
   method is_full(b@buffer)  { b.length = b.max_size }

predicate empty_buffer isa buffer when buffer.is_empty;
   method get(b@empty_buffer) { ... } -- raise error or block caller

predicate non_empty_buffer isa buffer when not(buffer.is_empty);
   method get(b@non_empty_buffer) { remove_from_front(b.elements) }

predicate full_buffer isa buffer when buffer.is_full;
   method put(b@full_buffer, x) { ... } -- raise error or block caller

predicate non_full_buffer isa buffer when not(buffer.is_full);
   method put(b@non_full_buffer, x) { add_to_back(b.elements, x); }

predicate partially_full_buffer isa non_empty_buffer, non_full_buffer;
```

The following diagram illustrates the inheritance hierarchy created by this example (the explicit inheritance link from the buffer object to `buffer` is omitted):



Predicate objects increase expressiveness for this example in two ways. First, important states of bounded buffers, e.g., empty and full states, are explicitly identified in the program and named. Besides documenting the important conditions of a bounded buffer, the predicate objects remind the programmer of the special situations that code must handle. This can be particularly useful during maintenance phases as code is later extended with new functionality. Second, attaching methods directly to states supports better factoring of code and eliminates `if` and `case` statements, much as does distributing methods among classes in a traditional object-oriented language. In the absence of predicate objects, a method whose behavior depended on the state of an argument object would include an `if` or `case` statement to identify and branch to the appropriate case; predicate objects eliminate the clutter of these tests and clearly separate the code for each case. In a more complete example, several methods might be associated with each special state of the buffer. By factoring the code, separating out all the code associated with a particular state or behavior mode, we hope to improve the readability and maintainability of the code.

The syntax for a predicate object declaration is as follows:

```
predicate_decl ::= "predicate" name {relation} [field_inits] ["when" expr] ";"
```

2.4.1 Predicate Objects and Inheritance

For normal objects, one object is a child of another object exactly when the relationship is declared explicitly through `isa` declarations by the programmer. Predicate objects, on the other hand, support a form of automatic property-based classification: an object *O* is automatically considered a child of a predicate object *P* exactly when the following two conditions are satisfied:

- the object *O* is a descendant of each of the parents of the predicate object *P*, and
- the predicate expression of the predicate object *P* evaluates to true, when evaluated in a scope where each of the predicate object's parent names is bound to the object *O*.

By evaluating the predicate expression in a context where the parent names refer to the object being tested, the predicate expression can query the value or state of the object.

Since the state of an object can change over time (fields can be mutable), the results of predicate expressions evaluated on the object can change. If this happens, the system will automatically reclassify the object, recomputing its implicit inheritance links. For example, when a buffer object becomes full, the predicates associated with the non_full_buffer and full_buffer predicate objects both change, and the inheritance graph of the buffer object is updated. As a result, different methods may be used to respond to messages, such as the put message in the filled buffer example. Predicate expressions are evaluated lazily as part of method lookup, rather than eagerly as the state of an object changes. Only when the value of some predicate expression is needed to determine the outcome of method lookup is the predicate evaluated. A separate paper describes efficient implementation schemes for predicate objects [Chambers 93].

If a predicate object inherits from another predicate object, it is a special case of that parent predicate object. This is because the child predicate object will only be in force whenever its parent predicate object's predicate evaluates to true. In essence, the parent's predicate expression is implicitly conjoined with the child's predicate expression. A non-predicate object also may inherit explicitly from a predicate object, with the implication that the predicate expression will always evaluate to true for the child object; the system verifies this assertion dynamically. For example, an unbounded buffer object might inherit explicitly from the non_full_buffer predicate object.

A predicate object need not have a when clause, as illustrated by the partially_full_buffer predicate object defined above. Such a predicate object may still depend on a condition if at least one of its ancestors is a predicate object. In the above example, the partially_full_buffer predicate object has no explicit predicate expression, yet since an object only inherits from partially_full_buffer whenever it already inherits from both non_empty_buffer and non_full_buffer, the partially_full_buffer predicate object effectively repeats the conjunction of the predicate expressions of its parents, in this case that the buffer be neither empty nor full.

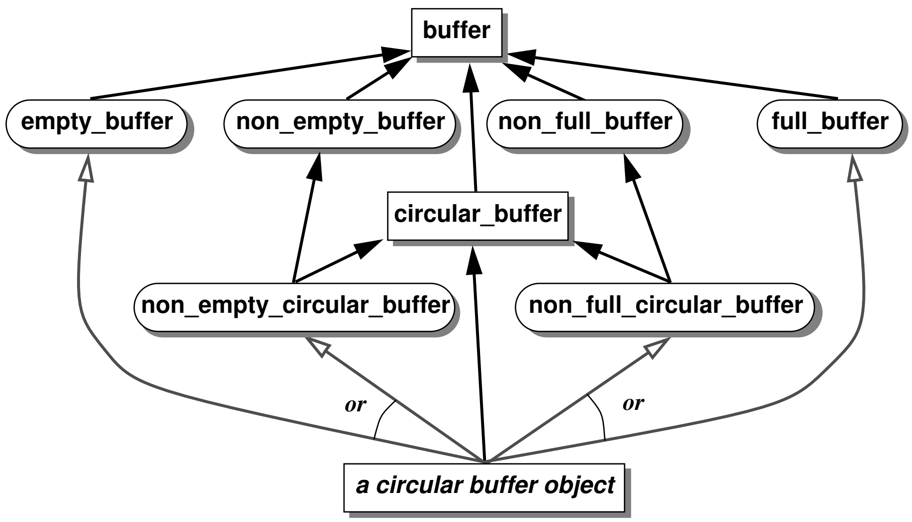
Predicate objects are intended to interact well with normal inheritance among data abstractions. If an abstraction is implemented by inheriting from some other implementation, any predicate objects that specialize the parent implementation will automatically specialize the child implementation whenever it is in the appropriate state. For example, a new implementation of bounded buffers could be built that used a fixed-length array with insert and remove positions that cycle around the array:*

```
object circular_buffer isa buffer;

    field array(b@circular_buffer);  -- a fixed-length array of elements
    var field insert_pos(b@circular_buffer);  -- an index into the array
    var field remove_pos(b@circular_buffer);  -- another integer index

    method max_size(b@circular_buffer) { b.array.length }

    method length(b@circular_buffer) {
        -- % is modulus operator
        (b.insert_pos - b.remove_pos) % b.array.length }


predicate non_empty_circular_buffer isa circular_buffer, non_empty_buffer;
```

* This implementation overrides buffer's max_size field with a method and then ignores the buffer's elements field. In practice a more efficient implementation would break up buffer into an abstract parent object and two child objects for the queue-based implementation and the circular array implementation.

```
    method get(b@non_empty_circular_buffer) {
        var x := fetch(b.array, b.remove_pos);
        b.remove_pos := (b.remove_pos + 1) % b.array.length;
        x }

predicate non_full_circular_buffer isa circular_buffer, non_full_buffer;
    method put(b@non_full_circular_buffer, x) {
        store(b.array, b.insert_pos, x);
        b.insert_pos := (b.insert_pos + 1) % b.array.length; }
```

The following diagram illustrates the extended inheritance graph for bounded and circular buffers (the `partially_full_buffer` predicate object is omitted):



Since the `circular_buffer` implementation inherits from the original `buffer` object, a `circular_buffer` object will automatically inherit from the `empty_buffer` or `full_buffer` predicate object whenever the `circular_buffer` happens to be in one of those states. No `empty_circular_buffer` or `full_circular_buffer` objects need to be implemented if specialized behavior is not needed. The `non_empty_circular_buffer` and `non_full_circular_buffer` predicate objects are needed to override the default `get` and `put` methods in the non-blocking states. Any object that inherits from `circular_buffer` and that also satisfies the predicate associated with `non_empty_buffer` will automatically be classified as a `non_empty_circular_buffer`.

The specification of when an object inherits from a predicate object implicitly places a predicate object just below its immediate parents and after all other normal children of the parents. For example, consider an empty circular buffer object. Both the buffer object and its parent, the `circular_buffer` object, will be considered to inherit from the `empty_buffer` predicate object. Because `circular_buffer` is considered to inherit from `empty_buffer`, any methods attached to `circular_buffer` will override methods attached to `empty_buffer`. Often this is the desired behavior, but at other times it might be preferable for methods attached to predicate objects to override methods attached to "cousin" normal objects.* If this were the case, then the buffer code could be simplified somewhat, as follows:

```
object buffer isa collection;
```

```
   ... -- elements, length, etc.
   method get(b@buffer) { remove_from_front(b.elements) }
   method put(b@buffer, x) { add_to_back(b.elements, x); }

predicate empty_buffer isa buffer when buffer.is_empty;
   method get(b@empty_buffer) { ... } -- raise error or block caller

predicate full_buffer isa buffer when buffer.is_full;
   method put(b@full_buffer, x) { ... } -- raise error or block caller

object circular_buffer isa buffer;
   ... -- array, insert_pos, length, etc.
   method get(b@circular_buffer) {
      var x := fetch(b.array, b.remove_pos);
      b.remove_pos := (b.remove_pos + 1) % b.array.length;
      x }
   method put(b@circular_buffer, x) {
      store(b.array, b.insert_pos, x);
      b.insert_pos := (b.insert_pos + 1) % b.array.length; }
```

The non-blocking versions of `get` and `put` would be associated with the `buffer` object directly, and the `non_empty_buffer`, `non_full_buffer`, and `partially_full_buffer` predicate objects could be removed (if desired). The non-blocking `get` and `put` routines for circular buffers would similarly be moved up to the `circular_buffer` object itself, with the `non_empty_circular_buffer` and `non_full_circular_buffer` predicate objects being removed also. If the methods attached to the `empty_buffer` object were considered to override those of the `circular_buffer` object, then sending `get` to a circular buffer that was empty would (correctly) invoke the `empty_buffer` implementation. In the current semantics of predicate objects in Cecil, however, the `circular_buffer`'s implementation of `get` would be invoked, leading to an error. A third potential semantics would be to consider the predicate object to be unordered with respect to "cousin" objects, and methods defined on two cousins to be mutually ambiguous. More experience with predicate objects is needed to adequately resolve this question.

2.4.2 Predicate Objects and Fields

Fields may be associated with a predicate object. This has the effect of reserving persistent space for the field in any object that might be classified as a descendant of the predicate object. The value stored in the field persists even when the field is inaccessible. At object-creation time, an initial value may be provided for fields potentially inherited from predicate objects, even if those fields may not be visible in the newly-created object. The semantics of accessing a field attached to a predicate object is governed by the semantics of accessing its corresponding accessor methods.

The following example exploits this semantics to implement a graphical window object that can be either expanded or iconified. Each of the two important states of the window remembers its own screen location (using a field named `position` in both cases), plus some other mode-specific

* One object is a cousin of another if they share a common ancestor but are otherwise unrelated.

information such as the text in the window and the bitmap of the icon, and this data persists across openings and closings of the window:

```
object window isa interactive_graphical_object;
    var field iconified(@window) := false;
    method display(w@window) {
        -- draw window using w.position
        ... }
    method erase(w@window) {
        -- clear space where window is
        ... }
    method move(w@window, new_position) {
        -- works for both expanded and iconified windows!
        w.erase; w.position := new_position; w.display; }

predicate expanded_window isa window when not(window.iconified);
    var field position(@expanded_window) := upper_left;
    field text(@expanded_window);
    method iconify(w@expanded_window) {
        w.erase; w.iconified := true; w.display; }

predicate iconified_window isa window when window.iconified;
    var field position(@iconfied_window) := lower_right;
    field icon(@iconified_window);
    method open(w@iconified_window) {
        w.erase; w.iconified := false; w.display; }

method create_window(open_position, iconified_position,
                     text, icon) {
    object isa window {
        iconified := false,
        position@open_window := open_position,
        position@iconified_window := iconified_position,
        text := text, icon := icon } }
```

A window object has two `position` fields, but only one is visible at a time. This allows the `display`, `erase`, and `move` routines to send the message `position` as part of their implementation, without needing to know whether the window is open or closed. The `create_window` method initializes both `position` fields when the window is created, even though the position of the icon is not visible initially. The `position@`*object* notation used in the field initialization resolves the ambiguity between the two `position` fields.

2.5 Statements and Expressions

A statement is a declaration block, an assignment, or an expression:

```
stmt            ::= decl_block
                 |  assignment ";"
                 |  expr ";"
```

An expression is either a literal, a reference to a variable or a named object, an object constructor expression, a vector constructor expression, a closure constructor expression, a message, a resend, or a parenthetical subexpression:

```
expr            ::=  binop_expr
binop_expr      ::=  binop_msg | unop_expr
unop_expr       ::=  unop_msg | dot_expr
dot_expr        ::=  dot_msg | simple_expr
simple_expr     ::=  literal
                  |  ref_expr
                  |  vector_expr
                  |  closure_expr
                  |  object_expr
                  |  message
                  |  resend
                  |  paren_expr
```

All of these constructs are described below, except for resends which are described later in section 2.8 and declarations other than variable declarations which are described in other sections.

2.5.1 Declaration Blocks

A declaration block is an unbroken sequence of declarations:

```
decl_block      ::=  decl { decl }
```

Names introduced as part of the declarations in the declaration block are visible throughout the declaration block and also for the remainder of the scope containing the declaration block; the names go out of scope once the scope exits. Because the name of an object is visible throughout its declaration block, objects can inherit from objects defined later within the declaration block and methods can be specialized on objects defined later in the declaration block. Similarly, methods declared within a single declaration block can be mutually recursive and there is no need for forward declarations or the like. In environments where the top-level declaration block comprising the program is spread across multiple files, as in the UW Cecil implementation, the ability to attach methods to objects defined later in some other file is important.

2.5.2 Variable Declarations

Variable declarations have the following syntax:

```
let_decl        ::=  "let" ["var"] name {pragma} ":=" expr ";"
```

If the `var` annotation is used, the variable may be assigned a new value using an assignment statement. Otherwise, the variable binding is constant. (The contents of the variable may still be mutable.) Formal parameters are treated as constant variable bindings and so are not assignable. The initializing expression is evaluated in a context where the name of the variable being declared and any variables declared later within the same declaration block are considered undefined. This avoids potential misunderstandings about the meaning of apparently self-referential or mutually recursive initializers while still supporting a kind of `let*` [Steele 84] variable binding sequence.

Variable declarations may appear at the top level as well as inside a method. However, the ordering of variable declarations at the top level (and consequently the order of evaluation of the initializing expressions) is less obvious. In the current UW Cecil implementation, the textual ordering of variable declarations is used to define an ordering for evaluating variable initializers. (Similarly, statements interspersed with top-level declarations are evaluated in the order given.) We would prefer a semantics that was independent of the "order" of variable declarations at the top level, so that all top-level declarations are considered unordered. Possible alternative semantics which have this property include restricting variable initialization expressions to be simple expressions without side-effects (thereby making the issue of evaluation order unimportant), eliminating variable declarations at the top level entirely, or supporting a form of on-demand at-most-once evaluation of top-level variable initializers akin to the lazy evaluation semantics of field initializers (see section 2.3.4).

2.5.3 Variable References

A variable or named object is referenced simply by naming the variable or object:

```
ref_expr        ::=  name
```

The names of objects and variables are in the same name space. Lexical scoping is used to locate the closest lexically-enclosing variable or object binding for the name.

2.5.4 Assignment Statements

Assignment statements have the following syntax:

```
assignment      ::=  name ":=" expr             assignment to a variable
                 |   assign_msg                 assignment-like syntax for messages
```

If the left-hand-side is a simple name, then the closest lexically-enclosing binding of the name is located and changed to refer to the result of evaluating the right-hand-side expression. It is an error to try to assign to an object, a formal parameter, or to a variable declared without the `var` keyword.

If the left-hand-side has the syntax of a message, then the assignment statement is really syntactic sugar for a message send, as described in section 2.5.6.

2.5.5 Literals

Cecil literal constants can be integers, floating point numbers, characters, or strings:

```
literal         ::=  integer
                 |   float
                 |   character
                 |   string
```

Literals are immutable objects.

2.5.6 Message Sends

The syntax of a message send is as follows:

```
message         ::=  msg_name "(" [exprs] ")"
exprs           ::=  expr { "," expr }
unop_msg        ::=  op_name unop_expr
binop_msg       ::=  binop_expr op_name binop_expr
```

A message is written in one of three forms:

- named prefix form, with the name of the message followed by a parenthesized list of expressions,[*]
- unary operator prefix form, with the message name listed before the argument expression, or
- infix form, with the message name in between a pair of argument subexpressions.

Normally, a message whose name begins with a letter is written in named prefix form, while a message whose name begins with a punctuation symbol is written in unary prefix form or in infix form.[†] To invoke a named message as an operator, or to invoke an operator as a named message, the name of the message is prefixed with an underscore (the leading underscore is not considered part of the message name). For example, the following two expressions both send the + message to 3 and 4:

```
3 + 4
_+(3, 4)
```

and the following two expressions both send the `bit_and` message to 3 and 4:

```
bit_and(3, 4)
3 _bit_and 4
```

The precedence and associativity of infix messages is specified through precedence declarations, described in section 2.6. The semantics of method lookup is described in section 2.7. Resends, a special kind of message send, are described in section 2.8.

Syntactic sugar exists for several common forms of messages. Dot notation allows the first argument of the message to be written first:

```
dot_msg         ::=  dot_expr "." msg_name ["(" [exprs] ")"]
```

If the message takes only one argument, the trailing parentheses can be omitted. Consequently, the following three expressions all send the `x` message to `p`:

```
x(p)
p.x()
p.x
```

The following two expressions both send the `bit_and` message to 3 and 4:

```
bit_and(3, 4)
3.bit_and(4)
```

This syntax may suggest that the first argument is more important than the others, but in fact the semantics is still that all arguments are treated uniformly, and any subset of the arguments might be dispatched at method-lookup time.

Other syntactic sugars support message sends written like assignments. Any message can appear on the left-hand-side of an assignment statement:

`assign_msg ::= lvalue_msg ":=" expr` *sugar for set_msg (exprs..., expr)*

[*] All arguments to the message must be listed explicitly; there is no implicit `self` argument.
[†] Named prefix form is always used for method declarations.

```
lvalue_msg      ::= message
                  | dot_msg
                  | unop_msg
                  | binop_msg
```

In each of these cases, the name of the message sent to carry out the "assignment" is `set_` followed by the name of the message in the `lvalue_msg` expression, and the arguments to the real message are the arguments of the `lvalue_msg` expression followed by the expression on the right-hand-side of the "assignment." So the following three expressions are all equivalent:

```
set_foo(p, q, r);
foo(p, q) := r;
p.foo(q) := r;
```

as are the following two expressions:

```
set_top(rectangle, x);
rectangle.top := x;        -- frequently used for set accessor methods
```

as are the following two expressions:

```
set_!(v, i, x);
v!i := x;
```

Note that these syntactic sugars are assignments in syntax only. Semantically, they are all messages.

2.5.7 Object Constructors

New objects are created either through object declarations (as described in section 2.1) or by evaluating object constructor expressions. The syntax of an object constructor expression is as follows:

```
object_expr    ::= "object" {relation} [field_inits]
```

This syntax is the same as for an object declaration except that no object name is specified. Object constructor expressions are analogous to object instantiation operations found in class-based languages. The only difference between named objects introduced through object declarations and anonymous objects created through object constructor expressions is that named objects have statically-known names. As a consequence, only named objects can have methods and fields attached to them and can have descendants.

2.5.8 Vector Constructors

A vector constructor expression is written as follows:

```
vector_expr    ::= "[" [exprs] "]"
```

The result of evaluating a vector constructor expression is a new immutable object that inherits from the predefined `i_vector` object and is initialized with the corresponding elements.

2.5.9 Closures

The syntax of a closure constructor expression is as follows:

```
closure_expr   ::= [ "&" "(" [closure_formals] ")" ] "{" body "}"
```

```
closure_formals::=  closure_formal { "," closure_formal }
closure_formal ::=  [name]                        formal names are optional, if never referenced
```

This syntax is identical to that of a method declaration, except that the `method` keyword and message name are replaced with the `&` symbol (intended to be suggestive of the λ symbol). If the closure takes no arguments, then the `&()` prefix may be omitted. When evaluated, a closure constructor produces two things:

- a new closure object that inherits from the predefined `closure` object, which is returned as the result of the closure constructor expression, and
- a method named `eval` whose anonymous first argument is specialized on the newly-created closure object and whose remaining arguments are those listed as formal parameters in the closure constructor expression.

As with other nested method declarations, the body of a closure's `eval` method is lexically-scoped within the scope that was active when the closure was created. However, unlike nested method declarations, the `eval` method is globally visible (as long as the connected closure object is reachable). Closures may be invoked after their lexically-enclosing scopes have returned.*

All control structures in Cecil are implemented at user level using messages and closures, with the sole exception of the `loop` primitive method described in section 2.2.3. Additionally, closures can be used to achieve much the same effect as exceptions and multiple results, so these other constructs are currently omitted from the Cecil language. Sometimes the use of closures is syntactically more verbose than a built-in language construct might be, and we are considering various alternatives for allowing programmers to define syntactic extensions to the language to provide a cleaner syntax for their user-defined control structures.

2.5.10 Parenthetical Subexpressions

A parenthesized subexpression has the same syntax as the body of a method:

```
paren_expr     ::=  "(" body ")"
```

Like the body of a method or a closure, a parenthetical subexpression introduces a new nested scope and may contain statements and local declarations.

2.6 Precedence Declarations

Cecil programmers can define their own infix binary operators. Parsing expressions with several infix operators becomes problematic, however, since the precedence and associativity of the infix operators needs to be known to parse unambiguously. For example, in the following Cecil expression

```
foo ++ bar *&&! baz *&&! qux _max blop
```

the relative precedences of the `++`, `*&&!`, and `_max` infix operators is needed, as is the associativity of the `*&&!` infix operator. For a more familiar example, we'd like the following Cecil expression (`**` represents exponentiation)

```
x + y * z ** e ** f * q
```

* In the current UW Cecil implementation, there are some caveats to the use of such non-LIFO closures. See the system documentation for additional details.

to parse using standard mathematical rules, as if it were parenthesized as follows:

```
x + ((y * (z ** (e ** f))) * q)
```

2.6.1 Previous Approaches

Most languages restrict infix operators to a fixed set, with a fixed set of precedences and associativities. This is not appropriate for Cecil, since we'd like the set of infix messages to be user-extensible.

Smalltalk defines all infix operators to be of equal precedence and left-associative. While simple, this rule differs from standard mathematical rules, sometimes leading to hard-to-find bugs. For example, in Smalltalk, the expression `3 + 4 * 5` returns 35, not 23.

Self attempts to rectify this problem with Smalltalk by specifying the relative precedence of infix operators to be undefined, requiring programmers to explicitly parenthesize their code. This avoids problems with Smalltalk's approach, but leads to many unsightly parentheses. For example, the parentheses in the following Self code are all required:

```
(x <= y) && (y <= (z + 1))
```

Self makes an exception for the case where the same binary operator is used in series, treating that case as left-associative. For example, the expression

```
x + y + z
```

parses as expected in Self. Even so, the expression

```
x ** y ** z
```

would parse "backwards" in Self, if `**` were defined. (Self uses `power:` for exponentiation, perhaps to avoid problems like this.) Also, expressions like

```
x + y - z
```

are illegal in Self, requiring explicit parenthesization.

Standard ML [Milner *et al.* 90] allows any operator to be declared prefix (called "nonfix" in SML) or infix, and infix operators can be declared left- or right-associative. Infix declarations also specify a precedence level, which is an integer from 0 (loosest binding) to 9 (tightest binding), with 0 being the default. For example, the following SML declarations are standard:

```
infix 7 *, /, div, mod;
infix 6 +, -;
infix 4 = <> < > <= >=;
infix 3 :=;
nonfix ~;
```

SML also provides special syntax to use an infix operator as a prefix operator, and vice versa.

A fixity declaration can appear wherever any other declaration can appear, and affect any parsing of expressions while the fixity declaration is in scope. Fixity declarations can be spread throughout a program, and multiple declarations can add independent operators to the same precedence level. Fixity declarations in one scope override any fixity declarations of the same operator from enclosing scopes.

One disadvantage of SML's approach is that is supports only 10 levels of precedence. It is not possible to add a new operator that is higher precedence than some operator already defined at level 9, nor is it possible to squeeze a new operator in between operators at adjacent levels. Finally, all operators at one level bind tighter than all operators at lower levels, even if the programmer might have preferred that expressions mixing operators from completely different applications be explicitly parenthesized, for readability.

2.6.2 Precedence and Associativity Declarations in Cecil

Cecil allows the precedence and associativity of infix operators to be specified by programmers through precedence declarations. The syntax of these declarations is as follows:

```
prec_decl      ::= "precedence" op_list [associativity] {precedence} ";"
associativity  ::= "left_associative" | "right_associative" | "non_associative"
precedence     ::= "below" op_list | "above" op_list | "with" op_list
op_list        ::= op_name { "," op_name }
```

For example, the following declarations might appear as part of the standard prelude for Cecil:

```
precedence ** right_associative;   -- exponentiation
precedence *, / left_associative below ** above +;
precedence +, - left_associative below * above =;
precedence =, !=, <, <=, >=, > non_associative below * above;
precedence & left_associative below = above |;
precedence | left_associative below &;
precedence % with *;
precedence ! left_associative above =;   -- array indexing
```

By default, an infix operator has its own unique precedence, unrelated to the precedence of any other infix operator, and is non-associative. Expressions mixing operators of unrelated precedences or multiple sequential occurrences of an operator that is non-associative must be explicitly parenthesized.

The effect of a precedence declaration is to declare the relationship of the precedences of several binary operators and/or to specify the associativity of a binary operator. Like SML, the information provided by a precedence declaration is used during the scope of the declaration, and declarations of the same operator at one scope override any from an enclosing scope. Two precedence declarations cannot define the precedence of the same operator in the same scope.

A precedence declaration of the form

> **precedence** *bin-op*$_1$, ..., *bin-op*$_n$
> *associativity*
> **below** *bin-op*$_{B1}$, ..., *bin-op*$_{Bn}$
> **above** *bin-op*$_{A1}$, ..., *bin-op*$_{An}$
> **with** *bin-op*$_{W1}$, ..., *bin-op*$_{Wn}$;

declares that all the *bin-op*$_i$ belong to the same precedence group, and that this group is less tightly binding than the precedence groups of any of the *bin-op*$_{Bi}$ and more tightly binding than those of the *bin-op*$_{Ai}$. If any *bin-op*$_{Wi}$ are provided, then the *bin-op*$_i$ belong to the same precedence group as the *bin-op*$_{Wi}$; all the *bin-op*$_{Wi}$ must already belong to the same precedence group. Otherwise, the *bin-op*$_i$ form a new precedence group. The associativity of the *bin-op*$_i$ is as specified by *associativity*, if present. If absent, then the associativity of the *bin-op*$_i$ is the same as the *bin-op*$_{Wi}$, if provided, and non-associative otherwise. As illustrated by the example above, the ordering of two precedence groups may be redundantly specified. Cycles in the tighter-binding-than relation on precedence groups are not allowed. All operators in the same precedence group must have the same associativity.

Taken together, precedence declarations form a partial order on groups of infix operators. Parentheses may be omitted if adjacent infix operators are ordered according to the precedence declarations, or if adjacent infix operators are from the same precedence group and the precedence group has either left- or right-associativity. Otherwise, parentheses must be included. For example, in the expression

```
v ! (i + 1) < (v ! i) + 1
```

the parentheses around `i+1` and `v!i` are required, since `!` and + are not ordered by the above precedence declarations. However, both `!` and + are more tightly binding than <, so no additional parentheses are required.

In Cecil, a declaration within a declaration block is visible throughout the block, including during textually earlier declarations within the block. This applies to precedence declarations as well, somewhat complicating parsing. The implementation strategy used in the UW Cecil system parses expressions involving binary operators into a list of operators and operands, and these lists are converted into a traditional parse tree form only after all visible declarations have been processed.

Precedence declarations apply to infix message names, not to individual methods. Multiple methods may implement the same infix message, for different kinds of arguments, but all methods with a particular name share the same precedence in a given scope.

2.7 Method Lookup

This section details the semantics of multi-method lookup, beginning with a discussion of the motivations and assumptions that led to the semantics.

2.7.1 Philosophy

All computation in Cecil is accomplished by sending messages to objects. The lion's share of the semantics of message passing specifies method lookup, and these method lookup rules typically reduce to defining a search of the inheritance graph. In single inheritance languages, method lookup is straightforward. Most object-oriented languages today, including Cecil, support multiple inheritance to allow more flexible forms of code inheritance and/or subtyping. However, multiple inheritance introduces the possibility of ambiguity during method lookup: two methods with the same name may be inherited along different paths, thus forcing either the system or the programmer to determine which method to run or how to run the two methods in combination. Multiple dispatching introduces a similar potential ambiguity even in the absence of multiple inheritance, since two methods with differing argument specializers could both be applicable but neither be uniformly more specific than the other. Consequently, the key distinguishing characteristic of method lookup in a language with multiple inheritance and/or multiple dispatching is how exactly this ambiguity problem is resolved.

Some languages resolve all ambiguities automatically. For example, Flavors [Moon 86] linearizes the class hierarchy, producing a total ordering on classes, derived from each class' local left-to-right ordering of superclasses, that can be searched without ambiguity just as in the single inheritance case. However, linearization can produce unexpected method lookup results, especially if the program contains errors [Snyder 86]. CommonLoops [Bobrow *et al.* 86] and CLOS extend this linearization approach to multi-methods, totally ordering multi-methods by prioritizing argument position, with earlier argument positions completely dominating later argument positions. Again, this removes the possibility of run-time ambiguities, at the cost of automatically resolving ambiguities that may be the result of programming errors.

Cecil takes a different view on ambiguity, motivated by several assumptions:

- We expect programmers will sometimes make mistakes during program development. The language should help identify these mistakes rather than mask or misinterpret them.
- Our experience with Self leads us to believe that programming errors that are hidden by such automatic language mechanisms are some of the most difficult and time-consuming to find.
- Our experience with Self also encourages us to strive for the simplest possible inheritance rules that are adequate. Even apparently straightforward extensions can have subtle interactions that make the extensions difficult to understand and use [Chambers *et al.* 91].
- Complex inheritance patterns can hinder future program evolution, since method lookup can depend on program details such as parent ordering and argument ordering, and it usually is unclear from the program text which details are important for a particular application.

Accordingly, we have striven for a very simple system of multiple inheritance and multiple dispatching for Cecil.

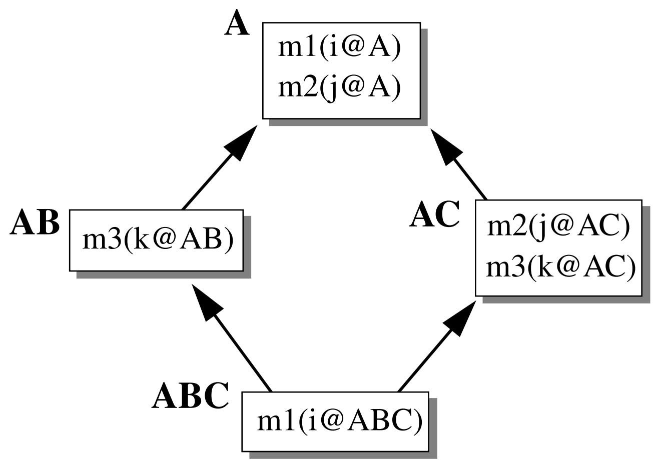
2.7.2 Semantics

Method lookup in Cecil uses a form of Touretzky's inferential distance heuristic [Touretzky 86], where children override parents. The method lookup rules interpret a program's inheritance graph as a partial ordering on objects, where being less in the partial order corresponds to being more specific: an object *A* is less than (more specific than) another object *B* in the partial order if and only if *A* is a proper descendant of *B*. This ordering on objects in turn induces an analogous ordering on the set of methods specialized on the objects, reflecting which methods override which other methods. In the partial ordering on methods with a particular name and number of arguments, one method *M* is less than (more specific than) another method *N* if and only if each of the argument specializers of *M* is equal to or less than (more specific than) the corresponding argument specializer of *N*. Since two methods cannot have the same argument specializers, at least one argument specializer of *M* must be strictly less than (more specific than) the corresponding specializer of *N*. An unspecialized argument is considered specialized on the `any` object which is an ancestor of all other objects; a specialized argument therefore is strictly less than (more specific than) an unspecialized argument. The ordering on methods is only partial since ambiguities are possible.

Given the partial ordering on methods, method lookup is straightforward. For a particular message send, the system constructs the partial ordering of methods with the same name and number of arguments as the message. The system then throws out of the ordering any method that has an argument specializer that is not equal to or an ancestor of the corresponding actual argument passed in the message; such a method is not applicable to the actual call. Finally, the system attempts to locate the single most-specific method remaining, i.e., the method that is least in the partial order over applicable methods. If no methods are left in the partial order, then the system reports a "message not understood" error. If more than one method remains in the partial order, but there is no single method that overrides all others, then the system reports a "message ambiguous" error. Otherwise, there is exactly one method in the partial order that is strictly more specific than all other methods, and this method is returned as the result of the message lookup.

2.7.3 Examples

For example, consider the following inheritance graph (containing only singly-dispatched methods for the moment):



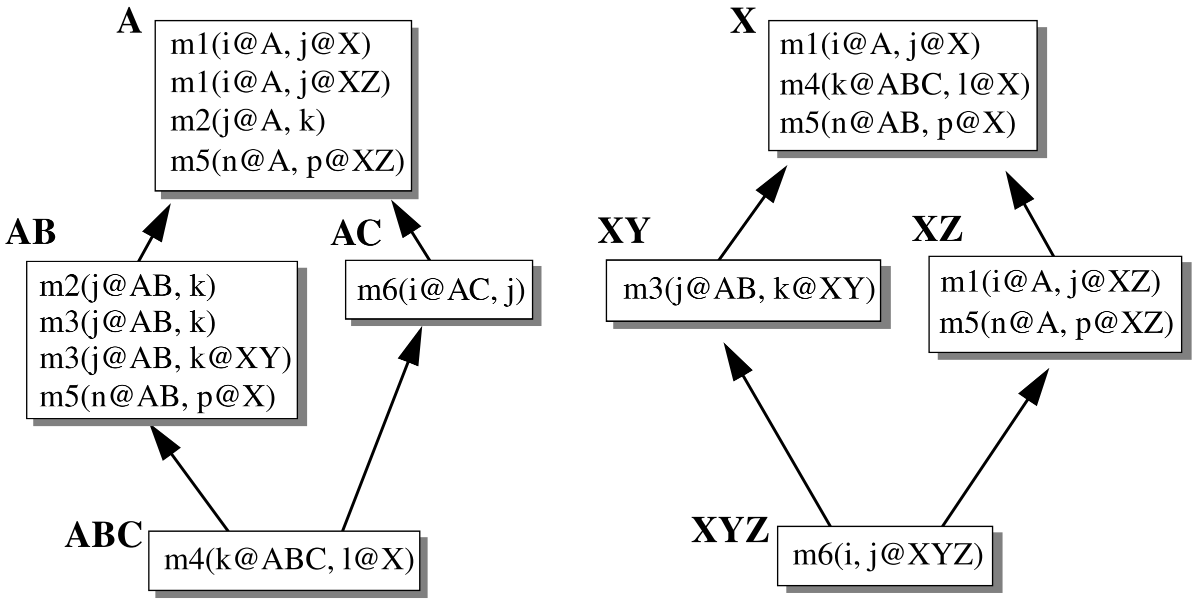
The partial ordering on objects in this graph defines `ABC` to be more specific than either `AB` or `AC`, and both `AB` and `AC` are more specific than `A`. Thus, methods defined for `ABC` will be more specific (will override) methods defined in `A`, `AB`, or `AC`, and methods defined in either `AB` or `AC` will be more specific (will override) methods defined in `A`. The `AB` and `AC` objects are mutually unordered, and so any methods defined for both `AB` and `AC` will be unordered.

If the message `m1` is sent to the `ABC` object, both the implementation of `m1` whose formal argument is specialized on the `ABC` object and the implementation of `m1` specialized on `A` will apply, but the method specialized on `ABC` will be more specific than the one specialized on `A` (since `ABC` is more specific than `A`), and so `ABC`'s `m1` will be chosen. If instead the `m1` message were sent to the `AB` object, then the version of `m1` specialized on the `A` object would be chosen; the version of `m1` specialized on `ABC` would be too specific and so would not apply.

If the `m2` message is sent to `ABC`, then both the version of `m2` whose formal argument is specialized on `A` and the one whose formal is specialized on `AC` apply. But the partial ordering places the `AC` object ahead of the `A` object, and so `AC`'s version of `m2` is selected.

If the `m3` message is sent to `ABC`, then both `AB`'s and `AC`'s versions of `m3` apply. Neither `AB` nor `AC` is the single most-specific object, however; the two objects are mutually incomparable. Since the system cannot select an implementation of `m3` automatically without having a good chance of being wrong and so introducing a subtle bug, the system therefore reports an ambiguous message error. The programmer then is responsible for resolving the ambiguity explicitly, typically by writing a method in the child object which resends the message to a particular ancestor; resends are described in section 2.8. Sends of `m3` to either `AB` or `AC` would be unambiguous, since the other method would not apply.

To illustrate these rules in the presence of multi-methods, consider the following inheritance graph (methods dispatched on two arguments are shown twice in this picture):



Methods `m1` in `A` and `m3` in `AB` illustrate that multiple methods with the same name and number of arguments may be associated with (specialized on) the same object, as long as some other arguments are specialized differently. The following table reports the results of several message sends using this inheritance graph.

message	invoked method or error	explanation
m1(ABC, XYZ)	m1(i@A, j@XZ)	XZ overrides X
m2(ABC, XYZ)	m2(j@AB, k)	AB overrides A
m3(ABC, XYZ)	m3(j@AB, k@XY)	XY overrides unspecialized
m4(AB, XY)	"message not understood"	ABC too specific for AB ⇒ no applicable method
m5(ABC, XYZ)	"message ambiguous"	AB overrides A but XZ overrides X ⇒ no single most-specific applicable method
m6(ABC, XYZ)	"message ambiguous"	AC overrides unspecialized but XYZ overrides unspecialized ⇒ no single most-specific method

2.7.4 Strengths and Limitations

The partial ordering view of multiple inheritance has several desirable properties:

- It is simple. It implements the intuitive rule that children override their parents (they are lesser in the partial ordering), but does not otherwise order parents or count inheritance links or invoke other sorts of complicated rules.

- Ambiguities are not masked. These ambiguities are reported back to the programmer at message lookup time before the error can get hidden. If the programmer has included static type declarations, the system will report the ambiguity at type-check-time.
- This form of multiple inheritance is robust under programming changes. Programmers can change programs fairly easily, and the system will report any ambiguities which may arise because of programming errors. More complex inheritance rules tend to be more brittle, possibly hindering changes to programs that exploit the intricacies of the inheritance rules and hiding ambiguities that reflect programming errors.
- Cecil's partial ordering view of multiple inheritance does not transform the inheritance graph prior to determining method lookup, as does linearization. This allows programmers to reason about method lookup using the same inheritance graph that they use to write their programs.

Of course, there may be times when having a priority ordering over parents or over argument positions would resolve an ambiguity automatically with no fuss. For these situations, it might be nice to be able to inform the system about such preferences. Self's prioritized multiple inheritance strategy can blend ordered and unordered inheritance, but it has some undesirable properties (such as sometimes preferring a method in an ancestor to one in a child) and interacts poorly with resends and dynamic inheritance.* It may be that Cecil could support something akin to prioritized multiple inheritance (and perhaps even a prioritized argument list), but use these preferences as a last resort to resolving ambiguities; only if ambiguities remain after favoring children over parents would preferences on parents or argument position be considered. Such as design appears to have fewer drawbacks than Self's approach or CLOS's approach while gaining most of the benefits.

An alternative approach might be to support explicit declarations that one method is intended to override another method. These declarations would add relations to the partial order over methods, potentially resolving ambiguities. This approach has the advantage that it operates directly on the method overriding relationship rather than on parent order or the like which only indirectly affects method overriding relationships. Moreover, this approach can only resolve existing ambiguities, not change any existing overriding relationships, thereby making it easier to reason about the results of method lookup. To implement this approach, a mechanism for naming particular methods (e.g., the method's name and its specializers) must be added.

2.7.5 Multiple Inheritance of Fields

In other languages with multiple inheritance, in addition to the possibility of name clashes for methods, the possibility exists for name clashes for instance variables. Some languages maintain separate copies of instance variables inherited from different classes, while other languages merge like-named instance variables together in the subclass. The situation is simpler in Cecil, since all access to instance variables is through field accessor methods. An object (conceptually at least) maintains space for each inherited copy-down field, independently of their names (distinct fields with the same name are not merged automatically). Accesses to these fields are mediated by their accessor methods, and the normal multiple inheritance rules are used to resolve any ambiguities

* Recently, Self's multiple inheritance semantics has been greatly simplified, eliminating prioritized inheritance. Self's rules are now similar to Cecil's, except that Self omits the "children-override-parents" global rule. This has the effect of declaring as ambiguous messages such as `m2(ABC)` in the first example in section 2.7.3.

among like-named field accessor methods. In particular, a method in the child with the same name as a field accessor method could send directed resend messages (described later in section 2.8) to access the contents of one or the other of the ambiguous fields.

2.7.6 Cyclic Inheritance

In the current version of Cecil, inheritance is required to be acyclic. However, cycles in the inheritance graph would be easy to allow. Instead of defining a partial order over objects, inheritance would define a preorder, where all objects participating in a cycle are considered to inherit from all other objects in the cycle, but not be more specific than any of them. This preorder on inheritance induces a corresponding preorder on methods. The same rules for successful method lookup still apply: a single most specific method must be found. If two methods are in a cycle in the method specificity preorder, then neither is more specific than the other. In effect, objects can participate in inheritance cycles if they define disjoint sets of methods. This design of "mutually-recursive" objects could be used to factor a single large object into multiple separate objects, each implementing a separate facet of the original object's implementation.

2.7.7 Method Lookup and Lexical Scoping

Since methods may be declared both at the top level and nested inside of methods, method lookup must take into account not only which methods are more specialized than which others but also which methods are defined in more deeply-nested scopes. The interaction between lexical scoping and inheritance becomes even more significant in the presence of modules as described in section 5.

The view of lexically-nested methods in Cecil is that nested methods *extend* the inheritance graph defined in the enclosing scope, rather than *override* it. We call this "porous" lexical scoping of methods, since the enclosing scope filters through into the nested scope. When performing method lookup for a message within some nested scope, the set of methods under consideration are those declared in the current scope plus any methods defined in lexically-enclosing scopes. If a local method has the same name, number of arguments, and argument specializers as a method defined in an enclosing scope, then the local method shadows (replaces) the method in the enclosing scope. Additionally, any object declarations or object extension declarations in the local scope are added to those declarations and extensions defined in enclosing scopes. Once this augmented inheritance graph is constructed, method lookup proceeds as before without reference to the scope in which some object or method is defined.

Other languages, such as BETA [Kristensen *et al.* 87], take the opposite approach, searching for a matching method in one scope before proceeding to the enclosing scope. If a matching method is found in one scope, it is selected even if a more specialized method is defined in an enclosing scope. More experience is needed to judge which of these alternatives is preferable. Cecil's approach gets some advantage by distinguishing variable references, which always respect only the lexical scope, from field references, which always are treated as message sends and primarily respect inheritance links. BETA uses the same syntax to access both global variables and inherited instance variables, making the semantics of the construct somewhat more complicated.

Nested methods can be used to achieve the effect of a `typecase` statement as found in other languages, including Trellis and Modula-3 [Nelson 91, Harbison 92]. For example, to test the implementation of an object, executing different code for each case, the programmer could write something like the following:

```
method test(x) {
   method typecase(z@obj1) { (-- code for case where x inherits from obj1 --) }
   method typecase(z@obj2) { (-- code for case where x inherits from obj2 --) }
   method typecase(z@obj3) { (-- code for case where x inherits from obj3 --) }
   method typecase(z)      { (-- code for default case --) }
   typecase(x);
}
```

In the example, `obj1`, `obj2`, and `obj3` may be related in the inheritance hierarchy, in which case the most-specific case will be chosen. If no case applies or no one case is most specific, then a "message not understood" or an "ambiguous message" error will result. These results fall out of the semantics of method lookup. By nesting the `typecase` methods inside the calling method, the method bodies can access other variables in the calling method through lexical scoping, plus the scope of the temporary `typecase` methods is limited to that particular method invocation. Eiffel's reverse assignment attempt and Modula-3's `NARROW` operation can be handled similarly.

2.7.8 Method Invocation

If method lookup is successful in locating a single target method without error, the method is invoked. A new activation record is created, formals in the new scope are initialized with actuals, the statements within the body of the method are executed in the context of this new activation record (or the primitive method is executed, or the field accessor method is executed), and the result of the method (possibly `void`) is returned to the caller.

2.8 Resends

Most existing object-oriented languages allow one method to override another method while preserving the ability of the overriding method to invoke the overridden version: Smalltalk-80 has `super`, CLOS has `call-next-method`, C++ has qualified messages using the `::` operator, Trellis has qualified messages using the `'` operator, and Self has undirected and directed `resend` (integrating unqualified `super`-like messages and qualified messages). Such a facility allows a method to be defined as an incremental extension of an existing method by overriding it with a new definition and invoking the overridden method as part of the implementation of the overriding method. This same facility also allows ambiguities in message lookup to be resolved by explicitly forwarding the message to a particular ancestor.

Cecil includes a construct for resending messages that adapts the Self undirected and directed resend model to the multiply-dispatched case. The syntax for a resend is as follows:

```
resend        ::= "resend" [ "(" resend_args ")" ]
resend_args   ::= resend_arg { "," resend_arg }
resend_arg    ::= expr                  corresponding formal of sender must be
                                           unspecialized
               |  name                  undirected resend (name is a specialized formal)
               |  name "@" named_object directed resend (name is a specialized formal)
```

The purpose of the resend construct is to allow a method to invoke one of the methods that the resending method overrides. Consequently, only methods with the same name and number of arguments as the resending method whose argument specializers are ancestors of the resending method's argument specializers are considered possible targets of a resend.

To invoke an overridden method, the normal prefix message sending syntax is used but with the following changes and restrictions:

- Syntactically, the name of the message is the keyword `resend`; semantically, the name of the message is implicitly the same as the name of the sending method.
- The number of arguments to the message must be the same as for the sending method.
- All specialized formal arguments of the resending method must be passed through unchanged as the corresponding arguments to the resend.

As a syntactic convenience, if all formals of the sender are passed through as arguments to the resend unchanged, then the simple `resend` keyword without an argument list is sufficient.

The semantics of a resent message are similar to a normal message, except that only methods that are less specific than the resending method in the partial order over methods are considered possible matches; this has the effect of "searching upwards" in the inheritance graph to find the single most-specific method that the resending method overrides. The restrictions on the name, on the number of arguments, and on passing specialized objects through unchanged ensures that the methods considered as candidates are applicable to the name and arguments of the send. Single-dispatching languages often have similar restrictions: Smalltalk-80 requires that the implicit `self` argument be passed through unchanged with the `super` send, and CLOS's `call-next-method` uses the same name and arguments as the calling method.

For example, the following illustrates how resends may be used to provide incremental extensions to existing methods:

```
object colored_rectangle isa rectangle;
    field color(@colored_rectangle);
    method display(r@colored_rectangle, d@output_device) {
        d.color := r.color;  -- set the right color for this rectangle
        resend;  -- do the normal rectangle drawing; sugar for resend(r, d)
    }
```

Resends may also be used to explicitly resolve ambiguities in the method lookup by filtering out undesired methods. Any of the required arguments to a resend (those that are specialized formals of the resending method) may be suffixed with the `@` symbol and the name of an ancestor of the corresponding argument specializer. This restricts methods considered in the resulting partial order to be those whose corresponding argument specializers (if present) are equal to or ancestors of the object named as part of the resend.

To illustrate, the following method resolves the ambiguity of `height` for `vlsi_cell` in favor of the `rectangle` version of height:*

* This example was adapted from Ungar and Smith's original Self paper [Ungar & Smith 87].

```
object rectangle;
   field height(@rectangle);

object tree_node;
   method height(t@tree_node) { 1 + height(t.parent) }

object vlsi_cell isa rectangle, tree_node;
   method height(v@vlsi_cell) { resend(v@rectangle) }
```

This model of undirected and directed resends is a simplification of the Self rules, extended to the multiple dispatching case. Self's rules additionally support prioritized multiple inheritance and dynamic inheritance, neither of which is present in Cecil. Self also allows the name and number of arguments to be changed as part of the resend. In some cases, it appears to be useful to be able to change the name of the message as part of the resend. For example, it might be useful to be able to provide access to the `tree_node` version of the `height` method under some other name, but this currently is not possible in Cecil. We are investigating possible semantics for resends where the name of the message is changed, so that both ambiguously-inherited methods can be invoked.

As demonstrated by Self, supporting both undirected and directed resends is preferable to just supporting directed resends as does C++ and Trellis, since the resending code does not need to be changed if the local inheritance graph is adjusted. Since CLOS does not admit the possibility of ambiguity, it need only support undirected resends (i.e., `call-next-method`); there is no need for directed resends.

2.9 Files and Include Declarations

The current UW Cecil implementation is file-based. The compiler is given a single file name, naming the file containing the program to compile. To include other files into the program, a file can include an include declaration, at the global scope:

```
include_decl   ::= "include" file_name ";"
file_name      ::= string
included_file  ::= file_body
```

The effect of an include declaration is to include the declarations from the named file into the current scope. The named file must have the syntax of a single declaration block. File inclusion is idempotent: redundant inclusions of a file into a particular scope have no effect.

2.10 Pragmas

Pragmas can be used by the Cecil programmer to provide additional information and implementation directives to the Cecil implementation. The set of recognized pragmas and their interpretation is implementation-dependent. A description of some of the pragmas supported by the UW Cecil implementation is provided in its documentation.

Pragmas are written as follows:

```
pragma         ::= "(**" expr "**)"
```

The body of a pragma uses the syntax of a Cecil expression, but its interpretation is different (and implementation-dependent). Currently, pragmas may appear as part of most Cecil declarations. In the future, pragmas will likely be able to be provided for any declaration and any expression.

3 Static Types

Cecil supports a static type system which is layered on top of the dynamically-typed core language. The type system's chief characteristics are the following:

- Type declarations specify the interface required of an object stored in a variable or returned from a method, without placing any constraints on its representation or implementation.
- Argument specializers for method dispatching are separate from type declarations, enabling the type system to contain as special cases type systems for traditional single-dispatching and non-object-oriented languages.
- Code inheritance can be distinct from subtyping, but the common case where the two are parallel requires only one set of declarations.
- The type checker can detect statically when a message might be ambiguously defined as a result of multiple inheritance or multiple dispatching. It does not rely on the absence of ambiguities to be correct.
- The type system can check programs statically despite Cecil's classless object model.
- Type declarations are optional, providing partial language support for mixed exploratory and production programming.
- Parameterized objects, types, and methods support flexible forms of parametric polymorphism, complementing the inclusion polymorphism supported through subtyping.

This section describes Cecil's static type system in the absence of parameterization; section 4 extends this section to cope with parameterized objects and methods. Section 3.1 presents the major goals for the type system. Section 3.2 presents the overall structure of the type system. Sections 3.3, 3.4, and 3.5 describe the important kinds of declarations provided by programmers that extend the base dynamically-typed core language described in section 2. Sections 3.6, 3.7, 3.8, and 3.9 detail the type-checking rules for the language. Section 3.10 describes how the language supports mixed statically- and dynamically-typed code.

3.1 Goals

Static type systems historically have addressed many concerns, ranging from program verification to improved run-time efficiency. Often these goals conflict with other goals of the type system or of the language, such as the conflict between type systems designed to improve efficiency and type systems designed to allow full reusability of statically-typed code.

The Cecil type system is designed to provide the programmer with extra support in two areas: machine-checkable documentation and early detection of some kinds of programming errors. The first goal is addressed by allowing the programmer to annotate variable declarations, method arguments, and method results with explicit type declarations. These declarations help to document the interfaces to abstractions, and the system can ensure that the documentation does not become out-of-date with respect to the code it is documenting. Type inference may be useful as a programming environment tool for introducing explicit type declarations into untyped programs.

The Cecil type system also is intended to help detect programming errors at program definition time rather than later at run-time. These statically-detected errors include "message not understood," "message ambiguous," and "uninitialized field accessed." The type system is designed to verify that there is no possibility of any of the above errors in programs, guaranteeing type safety but possibly reporting errors that are not actually a problem for any particular execution of the program. To make work on incomplete or inconsistent programs easier, type errors are considered warnings, and the programmer always is able to run a program that contains type errors. Dynamic type checking at run-time is the final arbiter of type safety.

Cecil's type system is not designed to improve run-time efficiency. For object-oriented languages, the goal of reusable code is often at odds with the goal of efficiency through static type declarations; efficiency usually is gained by expressing additional representational constraints as part of a type declaration that artificially limit the generality of the code. Cecil's type system strives for specification only of those properties of objects that affect program correctness, i.e., the interfaces to objects, and not of how those properties are implemented. To achieve run-time efficiency, Cecil relies on advanced implementation techniques [e.g., Dean & Chambers 94, Dean *et al.* 95a, Dean *et al.* 95b, Grove *et al.* 95, Grove 95].

Finally, Cecil's type system is *descriptive* rather than *prescriptive*. The semantics of a Cecil program are determined completely by the dynamically-typed core of the program. Type declarations serve only as documentation and partial redundancy checks, and they do not influence the execution behavior of programs. This is in contrast to some type systems, such as Dylan's, where an argument type declaration can mean a run-time type check in some contexts and act as a method lookup specializer in other contexts.

The design of the Cecil type system is affected strongly by certain language features. Foremost of these is multi-methods. Type systems for single dispatching languages are based on the first argument of a message having control, consulting its static type to determine which operations are legal. In Cecil, however, any subset of the arguments to a method may be specialized, leaving the others unspecialized. This enables Cecil to easily model both procedure-based non-object-oriented languages and singly-dispatched object-oriented languages as important special cases, but it also requires the type system to treat specialized arguments differently than unspecialized arguments.

3.2 Types and Signatures

A *type* in Cecil is an abstraction of an object. A type represents a machine-checkable interface and an implied but unchecked behavioral specification, and all objects that *conform* to the type must support the type's interface and promise to satisfy its behavioral specification. One type may claim to be a *subtype* of another, in which case all objects that conform to the subtype are guaranteed also to conform to the supertype. The type checker verifies that the interface of the subtype conforms to the interface of the supertype, but the system must accept the programmer's promise that the subtype satisfies the implied behavioral specification of the supertype. Subtyping is explicit in Cecil just so that these implied behavior specifications can be indicated. A type may have multiple direct supertypes, and in general the explicit subtyping relationships form a partial order. As described in subsection 3.4, additional type constructors plus a few special types expand the type partial order to a full lattice.

A *signature* in Cecil is an abstraction of a collection of overloaded methods, specifying both an interface (a name, a sequence of argument types, and a result type) and an implied but uncheckable behavioral specification. The interface of a type is defined as the set of signatures that mention that type as one of their argument or result types.

For example, the following types and signatures describe the interface to lists of integers:

```
type int_list subtypes int_collection;
    signature is_empty(int_list):bool;
    signature length(int_list):int;
    signature do(int_list, &(int):void):void;
    signature pair_do(int_list, int_list, &(int,int):void):void;
    signature prepend(int, int_list):int_list;
```

Types and signatures represent a contract between clients and implementors that enable message sends to be type-checked. The presence of a signature allows clients to send messages whose argument types are subtypes of the corresponding argument types in the signature, and guarantees that the type of the result of such a message will be a subtype of the result type appearing in the signature. Any message not covered by some signature will produce a “message not understood” error. Signatures also impose constraints on the implementations of methods, in order to make the above guarantees to clients. The collection of methods implementing a signature must be *conforming*, *complete*, and *consistent*:

- Conformance implies that each method implementing a signature has unspecialized argument types that are supertypes of the corresponding argument types of the signature and a result type that is a subtype of the signature’s result type; conformance is Cecil’s version of the standard contravariance rule found in singly-dispatched statically-typed languages.
- Completeness implies that the methods must handle all possible argument types that might appear at run-time as an argument to a message declared legal by the signature.
- Consistency implies that the methods must not be ambiguous for any combination of run-time arguments.

Checking completeness and consistency is the subject of section 3.6.2.

In a singly-dispatched language, each type has an associated set of signatures that defines the interface to the type. This association relies on the asymmetry of message passing in such languages, where only the receiver argument impacts method lookup. When type-checking a singly-dispatched message, the type of the receiver determines the set of legal operations, i.e., the set of associated signatures. If a matching signature is found, then the message will be understood at run-time; the static types of the remaining message arguments is checked against the static argument types listed in the signature. For Cecil, we wish to avoid the asymmetry of this sort of type system. Consequently, we view a signature as associated with each of its argument types, not just the first, much as a multi-method in Cecil is associated with each of its argument specializer objects. For example, the `prepend` signature above is considered part of both the `int` type and the `int_list` type.

In most object-oriented languages, the code inheritance graph and the subtyping graph are joined: a class is a subtype of another class if and only if it inherits from that other class. Sometimes this constraint becomes awkward [Snyder 86], for example when a class supports the interface of some other class or type, but does not wish to inherit any code. Other times, a class reusing another class's code cannot or should not be considered a subtype; covariant redefinition as commonly occurs in Eiffel programs is one example of this case [Cook 89].

To increase flexibility and expressiveness, Cecil separates subtyping from code inheritance. Types and signatures can be declared independently of object representations and method implementations. However, since in most cases the subtyping graphs and the inheritance graphs are parallel, requiring programmers to define and maintain two separate hierarchies would become too onerous to be practical. To simplify specification and maintenance of the two graphs, in Cecil the programmer can specify both a type and a representation, and the associated subtyping, conformance, and inheritance relations, with a single declaration. Similarly, a single declaration can be used to specify both a signature and a method implementation. In this way we hope to provide the benefits of separating subtyping from code inheritance when it is useful, without imposing additional costs when the separation is not needed.

3.3 Type and Signature Declarations

Variable declarations and formal arguments and results of methods, closures, and fields may be annotated with type declarations. The syntax of declarations is extended to include some new declarations:

```
decl            ::=  let_decl
                  |  tp_decl
                  |  type_ext_decl
                  |  object_decl
                  |  obj_ext_decl
                  |  predicate_decl
                  |  disjoint_decl
                  |  cover_decl
                  |  divide_decl
                  |  signature_decl
                  |  method_decl
                  |  field_sig_decl
                  |  field_decl
                  |  precedence_decl
                  |  include_decl
                  |  prim_decl
```

In this and subsequent syntax specifications, changes to specifications as described in section 2 are in boldface.

The following example illustrates some of the extensions:

```
object list;
   method is_empty(l@:list):bool { l.length = 0 }
   signature length(l:list):int;
```

```
    signature do(l:list, closure:&(int):void):void;

    signature pair_do(l1:list, l2:list, closure:&(int,int):void):void;

    method prepend(x:int, l@:list):list {
       object inherits cons { head := x, tail := l } }

    method copy_reverse(l:list):list {
       let var l2:list := nil;
       do(l, &(x:int){ l2 := prepend(x, l2); };
       l2 }

 representation cons isa list;
    field head(@:cons):int;
    field tail(@:cons):list;
```

3.3.1 Type Declarations

New user-defined types are introduced with type declarations of the following form (ignoring parameterization and encapsulation aspects):

```
tp_decl         ::=  "type" name {type_relation} ";"
type_relation   ::=  "subtypes" types
types           ::=  type { "," type }
```

The new type is considered to be a subtype of each of the types listed in the `subtypes` clause. The induced subtype relation over used-defined types must be a partial order (i.e., it cannot contain cycles).*

Type names are interpreted in a name space distinct from that of objects and variables and that of message names. A type, an object, and a method may all be named `list` unambiguously.

3.3.2 Representation and Object Declarations

New user-defined objects are introduced with representation declarations of the following form (again, ignoring parameterization and encapsulation):

```
object_decl     ::=  rep_role rep_kind name {relation} [field_inits] ";"
rep_kind        ::=  "representation"          declares an object implementation
                |    "object"                  declares an object type and implementation
relation        ::=  "subtypes" types          impl conforms to type, type subtypes from type
                |    "inherits" parents        impl inherits from impl
                |    "isa" parents             impl inherits from impl, type subtypes from type
```

Representation roles will be described in section 3.5.

If the `representation` keyword is used, the declaration introduces a new object representation. This object inherits from the objects named in each `inherits` clause and conforms to the types named in each `subtypes` clause. As mentioned in section 2.1.1, the inheritance graph cannot have cycles.

An `isa` clause is syntactic sugar for both a `subtypes` clause and an `inherits` clause, i.e., sugar for the case where inheritance and subtyping are parallel. So the following declaration

---

* It is not strictly necessary to restrict subtyping to a partial order. Cycles in the subtypes relation could be allowed, producing a preorder over types instead of a partial order. This would have any type in a cycle being a subtype of and therefore substitutable for every other type in the cycle. In essense, all types in a cycle would be equivalent as far as the type checker was concerned.

```
representation cons isa list;
```

is syntactic sugar for the declaration

```
representation cons inherits list subtypes typeof(list);
```

where *typeof*`(list)` represents the most-specific type(s) to which the `list` object conforms (*typeof* is not legal Cecil syntax).

If the `object` keyword is used, then the declaration is syntactic sugar for the pair of an object representation declaration and a type declaration. A declaration of the form

object *name* **inherits** $name_{i1}$, $name_{i2}$, ..., $name_{in}$
subtypes $name_{s1}$, $name_{s2}$, ..., $name_{sm}$
isa $name_{b1}$, $name_{b2}$, ..., $name_{bk}$;

is syntactic sugar for the following two declarations:

type *name* **subtypes** $name_{s1}$, $name_{s2}$, ..., $name_{sm}$,
typeof($name_{b1}$), *typeof*($name_{b2}$), ..., *typeof*($name_{bk}$) ;

representation *name*
inherits $name_{i1}$, $name_{i2}$, ..., $name_{in}$, $name_{b1}$, $name_{b2}$, ..., $name_{bk}$
subtypes *name* ;

Both the object and the type have the same name, but there is no potential for ambiguity since object and type names are resolved in separate name spaces. The new type subtypes from all the types listed in the `subtypes` clause and from the types to which the objects in the `isa` clause of the original declaration conform. The new object representation conforms to the new type and inherits from the object representations listed in the `inherits` and `isa` clauses of the original declaration.

The `object` and `isa` syntactic sugars are designed to make it easy to specify the inheritance and subtyping properties of an object/type pair for the common case that code inheritance and subtyping are parallel. We expect that in most programs, only `object` and `isa` declarations will be used; `type`, `representation`, `inherits`, and `subtypes` declarations are intended for relatively rare cases where finer control over inheritance and subtyping are required.

Object constructor expressions are similarly extended with representation roles, representation kinds, and subtyping relationships:

```
object_expr    ::= rep_role rep_kind {relation} [field_inits]
```

In an object constructor expression, both the `representation` keyword and the `object` keyword have the same effect; the presence or absence of an anonymous type is immaterial.

Representations often add new, implementation-specific operations. For example, the `cons` representation defined earlier introduced the `head` and `tail` fields. To be able to send messages that access these new features, a type must exist that includes the appropriate signatures. If `cons` were only a representation, then a separate type would need to be defined that included signatures for `head` and `tail`. To avoid this extra step, a `representation` declaration, like an `object` declaration, introduces a corresponding type. Unlike an object declaration, however, the type derived from a representation declaration is anonymous. It can only be referenced indirectly through the *typeof* internal function that specifies the semantics of the `isa` and `@:` syntactic sugars (section 3.3.5 describes the `@:` sugar). Consequently, no variables or unspecialized formals may be declared to be of the anonymous type, and no types may be declared to be subtypes of the anonymous type. This enables object representations to be defined that are not treated as first-class types; the programmer has control over which types are intended to be used in type declarations.

3.3.3 Type and Object Extension Declarations

As described in section 2.1.3, objects can be extended with new inheritance relations after they have been defined. In a similar fashion, types can be extended with new subtyping relations using type extension declarations of the following form:

```
type_ext_decl  ::= "extend" "type" named_type {type_relation} ";"
```

The syntax of object extension declarations is extended to support augmenting either just an object representation or both an representation and a type:

```
obj_ext_decl   ::= "extend" extend_kind named_object
                       {relation} [field_inits] ";"
extend_kind    ::= "representation"          extend representation
                 |  ["object"]                extend both type and representation
```

If the extension uses the `representation` keyword, then the named representation is extended with the appropriate inheritance and conformance relations. Otherwise, both the representation and the type that are named by the extension are updated. A declaration of the form

extend object *name* **inherits** $name_{i1}$, $name_{i2}$, ..., $name_{in}$
subtypes $name_{s1}$, $name_{s2}$, ..., $name_{sm}$
isa $name_{b1}$, $name_{b2}$, ..., $name_{bk}$;

is syntactic sugar for the following two declarations:

extend type *name* **subtypes** $name_{s1}$, $name_{s2}$, ..., $name_{sm}$,
typeof($name_{b1}$), *typeof*($name_{b2}$), ..., *typeof*($name_{bk}$) ;

extend representation *name*
inherits $name_{i1}$, $name_{i2}$, ..., $name_{in}$, $name_{b1}$, $name_{b2}$, ..., $name_{bk}$
subtypes *name* ;

It is an error if there does not exist both a representation and a type with the corresponding name.

Allowing types to be extended externally to have additional supertypes allows third-parties to integrate separately-developed libraries without modifying the separate libraries directly [Hölzle 93].

3.3.4 Signature Declarations

Signatures can be declared using the following syntax:

```
signature_decl ::= "signature" method_name
                       "(" [arg_types] ")" [type_decl] ";"
arg_types      ::= arg_type { "," arg_type }
arg_type       ::= [[name] ":"] type
type_decl      ::= ":" type
```

The names of formals in a signature are for documentation purposes only; they do not impact the meaning of the signature nor do they have any effect during type checking.

Signatures can also be declared in a field-like notation, as follows:

```
field_sig_decl ::= ["var"] "field" "signature" method_name
                        "(" arg_type ")" [type_decl] ";"
```

A field signature declaration of the form

var field signature *name* (*type*) :$type_R$;

is syntactic sugar for the following two declarations:

signature *name* (*type*) :$type_R$;

signature set_*name* (*type*, $type_R$) :void;

A field signature declaration does not require that implementations of the resulting signatures be fields, only that their interface "looks" like they could be implemented by fields. If the `var` keyword is omitted, then the second set accessor signature is not generated.

3.3.5 Implementation and Method Declarations

The syntax of method implementations is extended in the following way to accommodate static types:

method_decl	::=	**impl_kind** method_name "(" [formals] ")" **[type_decl]** {pragma} "{" (body | prim_body) "}" [";"]	
impl_kind	::=	["method"] "implementation"	*declares a method implementation*
	|	"method"	*declares a method signature and implementation*
specializer	::=	[location] **[type_decl]**	*specialized formal*
	|	**"@" ":" object**	*sugar for @object :object*

Formal parameters of the method implementation and the result of the method implementation can be given explicit type declarations.

If the `implementation` keyword is used, the declaration introduces a new method implementation. If, however, the `method` keyword alone is used, the declaration is syntactic sugar for both a method implementation declaration and a signature declaration. A declaration of the form

method *name* (x_1@obj_1:$type_1$, ..., x_N@obj_N:$type_N$) :$type_R$ { *body* }

is syntactic sugar for the following two declarations:

signature *name* ($type_1$, ..., $type_N$) :$type_R$;

implementation *name* (x_1@obj_1:$type_1$, ..., x_N@obj_N:$type_N$) :$type_R$ { *body* }

As explained in section 2.2.1, if any of the obj_i are omitted, they default to `any`.

A formal in a method or field declaration can be specified with the *x*@:*object* syntax. This syntax is shorthand for *x*@*object*:*typeof*(*object*).

3.3.6 Field Implementation Declarations

Field implementation declarations are similarly extended to accommodate static types:

```
field_decl     ::= ["shared"] ["var"] "field" field_kind method_name
                        "(" formal ")" [type_decl] {pragma} [":=" expr] ";"
```

```
field_kind      ::= empty                      declare accessor method impl(s) and sig(s)
                  |   "implementation"         declare just accessor method implementation(s)
```

If the `implementation` keyword is used, then the declaration introduces a field get accessor method implementation, and also a set accessor method implementation if the field is declared with the `var` keyword. The result type of the field is used as the type of the second argument of the set accessor method; the result type of the set accessor method is `void`.

If the plain `field` keyword is used, then the field declaration is syntactic sugar for a field implementation declaration and a field signature declaration. A field declaration of the form

shared? var? **`field`** *name*`(`*x*`@`*obj*`:`*type*`)`:*type*$_R$ `:=` *expr*`;`

where *shared?* is either the `shared` keyword or empty and *var?* is the `var` keyword or empty, is syntactic sugar for the following declarations:

shared? var? **`field implementation`** *name*`(`*x*`@`*obj*`:`*type*`)`:*type*$_R$ `:=` *expr*`;`

var? **`field signature`** *name*`(`*type*`)`:*type*$_R$`;`

The field signature declaration is itself syntactic sugar for one or two signature declarations, depending on whether the `var` keyword was used.

3.3.7 Other Type Declarations

In addition to allowing the formals and results of methods and fields to be annotated with explicit type declarations, variable declarations and closure arguments and results can be annotated with explicit type declarations:

```
let_decl       ::= "let" ["var"] name [type_decl] {pragma} ":=" expr ";"
closure_expr   ::= [ "&" "(" [closure_formals] ")" [type_decl] ] "{" body "}"
closure_formal ::= [name] [type_decl]         formal names are optional, if never referenced
```

If the result type of a closure is omitted, instead of defaulting to `dynamic` as described in section 3.4.1, the result type is inferred from the type of the result expression in the closure's body. Similarly, if the type of a constant local variable is omitted, it is inferred from the type of its initializing expression; mutable variables and global variables should be given explicit types to avoid dynamic type checking.

3.3.8 Discussion

Subtyping and conformance in Cecil is explicit, in that the programmer must explicitly declare that an object conforms to a type and that a type is a subtype of another type. These explicit declarations are verified as part of type checking to ensure that they preserve the required properties of conformance and subtyping. Explicit declarations are used in Cecil instead of implicit inference of the subtyping relations (*structural subtyping*) for two reasons. One is to provide programmers with error-checking of their assumptions about what objects conform to what types and what types are subtypes of what other types. Another is to allow programmers to encode additional semantic information in the use of a particular type in additional to the information implied by the type's purely syntactic interface. Both of these benefits are desirable as part of Cecil's goal of supporting production of high-quality software. To make exploratory programming easier, a programming environment tool could infer the greatest possible subtype relationships (i.e., the implicit "structural" subtyping relationships) for a particular object and add the appropriate explicit subtype declarations automatically.

Separating subtyping from implementation inheritance increases the complexity of Cecil. A simpler language might provide only subtyping, and restrict objects to inherit code only from their supertypes; Trellis takes this approach, for example. However, there is merit in clearly separating the two concepts, and allowing inheritance of code from objects which are not legal supertypes. Studies have found this to be fairly common in dynamically-typed languages [Cook 92]. With the current Cecil design, the only way that an object might not be a legal (structural) subtype of an object from which it inherits is if the child overrides a method of the parent and restricts at least one argument type declaration, a relatively rare occurrence. However, Cecil may eventually support filtering and transforming operations as part of inheritance, such as the ability to exclude operations, to rename operations, or to systematically adjust the argument types of operations, and so would create more situations in one object would inherit from another without being a subtype.

Types cannot have default implementations; only object representations can have methods attached. In other languages, such as Axiom (formerly Scratchpad II) [Watt *et al.*, Jenks & Sutor 92], default implementations can be stored with the type (called the *category* in Axiom). However, in Axiom method lookup rules are complicated by the possibility of methods being inherited both from superclasses and from categories, i.e., along both inheritance and subtyping links. Cecil's inheritance rules are simplified by only searching the inheritance graph. We expect that most type-like entities will actually be declared using the `object` form so that there is a corresponding representation to hold any default method implementations.

3.4 Special Types and Type Constructors

The syntax of types (excluding parameterization) is as follows:

```
type            ::= named_type
                  | closure_type
                  | lub_type
                  | glb_type
                  | "(" type ")"                just for grouping
```

3.4.1 Named Types

Types with names can be directly named:

```
named_type      ::= name
```

As described in section 3.3.1, type names are resolved in a name space distinct from the names of variables and objects and of methods.

In addition to user-defined types introduced through `type` and `object` declarations, the Cecil type system includes four special predefined types:

- The type `void` is used as the result type of methods and closures that do not return a result. All types are subtypes of `void`, enabling a method that returns a result to be used in a context where none is required. The type `void` may only be used when declaring the result type of a method or closure. The predefined object `void` has type `void`.

- The type `any` is implicitly a supertype of all types other than `void`; `any` may be used whenever a method does not require any special operations of an object.
- The type `none` is implicitly a subtype of all other types, thus defining the bottom of the type lattice. It is the result type of a closure that terminates with a non-local return, since such a closure never returns to its caller. It also is the result type of the primitive `loop` method, which also never returns normally. Finally, `none` is an appropriate argument type for closures that will never be called.
- The type `dynamic` is used to indicate run-time type checking. Wherever type declarations are omitted, `dynamic` is implied (with the exception of closure results and constant local variable declarations, as described in section 3.3.7). The `dynamic` type selectively disables static type checking, in support of exploratory programming, as described in section 3.10.

3.4.2 Closure Types

The type of a closure is described using the following syntax:

```
closure_type   ::= "&" "(" [arg_types] ")" [type_decl]
```

(The syntax of `arg_types` is specified along with signatures in section 3.3.4.)

A closure type of the form

$\&(t_1, \ldots, t_N):t_R$

describes a closure whose `eval` method has the signature:

signature eval$(\&(t_1,\ldots,t_N):t_R,\ t_1,\ \ldots,\ t_N):t_R$

Closure types are related by implicit subtyping rules that reflect standard contravariant subtyping: a closure type of the form $\&(t_1,\ldots,t_N):t_R$ is a subtype of a closure type of the form $\&(s_1,\ldots,s_N):s_R$ iff each t_i is a supertype of the corresponding s_i and t_R is a subtype of s_R.

3.4.3 Least-Upper-Bound Types

The least upper bound of two types in the type lattice is notated with the following syntax:

```
lub_type        ::= type "|" type
```

The type $type_1 \mid type_2$ is a supertype of both $type_1$ and $type_2$, and a subtype of all types that are supertypes of both $type_1$ and $type_2$. Least-upper-bound types are most useful in conjunction with parameterized types, described in section 4.

3.4.4 Greatest-Lower-Bound Types

The greatest lower bound of two types is notated with the following syntax:

```
glb_type        ::= type "&" type
```

The type $type_1$ & $type_2$ is a subtype of both $type_1$ and $type_2$, and a supertype of all types that are subtypes of both $type_1$ and $type_2$. Syntactically, the greatest-lower-bound type constructor has higher precedence than the least-upper-bound type constructor.

Note that the greatest-lower-bound of two types is different than a named type that is a subtype of the two types. For example,

```
type1 & type2
```

is a different type than the type introduced by the declaration

```
type type3 subtypes type1, type2;
```

The type `type3` is a subtype of `type1 & type2` (all types that subtype both `type1` and `type2` are automatically subtypes of `type1 & type2`), but not identical to it. The reason is that the programmer might later define a `type4` type:

```
type type4 subtypes type1, type2;
```

The type `type4` is also a subtype of `type1 & type2`, but `type3` and `type4` are different and in fact mutually incomparable under the subtype relation. The two types are different because named types include implicit behavioral specifications, and the implication of the two separate type declarations is that the implied behavioral specifications of `type3` and `type4` are different.

The `void`, `any`, and `none` special types and the greatest-lower-bound and least-upper-bound type constructors serve to extend the explicitly-declared type partial order generated from type and object declarations to a full lattice.

3.5 Object Role Annotations

Because Cecil is classless, objects are used both as run-time entities and as static, program structure entities. Some objects, such as `nil` and objects created at run-time through object constructor expressions, are manipulated at run-time and can appear as arguments to messages at run-time. Such *concrete* objects are required to have all the signatures in their types be supported by corresponding method implementations and all their fields be initialized. In contrast, objects such as `cons` and `list` are not directly manipulated at run-time. Instead, they help organize programs, providing repositories for shared methods and defining locations in the type lattice. In return for restricted usage, such *abstract* objects are not required to have their fields fully initialized nor their signatures fully implemented.

To inform the type checker about the part played by an object, its declaration is prefixed with an object representation role annotation:

`rep_role`	`::=`	`"abstract"`	*only inherited from by named objects; allowed to be incomplete*
	|	`"template"`	*only inherited from or instantiated; uninitialized fields allowed*
	|	`"concrete"`	*directly usable; must be complete and initialized*
	|	`["dynamic"]`	*directly usable; no static checks*

Each of these role annotations appears in the list hierarchy:

```
abstract object list isa collection;
template representation cons isa list;
concrete representation nil isa list;
```

Abstract objects are potentially incomplete objects designed to be inherited from and fleshed out by other objects. Abstract objects need not have all their signatures fully implemented nor their fields initialized. For example, the `list` object is not required to implement the `do` signature defined for the type `list`; the implementation of this operation is deferred to children. Because an abstract object may be incomplete, it cannot be used directly at run-time, nor can it appear as a parent in an object constructor declaration. Abstract objects are similar to abstract classes in class-based languages.

Template objects are complete objects suitable for direct "instantiation" by object constructor expressions, but are not allowed to be used directly as a value at run-time. Because new method implementations cannot be specified for anonymous objects, all the signatures specified as part of the type of a template object are required to be fully implemented. For example, the `cons` object is required to fully implement all `list` operations, including `do`. However, because template objects will not be sent messages at run-time, they are not required to have their fields initialized. The `cons` object is not required to have its `head` and `tail` fields initialized. Template objects are analogous to concrete classes in class-based languages.

Concrete objects are complete, initialized objects that can be manipulated at run-time. Like template objects, all signatures must be implemented, and in addition all fields must be initialized, either as part of the field declaration or as part of the object declaration or object constructor expression. Like other named objects, named concrete objects can be inherited from as well. (The child object's role can revert to abstract or template.) Anonymous concrete objects correspond to instances in class-based languages; named concrete objects have no direct analogue and are a feature of Cecil's object model.

If the object role annotation is `dynamic` or omitted, the object is considered fully manipulable by programs but no static checks for incomplete implementation of signatures or uninitialized fields are performed. (The appropriate checks will be made dynamically, as messages are sent and fields accessed.) Dynamic objects are designed to support exploratory programming, as discussed in section 3.10.

Since object constructor expressions create objects to be used at run-time, neither `abstract` nor `template` annotations are allowed on object constructor expressions.

Object role annotations help document explicitly the programmer's intended uses of objects. Other languages provide similar support. C++ indirectly supports differentiating abstract from concrete classes through the use of pure virtual functions and private constructors. Eiffel supports a similar mechanism through its deferred features and classes mechanism. Cecil's `abstract` annotation is somewhat more flexible than these approaches, since an object can be labeled `abstract` explicitly, even if it has no abstract methods. Such a declaration can be useful to prevent direct instantiation of the object, perhaps because the method implementations are mutually recursive in a way where subclasses are expected to override at least one of the methods to break the recursion.

In an earlier version of Cecil, a fifth annotation, `unique`, could be used to document the fact that an object was unique. For example, `nil`, `true`, and `false` all were annotated as unique objects. While the exact semantics of `unique` was unclear, a plausible interpretation could be that a unique object is like a concrete object except that it could not be used as a parent in an object constructor expression (i.e., it could not be "instantiated" or "copied"). Unique objects could still be inherited from in object declarations, since they might have useful code to be inherited. Unique objects were removed because it was felt that the extra language mechanism was not worthwhile. The `template` annotation may be removed for a similar reason, since it is not strictly necessary for the type checker, but the distinction between abstract and template objects appears to be useful for documenting the programmer's intentions. The distinction between abstract objects and concrete objects, however, is crucial to being able to write and type-check realistic Cecil code.

3.6 Type Checking Messages

This section describes Cecil's type checking rules for message sends and method declarations. Section 3.7 describes type checking for other, simpler kinds of expressions. Parameterized types are described in section 4.

In Cecil, all control structures, instance variable accesses, and basic operators are implemented via message passing, so messages are the primary kind of expression to type-check. For a message to be type-correct, there must be a single most-specific applicable method implementation defined for all possible argument objects that might be used as an argument to the message. However, instead of directly checking each message occurring in the program against the methods in the program, in Cecil we check messages against the set of signatures defined for the argument types of the message, and then check that each signature in the program is implemented conformingly, completely, and consistently by some group of methods.

Using signatures as an intermediary for type checking has three important advantages. First, the type-checking problem is simplified by dividing it into two separable pieces. Second, checking signatures enables all interfaces to be checked for conformance, completeness, and consistency independent of whether messages exist in the program to exercise all possible argument types. Finally, signatures enable the type checker to assign blame for a mismatch between implementor and client. If some message is not implemented completely, the error is either "message not understood" or "message not implemented correctly." If the signature is absent, it is the former, otherwise the latter. Signatures inform the type checker (and the programmer) of the intended interfaces of abstractions, so that the system may report more informative error messages. Of course, the "missing signature" error is sometimes the appropriate message to report, but the type checker cannot accurately distinguish this from the "message not understood" alternative.

Subsection 3.6.1 describes checking messages against signatures, and subsection 3.6.2 describes checking signatures against method implementations.

3.6.1 Checking Messages Against Signatures

Given a message of the form *name* $(expr_1, \ldots, expr_N)$, where each $expr_i$ type-checks and has static type T_i, the type checker uses the T_i to locate all signatures of the form *name* $(S_1, \ldots, S_N)$ $: S_R$ where each type S_i is a supertype of the corresponding T_i. If this set of applicable signatures is empty, the checker reports a "message not understood" error. Otherwise, the message send is considered type-correct.

To determine the type of the result of the message send, the type system calculates the most-specific result type of any applicable signature. This most-specific result type is computed as the greatest lower bound of the result types of all applicable signatures. In the absence of other type errors, this greatest lower bound will normally correspond to the result type of the most-specific signature.

To illustrate, consider the message `copy(some_list)`, where the static type of `some_list` is `list`. The following types and signatures are assumed to exist:

```
type collection;
type list subtypes collection;
type array subtypes collection;

signature copy(collection):collection;
signature copy(list):list;
signature copy(array):array;
```

The signature `copy(array):array` is not applicable, since `list`, the static type of `some_list`, is not a subtype of `array`. The dynamic type of `some_list` might turn out to conform to `array` at run-time (e.g., if there were some data structure that was both a `list` and an `array`), but the static checker cannot assume this and so must ignore that signature. The first two signatures do apply, so the `copy` message is considered legal. The type of the result is known to be both a `list` and a `collection`. The greatest lower bound of these two is `list`, so the result of the `copy` message is of type `list`.

Unlike method dispatching, it is acceptable for more than one signature to be applicable to a message. Signatures are contracts that clients can assume, and if more than one signature is applicable, then the client can assume more guarantees about the type of the result. The greatest lower bound is used to calculate the message's result type, rather than the least upper bound, because each signature can be assumed to be in force. At run-time, a method will be selected, and that method will be required to honor the result type guarantees of all the applicable signatures, and so the target method implementation will return an object that conforms to the result types of all the applicable signatures, i.e., the greatest lower bound of these signatures. In common practice, some most-specific signature's result type will be the greatest lower bound, such as the `list` type selected above.

3.6.2 Checking Signatures Against Method Implementations

The type checker ensures that, for every signature in the program, all possible messages that could be declared type-safe by the signature would in fact locate a single most-specific method with appropriate argument and result type declarations, given the current set of representation and type declarations in the program. This involves locating all methods to which the signature is applicable (i.e., all those that could be invoked by a message covered by the signature) and ensuring that they conformingly, completely, and consistently implement the signature.

More precisely:

- A signature is considered *applicable* to a method iff they have the same name and number of arguments and there exists some sequence of argument objects that both inherits from the specializers of the method and conforms to the argument types of the signature. Abstract objects are not included when considering possible argument objects, since they are not required to be complete implementations and are restricted from being manipulated at run-time. (This is the key distinction between abstract and non-abstract objects.) Template objects are included, since they are required to fully implement all applicable signatures.

- A method *conforms* to a signature iff
  - for each formal, all objects that inherit from the formal's specializer and conform to the signature's corresponding argument type also conform to the formal's declared type (for unspecialized formals, this constraint amounts to requiring that the formal's type is a supertype of the signature argument's type), and
  - the method's result type is a subtype of the signature's result type.
- A set of methods *completely* implement a signature iff, for each possible sequence of argument objects that conforms to the argument types in the signature, there exists at least one method in the set that is applicable to the argument objects, i.e., where the argument objects inherit from the method's specializers.
- A set of methods *consistently* implement a signature iff, for each possible sequence of argument objects that conforms to the argument types in the signature, there exists a single most-specific applicable method in the set.

Conformance of a method against a signature can be checked in isolation of any other methods and signatures in the program. However, in the presence of multi-methods, it is not possible to check individual methods in isolation for completeness and consistency, since interactions among multi-methods can introduce ambiguities where none would exist if the multi-methods were not jointly defined within one program.

To type-check in the presence of Cecil's prototype-based object model, *object representatives* are extracted from the program. Each named template, concrete, and dynamic object is considered a distinct object representative, and each static occurrence of an object constructor expression is considered an object representative. A finite number of representatives are extracted from any given program. Representatives are then used as the potential run-time argument objects when testing whether a signature is applicable to a method and whether a set of methods completely and consistently implement a signature. The object representative for an object constructor expression acts as a proxy for all the objects created at run-time by executing that object constructor expression. Since each object created by a particular object constructor expression inherits the same set of methods and has the same type, only one representative need be checked to ensure type safety of all objects created by the object constructor expression at run-time. Object representatives are analogous to concrete classes in a class-based language and maps in the Self implementation [Chambers *et al.* 89].

Conceptually, for each signature, the type checker enumerates all possible *message representatives* that are covered by the signature, where the arguments to the message representative are object representatives that conform to the signature's argument types. (A much more efficient algorithm to perform this checking is described elsewhere [Chambers & Leavens 94].) For each message representative, the type checker simulates method lookup and checks that the simulated message would locate exactly one most-specific method. If no method is found, the type checker reports a "signature implemented incompletely" error. If multiple mutually ambiguous methods are found, the type checker reports a "signature implemented inconsistently" error. Otherwise, the single most-specific method has been found for the message representative. In this case, the type checker also verifies that the argument object representatives conform to the declared argument types of the target method and that the declared result type of the method is a subtype of the signature's result type. If all these tests succeed, then all run-time messages matching the message representative are guaranteed to execute successfully.

For example, consider type-checking the implementation of the following signature:

```
signature pair_do(collection, collection, &(int,int):void):void;
```

The type checker would first collect all object representatives that conform to `collection` and all those that conform to `&(int,int):void`. For a small system, the `collection`-conforming object representatives might be the following:

```
representation nil inherits list;
representation cons inherits list;
representation inherits cons;
representation array inherits collection;
```

The `list` and `collection` objects are not enumerated because they are `abstract`. The third representative is extracted from the object constructor expression in the `prepend` method. A single object representative stands for the closure object.

Once the applicable object representatives are collected, the type checker enumerates all possible combinations of object representatives conforming to the argument types in the signature to construct message representatives. These message representatives are the following:

```
pair_do(nil,nil,closure);
pair_do(nil,cons,closure);
pair_do(nil,representation inherits cons,closure);
pair_do(nil,array,closure);
pair_do(cons,nil,closure);
pair_do(cons,cons,closure);
...
pair_do(array,representation inherits cons,closure);
pair_do(array,array,closure);
```

For each message representative, method lookup is simulated to verify that the message is understood, that the declared argument types are respected, and that the target method returns a subtype of the signature's type.

3.6.3 Comparison with Other Type Systems

For singly-dispatched languages, most type systems apply contravariant rules to argument and result types when checking that the overriding method can safely be invoked in place of the overridden method: argument types in the overriding method must be supertypes of the corresponding argument types of the overridden method, while the result type must be a subtype. Cecil's type system does not directly compare one method against another to enforce contravariant redefinition rules, but instead compares one method against an applicable signature to enforce contravariant rules for non-specialized arguments. In Cecil terms, in a singly-dispatched language a signature is inferred from the superclass's method, and then all subclass methods (i.e., those methods that are applicable to the signature) are checked for conformance to the signature.

Specialized arguments need not obey contravariant restrictions. The type of a specialized argument for one method can be a subtype of the type of the corresponding argument for a more general method. This does not violate type safety because run-time dispatching will guarantee that the method will only be invoked for arguments that inherit from the argument specializer, and the static type checker has verified that all objects that inherit from the specializer also conform to the specialized argument's type. Unspecialized arguments cannot safely be covariantly redefined, because there is no run-time dispatching on such arguments ensuring that the method will only be invoked when the type declaration is correct.

Singly-dispatched languages make the same distinction between specialized and unspecialized arguments implicitly in the way they treat the type of the receiver. For most singly-dispatched languages, the receiver argument is omitted from the signatures being compared, leaving only unspecialized arguments and hence the contravariant redefinition rule. If the receiver type were included as an explicit first argument, it would be given special treatment and allowed to differ covariantly. (In fact, it must, since the receiver's type determines when one method overrides another!) For Cecil, any of the arguments can be specialized or unspecialized, requiring us to make the distinction explicit. If all methods in a Cecil program specialized on their first argument only, Cecil's type checking rules would reduce to those found in a traditional singly-dispatched language.

Few multiply-dispatched languages support static type systems. Two that are most relevant are Polyglot [Agrawal *et al.* 91] and Kea [Mugridge *et al.* 91]. In both of these systems, type checking of method consistency and completeness requires that all "related" methods (all methods in the same generic function in Polyglot and all variants of a function in Kea) be available to the type checker, just as does Cecil. Neither Polyglot nor Kea distinguishes subtyping from inheritance nor interfaces from implementations. Additionally, neither Polyglot nor Kea supports a notion of abstract classes that are not required to be completely implemented but that include some notion of an operation that is expected to be implemented by subclasses; signatures play this role in Cecil.

3.6.4 Type Checking Inherited Methods

Cecil does not require that a method be re-type-checked when inherited by a descendant, even if that descendant is not a subtype. This feat is accomplished by verifying that all descendant objects conform to the declared type of the corresponding formal of the inherited method. If the declared type is the type of the specializer, such as would arise with a type declaration using the `@:` syntax, then all descendant objects are required to be subtypes of the specializer as well. This may be constraining. For example, consider the following `set` and `bag` implementation fragments:

```
template object bag isa unordered_collection;
   field elems(@:bag):list;
   method add(b@:bag, x:int):void {
      b.elems := cons(b.elems, x); }
   method includes(b@:bag, x:int):bool {
      b.elems.includes(x) }
   ...
template object set isa unordered_collection inherits bag;
   method add(s@:set, x:int):void {
      if_not(includes(s, x), { resend(s, x) }) }
```

Here the type checker would report an error, since set inherits from bag but is not a subtype, violating the conformance requirements for bag's elems, add, and includes methods.* In this case, a new type bag_like_object could be created that understood the elems and set_elems messages and the b formal of the bag add and includes methods should be changed to be of this type:

```
abstract object bag_like_object;
   field elems(@:bag_like_object):list;

template object bag isa unordered_collection, bag_like_object;
   method add(b@bag:bag_like_object, x:int):void {
      b.elems := cons(b.elems, x); }
   method includes(b@bag:bag_like_object, x:int):bool {
      b.elems.includes(x) }
   ...

template object set isa unordered_collection, bag_like_object inherits bag;
   method add(s@set:bag_like_object, x:int):void {
      if_not(includes(s, x), { resend(s, x) }) }
```

The programmer could go further and move many of the bag operations into the bag_like_object. Eventually, set would simply inherit from bag_like_object, not bag. In this situation, all inheritance links would parallel subtyping links, and the two would not need to be distinguished.

If such reorganizations can always be made satisfactorily, with the resulting inheritance and subtyping graphs parallel, then it may not be necessary to separate inheritance from subtyping in the language. However, such an approach may not always be feasible. Creating the intermediate bag_like_object is somewhat tedious; the original code was easy to read and dynamically type-safe. Moreover, the implementation of bag might be written independently and not under control of the programmer building set. In these cases, simply reusing the implementation of bag for set is convenient. Unfortunately, Cecil's type rules currently seem to prevent the simple solution. One alternative would simply be to re-type-check a method whenever it was inherited by an object that was not also a subtype. The @: notation could be interpreted as indicating that this sort of re-type-checking was to be done. Re-type-checking would require access to at least part of the inherited method's source code, however. Another alternative would be to relax the conformance constraint for any object that inherited an overriding method. In this example, the bag add method would not need to be rewritten, since the set add method "shadows" it for the only descendant object that is not also a subtype; the includes method would still need to be rewritten. Also, the resend in the set add method would become type-incorrect, since it is passing an argument of type set to a method expecting an argument of type bag. This alternative is close to the idea of encapsulating the use of inheritance from clients, as with private inheritance in C++. We consider the separation of subtyping from inheritance, when coupled with the desire to avoid retypechecking methods when inherited, to be an important area for future work.

* Sets are not subtypes of bags since sets do not support the behavioral specification of bags. A client could detect the difference between a set and a bag by adding the same element twice to an unordered collection and testing how much the size of the collection changed.

3.7 Type Checking Expressions, Statements, and Declarations

Type checking an expression or statement determines whether it is type-correct, and if type-correct also determines the type of its result. Type checking a declaration simply checks for type correctness. All constructs are type-checked in a typing context containing the following information:

- a binding for each variable or object name in scope to either:
  - the variable's declared type and an indication of whether the variable binding is assignable or constant, or
  - to an object with a role annotation and a set of conformed-to types.
- the set of inheritance relations currently in scope;
- a binding for each type name in scope to the corresponding type;
- the set of subtyping relations currently in scope;
- the set of signatures currently in scope (for type checking messages);
- the set of method declarations currently in scope (for type checking resends).

The type checking rules for expressions are as follows:

- A literal constant is always type-correct. The type of the result of a literal constant is the corresponding predefined type.
- A reference *name* is type-correct iff *name* is defined in the typing context (i.e., if there exists a declaration of that name earlier in the same scope or in a lexically-enclosing scope) as either a variable or an object. If a variable, then the reference is type-correct, with the type of the result being the associated type of the variable in the typing context. If an object, then the reference is type-correct iff the object is a concrete or dynamic object, with the type of the result being the type of the named object.
- An object constructor expression of the general form

  role-annotation **`object inherits`** $parent_1$`, ..., `$parent_K$
  `subtypes` $supertype_1$`, ..., `$supertype_L$
  `isa` $parent\text{-}and\text{-}supertype_1$`, ..., `$parent\text{-}and\text{-}supertype_M$
  `{` $field_1$`@`obj_1 `:=` $expr_1$`, ..., `$field_N$`@`obj_N `:=` $expr_N$ `}`

  is type-correct iff:

  - each $parent_i$ name is bound to a non-abstract non-`void` object in the typing context;
  - each $supertype_i$ notates a type other than `none` in the current typing context;
  - each $parent\text{-}and\text{-}supertype_j$ name is bound to a non-abstract non-`void` object in the typing context;
  - if `@`obj_i is present, then obj_i names an ancestor of the newly created object (if absent, it is considered to be the same as the newly created object);
  - each $field_i$ names a field F_i specialized on or inherited unambiguously by obj_i, ignoring any overriding methods, and F_i is not `shared`;
  - each $expr_i$ is type-correct, returning an object of static type T_i, and T_i is a subtype of the type of the contents of the field F_i;
  - no field F_i is initialized more than once;

- *role-annotation* is neither `abstract` nor `template`; and
- if *role-annotation* is `concrete`, then there do not exist any fields specialized on or inherited by the newly created object that do not have a default initial value and are not initialized as part of the object creation expression.

The `representation` keyword may be used in place of the `object` keyword without effect. The type of the result of an object constructor expression is a new anonymous type that is a subtype of each of the *supertype*$_i$ types and each of the types of the *parent-and-supertype*$_j$ objects.

- A closure constructor expression of the general form

  `&(`x_1`:`*type*$_1$`, …, `x_N`:`*type*$_N$`):`*type*$_R$ `{` *body* `}`

  is type-correct iff:

  - the x_i, where provided, are distinct;
  - each of the *type*$_i$, if provided, notates a non-`void` type in the current typing context;
  - *type*$_R$, if provided, notates a type in the current typing context;
  - *body* is type-correct, checked in a typing context constructed by extending the current typing context with constant variable bindings for each of the x_i to the corresponding type *type*$_i$; and
  - the type of the result of *body* is a subtype of *type*$_R$, if provided; if `:`*type*$_R$ is omitted, then *type*$_R$ is inferred to be the type of the result of *body*.

  The type of the result of a closure constructor expression of the above form is

  `&(`*type*$_1$`, …, `*type*$_N$`):`*type*$_R$`.`

- A vector constructor expression of the general form `[`*expr*$_1$`,…,`*expr*$_N$`]` type-correct iff each of the *expr*$_i$ is type-correct, with static type T_i which is not `void`. The type of the result of a vector constructor expression is the predefined parameterized type `i_vector` instantiated with the least upper bound of the T_i. (See section 4 for information on parameterized types.)
- A message expression of the general form *name*(*expr*$_1$`, …, `*expr*$_N$) is type-correct iff:

  - each of the *expr*$_i$ is type-correct, with static type T_i which is not `void`;* and
  - the set $S = \{S_1, ..., S_M\}$ of applicable signatures is non-empty, where S is the set of signatures in the current typing context of the form S_i = **`signature`** *name*`(`t_{i1}`, …, `t_{iN}`):`t_{iR} where each T_i is a subtype of t_i.

  The type of the result of a message is the greatest lower bound of each of the result types t_{iR} of the applicable signatures. Verifying correctness of the implementation of signatures is described in subsection 3.6.2.

- A resend expression of the general form

  `resend``(…, `x_i`@`*parent*$_i$`, …, `*expr*$_j$`, …)`

  is type-correct iff:

  - each of the arguments x_i or *expr*$_i$ is type-correct, with static type T_i which is not `void`;
  - the resend is nested textually in the body of a method M;
  - M takes the same number of arguments, N, as does the resend;

* The check that the argument type is not `void` is not strictly necessary, since no signature will have an argument type that is a supertype of `void`.

- for each specialized formal parameter *formal*$_i$ of M, specialized on *object*$_i$, the i^{th} argument to the resend is *formal*$_i$, possibly suffixed with `@`*parent*$_i$, and *formal*$_i$ is not shadowed with a local variable of the same name;
- for each unspecialized formal parameter *formal*$_j$ of M, the j^{th} argument to the resend is not be suffixed with `@`*parent*$_j$;
- for each resend argument of the form *formal*$_i$`@`*parent*$_i$, *parent*$_i$ is a proper ancestor of *object*$_i$, the specializer of *formal*$_i$, other than `void`; and
- when method lookup is simulated with a message name the same as M and with N arguments, where argument i is either `any` (if *formal*$_i$ of M is unspecialized), *parent*$_i$ (if the argument of the resend is directed using the `@`*parent*$_i$ suffix notation), or *object*$_i$, the specializer of *formal*$_i$ (otherwise), and where the resending method M is removed from the set of applicable methods in the current typing context, exactly one most-specific target method R is located, and the argument type declarations of this target method S_i are supertypes of the corresponding T_i.

The type of the result of a resend expression is the declared result type of the target method R.

- A parenthetical expression of the form `(` *body* `)` is type-correct iff *body* is type-correct. The type of the result of a parenthetical expression is the type of the result of *body*.

The following rules define type-correctness of statements:

- An assignment statement of the form *name* `:=` *expr* is type-correct iff:
  - *expr* is type-correct, with static type T_{expr};
  - *name* is bound to an assignable variable of type T_{name} in the current typing context; and
  - T_{expr} is a subtype of T_{name}.

  The type of the result of an assignment statement is `void`.
- A declaration block is type-correct iff its declarations are type-correct, when checked in a typing context where all names in the declaration block are available throughout the declaration block. The type of the result of a declaration block is `void`.
- An expression statement is type-correct iff the expression is type-correct, with static type T. The type of the result of the expression statement is T.
- A non-local return statement, of the form `^` *expr*, is type-correct iff:
  - *expr* is type-correct, with static type T;
  - the non-local return statement is nested textually inside the body of a method M; and
  - T is a subtype of the declared result type of M.

  The type of the (local) result of a non-local return is `none`.

The body of a method, closure, or parenthetical expression is type-correct iff its statements are type-correct. The type of the result of a body is the type of its last statement, if present, or `void`, otherwise.

The following rules define type-correctness of declarations:

- A variable declaration of the form **`let`** *var name*`:`*type* `:=` *expr*, where *var* is either `var` or empty, is type-correct iff:
  - *name* is not otherwise defined in the same scope;

- *type* notates a type in the current typing context; and
- *expr* is type-correct in a typing context where *name* and all variables defined later in the same declaration block are unbound, resulting in static type *T*, and *T* is a subtype of *type*.

The typing context is extended to include a variable binding for *name* to the type *type* that is assignable if *var* is `var` and constant otherwise.

- A type declaration of the form

  `type` *name* **`subtypes`** $supertype_1$, ..., $supertype_N$

  is type-correct iff each of the $supertype_i$ notates a type other than `none` in the current typing context and no cycles are introduced into the subtyping graph as a result of the declaration. As a result of the declaration, the typing context is extended to include a type binding from *name* to a new type that is a subtype of each of the $supertype_i$ types.

- A representation declaration of the form

  ```
  role-annotation kind name inherits parent_1, ..., parent_K
                            subtypes supertype_1, ..., supertype_L
                            isa parent-and-supertype_1, ..., parent-and-supertype_M
                            { field_1@obj_1 := expr_1, ..., field_N@obj_N := expr_N }
  ```

  is type-correct under the same conditions as the analogous object constructor expression, with the following changes:

  - abstract objects may be named in `inherits` and `isa` clauses;
  - the `abstract` and `template` role annotations are allowed; and
  - no cycles are allowed to be introduced into the inheritance and subtyping graphs.

  The typing context is extended to include an object binding from *name* to a new object with role *role-annotation* that inherits from the $parent_i$ objects and the $parent\text{-}and\text{-}supertype_j$ objects. If *kind* is the `representation` keyword, then the new object conforms to the $supertype_k$ types. Otherwise, *kind* is the keyword `object`, and the typing context is also extended with a type binding from *name* to a new type that is a subtype of each of the $supertype_i$ types, and the new object conforms to the new type.

- A type extension declaration of the form

  `extend type` *name* **`subtypes`** $supertype_1$, ..., $supertype_N$

  is type-correct iff:

  - *name* is bound in the typing context to a type other than `void`, `any`, `none`, and `dynamic`; and
  - the same constraints on the `subtypes` clause as with the type declaration are satisfied.

  As a result of the declaration, the typing context is extended to reflect that the type *name* is a subtype of each of the $supertype_i$ types.

- A representation extension declaration of the form

  ```
  extend kind name inherits parent_1, ..., parent_K
                   subtypes supertype_1, ..., supertype_L
                   isa parent-and-supertype_1, ..., parent-and-supertype_M
                   { field_1@obj_1 := expr_1, ..., field_N@obj_N := expr_N }
  ```

  is type-correct iff:

  - *name* is bound in the typing context to an object other than `void` and `any`;

- if *kind* is `object` or omitted, then *name* also is bound in the typing context to a type other than `void`, `any`, `none`, and `dynamic`;
- the same constraints on the `inherits`, `subtypes`, `isa`, and field initialization clauses as with the object representation declaration are satisfied; and
- none of the $field_i$`@`obj_i initialize fields already specialized on or inherited by the object before the extension.

As a result of the declaration, the typing context is extended to reflect that the object *name* inherits from the $parent_i$ objects and the $parent\text{-}and\text{-}supertype_j$ objects. If *kind* is the `representation` keyword, then the typing context is extended to reflect that the object conforms to the $supertype_k$ types. Otherwise, *kind* is the keyword `object`, and the typing context is extended to reflect that the *name* type is a subtype of each of the $supertype_i$ types and that the *name* object conforms to the *name* type.

- A signature declaration of the form

  `signature` *name* `(`x_1`:`$type_1$`,`...`,`x_N`:`$type_N$`)` `:`$type_R$

  is type-correct iff:

  - the x_i, when provided, are distinct;
  - each of the $type_i$ notates a type other than `void` in the typing context; and
  - $type_R$ notates a type in the typing context.

  The typing context is extended to include the corresponding signature.

- A field signature declaration of the form

  var **`field signature`** *name* `(`x`:`$type$`)` `:`$type_R$

  is type-correct iff:

  - *type* notates a type other than `void` in the typing context; and
  - $type_R$ notates a type other than `void` in the typing context.

  The typing context is extended to include the signature

  `signature` *name* `(`$type$`)` `:`$type_R$

  and, if *var* is `var`, the signature

  `signature` `set_`*name* `(`$type$`,`$type_R$`)` `:void`

- A method implementation declaration of the general form

  `method` *kind* *name* `(`x_1`@`obj_1`:`$type_1$`,`...`,`x_N`@`obj_N`:`$type_N$`)` `:`$type_R$ `{` *body* `}`

  is type-correct iff:

  - the x_i, when provided, are distinct;
  - each of the $type_i$ notates a type other than `void` in the typing context;
  - if `@`obj_i is present, then obj_i conforms to $type_i$;
  - $type_R$ notates a type in the typing context;
  - *body* is type-correct when checked in a typing context constructed by extending the current typing context with constant variable bindings for each of the x_i to the corresponding type $type_i$; and
  - the type of the result of *body* is a subtype of $type_R$.

  The typing context is extended to include the declared method implementation. If *kind* is not `implementation`, then the typing context is also extended to include the signature

`signature` *name* `(`$type_1$`,`...`,`$type_N$`)` `:`$type_R$

- A field implementation declaration of the general form

  shared var **`field`** *kind name* `(`*x*`@`*obj*`:`*type*`)` `:`$type_R$ `:=` *expr*`;`

  is type-correct iff:

  - *type* notates a type other than `void` in the typing context;
  - if `@`*obj* is present, then *obj* conforms to *type*;
  - $type_R$ notates a type other than `void` in the typing context;
  - if `:=` *expr* is provided, then *expr* is type-correct, with static type *T*, and *T* is a subtype of $type_R$; and
  - if *shared* is `shared`, then `:=` *expr* is provided.

  The typing context is extended to include the declared field get accessor method implementation, plus the set accessor method implementation if *var* is `var`, plus the get (and possibly set) signature(s) if *kind* is not `implementation`.

3.8 Type Checking Subtyping Declarations

When the programmer declares that an object conforms to a type (via a `subtypes` or `isa` clause), the type system trusts this declaration and uses it when checking conformance and subtyping. However, it is possible that the programmer's claim is wrong, and that the object in fact does not faithfully implement the interface of the types to which it supposedly conforms. In this case, the signature implementation checking, described in section 3.6.2, is sufficient to detect and report the error, so no additional checking is required. When enumerating and checking message representatives matching a signature defined on the supertype, the object in question, if not abstract, will be enumerated, and the error will be detected because some signature will not be implemented properly for that object. If the object is abstract, no type error will be reported. This will not affect running programs since the abstract object cannot be used in a message. Also, since abstract objects are allowed to be incomplete, it is unclear whether a type error really exists.

3.9 Type Checking Predicate Objects

Predicate objects are intended to represent alternative ways of implementing an object's interface. Accordingly, it should be possible to type-check programs using predicate objects, under the assumption that the particular state of the object does not affect its external interface. In particular, to guarantee type safety in the presence of predicate objects, the type checker must verify that for each message declared in the interface of some object *O*:

- at all times there is an implementation of the message inherited by the object *O*; and
- at no time are there several mutually ambiguous implementations of the message inherited by the object *O*.

These two tests correspond to extending the tests of completeness and consistency of method implementations to cope with the presence of predicate objects.

The set of methods inherited by the object *O* from normal objects is fixed at program-definition time and can be type-checked in the standard way. Methods inherited from predicate objects pose more of a problem. If two predicate objects might be inherited simultaneously by an object, either one predicate object must be known to override the other or they must have disjoint method names. For example, in the bounded buffer implementation described in section 2.4, since an object can inherit from both the `non_empty_buffer` and the `non_full_buffer` predicate objects, the two predicate objects should not implement methods with the same name. Similarly, if the only implementations of some message are in some set of predicate objects, then one of the predicate objects must always be inherited for the message to be guaranteed to be understood. In other words, the checker needs to know when one predicate object *implies* another, when two predicate objects are *mutually exclusive*, and when a group of predicate objects is *exhaustive*. Once these relationships among predicate objects are determined, the rest of type-checking becomes straightforward.

Ideally, the system would be able to determine all these relationships automatically by examining the predicate expressions attached to the various predicate objects. However, predicate expressions in Cecil can run arbitrary user-defined code, and consequently the system would have a hard time automatically inferring implication, mutual exclusion, and exhaustiveness. Consequently, we rely on explicit user declarations to determine the relationships among predicate objects; the system can verify dynamically that these declarations are correct.

A declaration already exists to describe when one predicate object implies another: the `isa` declaration. If one predicate object explicitly inherits from another, then the first object's predicate is assumed to imply the second object's predicate. Any methods in the child predicate object override those in the ancestor, resolving any ambiguities between them.

Mutual exclusion and exhaustiveness are specified using declarations of the following form:

```
disjoint_decl  ::= "disjoint" named_objects ";"
cover_decl     ::= "cover" named_object "by" named_objects ";"
divide_decl    ::= "divide" named_object "into" named_objects ";"
named_objects  ::= named_object { "," named_object }
```

The disjoint declaration

disjoint *object*$_1$, ..., *object*$_n$;

implies to the static type checker that the predicate objects named by each of the *object*$_i$ will never be inherited simultaneously, i.e., that at most one of their predicate expressions will evaluate to true at any given time. Mutual exclusion of two predicate objects implies that the type checker should not be concerned if both predicate objects define methods with the same name, since they cannot both be inherited by an object. To illustrate, the following declarations extend the earlier bounded buffer example with mutual exclusion information:

```
disjoint empty_buffer, non_empty_buffer;
disjoint full_buffer, non_full_buffer;
```

The system can infer that `empty_buffer` and `full_buffer` are mutually exclusive with `partially_full_buffer`. Note that `empty_buffer` and `full_buffer` are not necessarily mutually exclusive.

The cover declaration

cover *object* **by** *object*$_1$, ..., *object*$_n$;

implies that whenever an object *O* descends from *object*, the object *O* will also descend from at least one of the $object_i$ predicate objects; each of the $object_i$ are expected to descend from *object* already. Exhaustiveness implies that if all of the $object_i$ implement some message, then any object inheriting from *object* will understand the message. For example, the following coverage declaration extends the bounded buffer predicate objects:

```
cover buffer by empty_buffer, partially_full_buffer, full_buffer;
```

Often a group of predicate objects divide an abstraction into a set of exhaustive, mutually-exclusive subcases. The divide syntactic sugar makes specifying such situations easier. A declaration of the form

divide *object* **into** $object_1$, ..., $object_n$;

is syntactic sugar for the following two declarations:

disjoint $object_1$, ..., $object_n$;
cover *object* **by** $object_1$, ..., $object_n$;

Since fields are accessed solely through accessor methods, checking accesses to fields in predicate objects reduces to checking legality of messages in the presence of predicate objects, as described above. To ensure that fields are always initialized before being accessed, the type checker simply checks that the values of all fields potentially inherited by an object are initialized either at the declaration of the field or at the creation of the object.

3.10 Mixed Statically- and Dynamically-Typed Code

One of Cecil's major design goals is to support both exploratory programming and production programming and in particular to support the gradual evolution from programs written in an exploratory style to programs written in a production programming style. Both styles benefit from object-oriented programming, a pure object model, user-defined control structures using closures, and a flexible, interactive development environment. The primary distinction between the two programming styles relates to how much effort programmers want to put into polishing their systems. Programmers in the exploratory style want the system to allow them to experiment with partially-implemented and partially-conceived systems, with a minimum of work to construct and subsequently revamp systems; rapid feedback on incomplete and potentially inconsistent designs is crucial. The production programmer, on the other hand, is concerned with building reliable, high-quality systems, and wants as much help from the system as possible in checking and polishing systems.

To partially support these two programming styles within the same language, type declarations and type checking are optional. Type declarations may be omitted for any argument, result, or local variable. Programs without explicit type declarations are smaller and less redundant, maximizing the exploratory programmer's ability to rapidly construct and modify programs. Later, as a program (or part of a program) matures, the programmer may add type declarations incrementally to evolve the system into a more polished and reliable production form.

Omitted type declarations are treated as `dynamic`; `dynamic` may also be specified explicitly as the type of some argument, result, or variable. An expression of type `dynamic` may legally be passed as an argument, returned as a result, or assigned to a variable of any type. Similarly, an expression of any type may be assigned to, passed to, or returned from a variable, argument, or result, respectively, of type `dynamic`. This approach to integrating dynamically-typed code with statically-typed code has the effect of checking type safety statically wherever two statically-typed expressions interact (assuming that at run-time the objects resulting from evaluating the statically-typed expressions actually conform to the given types), and deferring to run-time checking at message sends whenever a dynamically-typed expression is used.

A consequence of this semantics for the `dynamic` type is that the static type safety of statically-typed expressions can be broken by passing an incorrect dynamically-typed value to a statically-typed piece of the program. Dynamic type checking will catch errors eventually, but run-time type errors can occur inside statically-typed code even if the code passes the type checker. An alternative approach would check types dynamically at the "interface" between dynamically- and statically-typed code: whenever a dynamically-typed value is assigned to (or passed to, or returned as) a statically-typed variable or result, the system could perform a run-time type check of the dynamically-typed value as part of the assignment. This approach would then ensure the integrity of statically-typed code: no run-time type errors can occur within statically-typed code labeled type-correct by the typechecker, even when mixed with buggy dynamically-typed code. Unfortunately, this approach has some difficulties. One problem is that objects defined in exploratory mode should not be required to include explicit subtyping declarations; such declarations could hinder the free-flowing nature of exploratory programming. However, if such an object were passed to statically-typed code, the run-time type check at the interface would fail, since the object had not been declared to be a subtype of the expected static type. We have chosen for the moment to skip the run-time check at the interface to statically-typed code in order to support use of statically-typed code from exploratory code, relying on dynamic checking at each message send to ensure that the dynamically-typed object supports all required operations. An alternative might be to perform some form of inference of the subtyping relationships of dynamically-typed objects, like that incorporated in object-oriented systems based on implicit structural subtyping, and use these inferred subtyping relationships for the run-time type check.

Cecil supports the view that static type checking is a useful tool for programmers willing to add extra annotations to their programs, but that all static efficiently-decidable checking techniques are ultimately limited in power, and programmers should not be constrained by the inherent limitations of static type checking. The Cecil type system has been designed to be flexible and expressive (in particular by supporting multi-methods, separating the subtype and code inheritance graphs, and supporting explicit and implicit parameterization) so that many reasonable programs will successfully type-check statically, but we recognize that there may still be reasonable programs that either will be awkward to write in a statically-checkable way or will be difficult if not impossible to statically type-check in any form. Accordingly, error reports do not prevent the user from executing the suspect code; users are free to ignore any type checking errors reported by the system, relying instead of dynamic type checks. Static type checking is a useful tool, not a complete solution.

4 Parameterization and Bounded Parametric Polymorphism

Practical statically-typed languages need bounded parametric polymorphism. Without some mechanism for *type parameterization*, programmers must either resort to multiple similar implementations of the same abstraction that differ only in type annotations, or insert type casts, often at the client side, to indicate the more precise types of expressions than the type checker infers. For example, if parameterization is not available, several nearly identical implementations of `list` or `array` may be needed for lists or arrays of integers, strings, etc., and control structures such as `if` and `map` could not be reused for a variety of argument types. Accordingly, Cecil supports the definition of parameterized object representations, method and field implementations, types, signatures, and subtype and inheritance relations. The programmer is allowed to express the assumptions on the type parameters in such declarations using mixed subtype and signature *type constraints*. For example, a type parameter may be restricted to be a subtype of a certain type or to be any type such that a certain signature holds. Type constraints in Cecil generalize F-bounded polymorphism [Canning et al. 89] and Theta-style `where` clauses [Day et al. 95, Liskov et al. 94].

This section presents type parameterization and type constraints in Cecil. A more formal development, although in a simpler setting and using a slightly different notation, appears elsewhere [Litvinov 98]. The next subsection introduces parameterization. Subsection 4.2 adds constraints to achieve bounded polymorphism. Subsection 4.3 describes constraint solving and type inference. Subsection 4.4 describes an advanced use of the type system to express F-bounded polymorphism. The last subsection reviews related work.

4.1 Parameterized Declarations

4.1.1 Type Parameters, Type Variables, and Instantiating Types

Cecil supports parametric polymorphism by allowing declarations to be parameterized with *type variables*. (This facility is not provided for variable declarations, for which it would be unsound.) A `forall T1`, ... ,`Tn` prefix introduces type variables `T1`, ... ,`Tn` in a declaration. The scope of these type variables is the declaration that has this prefix, within which the type variables may be used as regular types:

```
tp_decl        ::= [type_cxt] "type" name [formal_params]
                       {type_relation} ";"
object_decl    ::= [type_cxt] rep_role rep_kind name [formal_params]
                       {relation} [field_inits] ";"
predicate_decl ::= [type_cxt] [privacy] "predicate" name [formal_params]
                       {relation} [field_inits] ["when" expr] ";"
type_ext_decl  ::= [type_cxt] [privacy] "extend" "type" named_type
                       {type_relation} ";"
obj_ext_decl   ::= [type_cxt] [privacy] "extend" extend_kind named_object
                       {relation} [field_inits] ";"
signature_decl ::= [type_cxt] [privacy] "signature" method_name
                       "(" [arg_types] ")" [type_decl] ";"
method_decl    ::= [type_cxt] [privacy] impl_kind method_name
                       "(" [formals] ")" [type_decl] {pragma}
                       "{" (body | prim_body) "}" [";"]
```

```
field_sig_decl ::= [type_cxt] [field_privacy] ["var"] "field" "signature"
                       msg_name [formal_params] "(" arg_type ")"
                       [type_decl] ";"
field_decl     ::= [type_cxt] [field_privacy] ["shared"] ["var"] "field"
                       field_kind msg_name [formal_params] "(" formal ")"
                       [type_decl] {pragma} [":=" expr] ";"
method_name    ::= msg_name [formal_params] | op_name
type_cxt       ::= "forall" formal_param { "," formal_param }
formal_params  ::= "[" formal_param { "," formal_param } "]"
formal_param   ::= ["`"] name_binding
name_binding   ::= name                      declares a type variable called name
```

To use a polymorphic declaration, it must be *instantiated* by providing the *instantiating type* for each type variable. Type variables are "formals" and instantiating types are "actuals" of a parameterized declaration. In the following example an immutable vector object `i_vector`, method `fetch`, and an `extend` declaration are parameterized with type variable `T` (intended to denote the type of the vector elements):

```
forall T: template object i_vector[T];
forall T: extend i_vector[T] isa collection[T];
forall T: method fetch(a@:i_vector[T], index:int):T  { ... }
var my_vec:i_vector[num] := concrete object isa i_vector[num];
var result:num := fetch(my_vec, 5);
```

Note that parameterization is unsound and is disallowed in the following cases:

- Variable (`let`) declarations cannot be parameterized.
- The type of a field cannot reference any type parameters except those of the object to which the field is attached. Moreover, the type of a field attached to a concrete object cannot reference any type parameters.

4.1.2 Explicit and Implicit Parameterization

A type parameter is *explicit* if the corresponding instantiating type is to be explicitly provided by clients of the declaration, or *implicit* if it is to be inferred automatically by the typechecker. A polymorphic object or method declaration specifies which parameters are explicit by listing the corresponding type variables in brackets following the name of the declared entity. The explicitly instantiating types should be similarly given in brackets whenever the object, type, or message is referenced:

```
named_object   ::= name [params]
named_type     ::= name [params]
message        ::= msg_name [params] "(" [exprs] ")"
dot_msg        ::= dot_expr "." msg_name [params] ["(" [exprs] ")"]
params         ::= "[" types "]"
```

In the previous example, the `i_vector` object is explicitly parameterized and requires all clients to provide instantiating types, as in `i_vector[num]`. Method `fetch`, in contrast, is parameterized implicitly; in the expression `fetch(my_vec, 5)` the instantiating type for `T` is inferred to be `num`. Inference allows the programmer to avoid writing the often obvious instantiating types; it is a key feature of Cecil. It is described in more details in Section 4.3.

The number of explicit type parameters is considered part of the "name" of the declared entity. For example, multiple objects can be declared with the same name, as long as they are declared with different numbers of explicit type parameters.* Also, method lookup is extended to include the number of explicit parameters of candidate methods (which must match that at the callsite) as part of the method selection process. Method lookup does not depend on the instantiating types, however. For example, message sends `foo[int]()` and `foo[string]()` will always be dispatched to the same method implementation, but `foo[int,string]()` will be dispatched to a different implementation.

Parameterized extension declarations are only instantiated internally by the typechecker. Their type parameters therefore are always implicit.

4.1.3 Omitting the Explicit `forall` Clause: the Backquote Sugar

Type variables can be introduced implicitly — without listing them in the `forall` clause. This provides a more concise notation for parameterized declarations by omitting the explicit `forall` prefix. A type variable is introduced implicitly in a declaration if:

- it is preceded by a backquote (`` ` ``) somewhere in its header (see more on the backquote sugar in Section 4.2), or
- it appears inside the square brackets following the name of a parameterized object or message being introduced in an object or method declaration, in which case the type variable is an explicit type parameter of that object or method.

So the declarations from the previous example can be rewritten more concisely as:

```
template object i_vector[T];
extend i_vector[`T] isa collection[T];
method fetch(a@:i_vector[`T], index:int):T  { ... }
```

The expressions remain the same since the backquote sugar does not affect whether or not a type parameter is explicit. Note that the names of type variables shadow names of globally visible types, whether the variables are introduced in a `forall` clause, using the backquote sugar, or as an explicit type parameter. For example, if a declaration introduces a type variable called `int`, all occurrences of `int` in that declaration will refer to that type variable and not the predefined integer type.

4.2 Bounded Polymorphism and Type Constraints

It is often necessary to express some assumptions or restrictions on type parameters. For example, a `sort` method can only sort collections whose elements can be compared with each other. A `matrix_multiply` method may require that matrix elements be numbers. This situation is known as bounded polymorphism [Cardelli & Wegner 85]. Cecil supports bounded polymorphism by allowing *type constraints* on type parameters.

* This feature does not interact well with mixed dynamic and static typing, since the number of parameters affects the execution behavior of the program, violating the principle that static types do not affect the execution semantics. In the future, the number of parameters may be removed from the "name" of an object or method, so that parameters are confined to the (optional) static type system.

There are two kinds of type constraints in Cecil. A *subtype constraint* specifies the requirement that one type be a subtype of another. A common use of subtype constraints is to specify upper or lower bounds of type variables. In the following example, the type of matrix elements is constrained to be a subtype of `num`:

```
method matrix_multiply(a:matrix[`T], b:matrix[`T]):matrix[T]
   where T<=num
{
   ...
}
```

A *signature constraint* specifies the requirement that the given signature hold. A common use of signature constraints is to require certain operations to be provided for the type parameters. In the following example, the message send of <= in the body of `sort` is guaranteed to be legal as long as the constraint is satisfied:

```
method sort(a:array[`T]):void
   where signature <=(T,T):bool
{
   ...
   let a_i:T := a!i;
   let a_j:T := a!j;
   if(a_i <= a_j, { ... swap a!i and a!j ... });
   ...
}
```

Type constraints are allowed in the `where` part of the `forall` clause and in a `where` clause following the header of a declaration. The common case of the subtype constraints, type variables' upper bounds, are also allowed wherever that type variable is introduced (in a `forall` clause, using the backquote sugar, or as an explicit type parameter).

```
type_cxt        ::= "forall" formal_param { "," formal_param } [type_cons] ":"
type_cons       ::= "where" type_constraint { "," type_constraint }
type_constraint::= sub_constraint | sig_constraint
sub_constraint  ::= type ("<=" | ">=") type
sig_constraint  ::= "signature" (msg_name [params] | op_name)
                        "(" [arg_types] ")" type_decl
name_binding    ::= name [">=" type] ["<=" type]

tp_decl         ::= [type_cxt] [privacy] "type" name [formal_params]
                        {type_relation} [type_cons] ";"
object_decl     ::= [type_cxt] [privacy] rep_role rep_kind name [formal_params]
                        {relation} [type_cons] [field_inits] ";"
predicate_decl ::= [type_cxt] [privacy] "predicate" name [formal_params]
                        {relation} [type_cons] [field_inits] ["when" expr] ";"
type_ext_decl   ::= [type_cxt] [privacy] "extend" "type" named_type
                        [type_cons] {type_relation} ";"
obj_ext_decl    ::= [type_cxt] [privacy] "extend" extend_kind named_object
                        {relation} [type_cons] [field_inits] ";"
signature_decl ::= [type_cxt] [privacy] "signature" method_name
                        "(" [arg_types] ")" [type_decl] [type_cons] ";"
method_decl     ::= [type_cxt] [privacy] impl_kind method_name
                        "(" [formals] ")" [type_decl] [type_cons] {pragma}
                        "{" (body | prim_body) "}" [";"]
```

```
field_sig_decl ::= [type_cxt] [field_privacy] ["var"] "field" "signature"
                        msg_name [formal_params] "(" arg_type ")"
                        [type_decl] [type_cons] ";"
field_decl     ::= [type_cxt] [field_privacy] ["shared"] ["var"] "field"
                        field_kind msg_name [formal_params] "(" formal ")"
                        [type_decl] [type_cons] {pragma} [":=" expr] ";"
```

The `matrix_multiply` method in the above example can be re-written more concisely as follows (this sacrifices the visual symmetry between the arguments `a` and `b`, but is semantically equivalent, because it introduces exactly the same type variable and constraint):

```
method matrix_multiply(a:matrix[`T<=num], b:matrix[T]):matrix[T]
{ ... }
```

The `@:` syntactic sugar is extended to allow a type variable with an upper bound. This idiom is used when it is desirable to give a more precise type to a formal that is specialized on some object (and the type of the formal is expected to subtype the specializer object). For example,

```
method foo(x@:`T<=bar):T { ... }
```

desugars into

```
method foo(x@bar:`T<=bar):T { ... }
```

It is a useful programming idiom to associate constraints with parameterized types. For example, the type of a binary search tree may require that the comparison operation be defined on the type of the tree element, similarly to the sorting method above:

```
template object binary_tree[T] where signature <=(T,T):bool;
```

Cecil provides a syntactic sugar that automatically inserts these associated constraints. If a backquoted type variable is used as an explicit instantiating parameter of a parameterized type, the constraints that the type associates with its explicit parameter in the corresponding position are imposed on the type variable. For example, in the following method:

```
method insert(t@:binary_tree[`T], elm:T):void { ... }
```

the constraint `where signature <=(T,T):bool` is automatically added, so in the body of `insert` it is legal to send the message <= to tree elements and `elm`.

4.3 Constraint Solving and Type Inference

The following typechecking tasks in Cecil lead to constraint solving:

- To typecheck a message send $m[T_1',...,T_m'](E_1,...,E_n)$, where the types of $E_1,...,E_n$ are $T_1,...,T_n$, the signature constraint `signature` $m[T_1',...,T_m']\ (T_1,...,T_n)$:$\mathtt{T}_{result}$ is solved. Here $\mathtt{T}_{result}$ is a fresh type variable and can be instantiated with some type. The type of the message send is the most specific type that $\mathtt{T}_{result}$ can take on while the signature constraint can be solved successfully.
- To check whether S_1 is a subtype of S_2, the subtype constraint $S_1 <= S_2$ is solved. S_1 is a subtype of S_2 iff the constraint can be solved successfully.
- Whenever a declaration with constraints in its header is instantiated, the instantiated constraints must be solved. If they cannot be solved successfully, such instantiation is not *legal* and so is disallowed.

Given a constraint to solve, *constraint solving* proceeds as follows. A "set-to-be-solved" of constraints is created, initially containing this one constraint. One constraint at a time is picked and removed from this set. A matching constraint is produced from the program declarations, if possible, otherwise constraint solving fails. Two constraints match if they have the same structure (*e.g.*, both are signature constraints for the same message name and number of arguments) and the types in the corresponding positions are the same; fresh type variables may be instantiated with types during matching. While producing the matching constraint, new constraints to be solved may arise, in which case they are added to the set-to-be-solved. Constraint solving succeeds when the set-to-be-solved becomes empty.

The matching constraint can be produced either by taking a constraint or declaration available in the program, or by combining other constraints produced from the program declarations. More specifically:

- A polymorphic subtype or signature declaration present in the program can be instantiated by substituting types or fresh type variables for its type variables; its constraints, if any, need to be solved and are added to the set-to-be-solved. A subtype of signature declaration with no type variables is treated as an available constraint itself.
- When typechecking the body of a polymorphic declaration, the constraints in its header are available.
- Constraints can be combined based on the standard properties of subtyping, such as transitivity, and of signatures, such as contravariance. For example, if the program contains declarations `signature =(num,num):bool` and `extend int isa num`, they can be combined to yield the constraint `signature =(int,num):bool`. Matching of types and substitutions of types for fresh type variables are performed as needed.

Inference of instantiating types is the part of constraint solving whereby the typechecker decides how to instantiate polymorphic declarations, i.e., what types to substitute for type variables. Intuitively, when typechecking a message send, the typechecker tries to find the "best" instantiations of declarations involved in solving the signature constraint, i.e., the instantiations that lead to the most precise result type. When checking whether a type is a subtype of another, the typechecker only needs to prove that some appropriate instantiations exist.

Consider, for example, typechecking the message send `print(my_coll)` in the context of the following declarations:

```
abstract object printable;
signature print(p:printable):void;

abstract object collection[T];
extend collection[`T <= printable] isa printable;

method print(a@:collection[`T <= printable]):void
{  print("[");  a.do(&(e:T){ print(e) });  print("]")  }

extend string isa printable;              -- assume string implements print

let my_coll:collection[string] := ...;
print(my_coll);
```

Since `my_coll` has type `collection[string]`, in order to check this send, the typechecker needs to solve the constraint `signature print(collection[string]):`$\mathtt{T}_{result}$ where $\mathtt{T}_{result}$ is a fresh type variable; this is the first constraint in the set-to-be-solved. A matching constraint can be produced by instantiating the signature corresponding to `method print (collection['T<=printable]):void`. Two substitutions need to take place to achieve matching: `void` for $\mathtt{T}_{result}$ and `string` for `T`. Also, the signature's instantiated subtype constraint `string <= printable` is added to the set-to-be-solved. This time, the program already contains an exactly matching declaration (the subtype declaration corresponding to `extend string isa printable`). No more constraints are added to the set-to-be-solved, so constraint solving is complete. The instantiating type `string` was automatically inferred for the type variable `T` of the polymorphic method declaration.

4.4 F-bounded Polymorphism

This subsection describes an example of advanced use of the Cecil type system, F-bounded polymorphism. As we will see, no special support for this powerful idiom is needed in the type system — it is made possible by allowing constraints to be recursive, whereby a type variable can appear in its own bound.

For our first example, let us consider an abstract object `ordered` and a binary method >. A *binary method* is a method that expects two arguments of similar types; the > method can be applied, for example, to two numbers or two strings, but not a string and a number. We would like to define this method once, in the `ordered` object, and have other objects, such as `num` and `string`, inherit it. The simplest way to achieve it seems to be as follows:

```
abstract object ordered;
   signature <=(x:ordered, y:ordered):bool;
   method >(x:ordered, y:ordered):bool { not(x <= y) }
extend num isa ordered;
extend string isa ordered;
```

This code, however, leads to an undesirable effect. Since > and <= are defined for `ordered` and `num` and `string` are its subclasses, we are required to write implementations of <= to compare a `num` and a `string`, which we may not want. To avoid mixing of subclasses of `ordered`, we can apply F-bounded polymorphism as follows:

```
abstract object ordered[T] where T <= ordered[T];
   signature <=(x:'T, y:'T):bool where T <= ordered[T];
   method >(x:'T, y:'T):bool where T <= ordered[T]
      { not(x <= y) }
extend num isa ordered[num];
   method <=(x@:num, y@:num):bool { ... }

extend string isa ordered[string];
   method <=(x@:string, y@:string):bool { ... }
```

Now method > can be instantiated with `num` for `T` (because the instantiated constraint `num <= ordered[num]` can be solved: there is a corresponding declaration in the program) or with `string` for `T`, but cannot with `(string|num)` for `T` (which would be required in order to compare a `num` and a `string`).

With this scheme, in addition to defining binary methods itself, `ordered` and all its subtypes can inherit binary methods from other objects, for example:

```
abstract object comparable[T] where T <= comparable[T];
   signature =(x:`T, y:`T):bool where T <= comparable[T];
   method !=(x:`T, y:`T):bool where T <= comparable[T]
      { not(x = y) }

extend ordered[`T] isa comparable[T];
   method =(x@:num, y@:num):bool { ... }
   method =(x@:string, y@:string):bool { ... }
```

Moreover, `num` can have subtypes, such as `int` or `float`, which can be compared with each other, but not with `string` or its subtypes:

```
extend int isa num;
extend float isa num;
3 != 3.14     -- legal
```

F-bounded polymorphism can be applied similarly to express families of two or more mutually recursive types. For example, consider a simplified model-view framework, where the model and the view must be able refer to each other and invoke operations on each other.* Moreover, instances of the model-view framework, such as a drawing model and a drawing view, must be able to invoke specific operations on each other without loss of type safety. The following code shows how the generic model-view framework can be defined:

```
abstract object model[`M <= model[M,V], `V <= view[M,V]];
   field views(@:model[`M,`V]):set[V] := new_set[V]();
   method register_view(m@:model[`M,`V], view:V):void {
      m.views.add(view); }
   method update(m@:model[`M,`V]):void {
      m.views.do(&(v:V){
         v.update();
      }); }

abstract object view[`M <= model[M,V], `V <= view[M,V]];
   field model(@:view[`M,`V]):M;
   signature update(v@:view[`M,`V]):void;
```

Both `model` and `view` are parameterized by the type of the model and the view with the corresponding upper bounds on these two parameters. Correspondingly, the code for the `model` and `view` objects is parameterized by the actual types of the instantiation of the framework. For example, the following code instantiates the generic model-view framework to construct a bitmap drawing model and view:

```
template object drawing isa model[drawing,drawing_view];
   field bitmap(@:drawing):bitmap;
   method set_pixel(m@:drawing, pos:position, value:color):void {
      bitmap.pixel(pos) := value;
      m.views.do(&(v:drawing_view){
         v.update_pixel(pos, value);
```

* Thanks to Gail Murphy for suggesting this problem to us.

```
    }); }

template object drawing_view isa view[drawing,drawing_view];
   method update(v@:drawing_view):void {
      screen.plot(v.model.bitmap); }
   method update_pixel(v@:drawing_view, pos:position, value:color):void {
      screen.plot_pixel(pos, value); }
   method new_drawing_view(m@:drawing):drawing_view {
      concrete object isa drawing_view { model := m } }
```

Both `drawing` and `drawing_view` add new operations that need to be called by the other type. By parameterizing `model` as was done, the type of the `views` field in `drawing` is known statically to be set of (subtypes of) `drawing_view`. This knowledge allows the `set_pixel` operation in `drawing` to invoke the `update_pixel` operation without generating either a static type-error or requiring a dynamic "typecase" or "narrow" operation. Similarly, because of the way `view` is parameterized, the `model` field in its child `drawing_view` will be known statically to refer to a (subtype of) `drawing`, allowing the `update` operation of `drawing_view` to access the `bitmap` field of the model in a statically type-safe manner. Note that it is legal to instantiate `model` and `view` with `drawing` and `drawing_view`, because the instantiated subtype constraints can be solved successfully.

Alternatively to the unparameterized `drawing` and `drawing_view`, the programmer could parameterize them in a way similar to how `model` and `view` are parameterized, in order to allow further refinement of these two types. This is similar to having the parameterized `ordered` subtype of `comparable`, as opposed to the unparameterized `num` and `string` subtypes of `ordered`, in our earlier examples.

4.5 Related Work

We categorize related work on polymorphic type systems for object-oriented languages into several groups: languages based on F-bounded polymorphism and explicit subtyping, languages based on `SelfType` or matching, languages based on signature constraints and implicit structural subtyping, languages based on instantiation-time checking, languages based on covariant redefinition, and languages offering local type inference. Cecil includes the core expressiveness of both F-bounded polymorphism (and its restrictions `SelfType` and matching) and signature constraints, provided uniformly over a wide range of declarations. Except where noted below, other languages based on these ideas support strict subsets of the expressiveness of Cecil, although sometimes with more compact syntax. Also, the other languages do not support multi-methods, complete separation of inheritance from subtyping, and least-upper-bound and greatest-lower-bound type expressions, except where noted below. In most other systems, classes corresponds to Cecil's abstract or template objects with parallel subtype and inheritance hierarchies.

4.5.1 Languages Based on F-Bounded Polymorphism

Pizza is an extension to Java based on F-bounded polymorphism [Odersky & Wadler 97]. Like Cecil, Pizza supports classes with mutually recursive bounds, crucial for supporting interrelated families of classes such as the `model-view` example from section 4.4. Also like Cecil, Pizza automatically infers instantiating type parameters of polymorphic methods and constructors, although the instantiating parameters must match the actual argument types exactly, which is more restrictive than Cecil which can infer appropriate supertypes of the argument types. Pizza lacks signature constraints and the resulting implicit structural subtyping. Pizza does not support any subtyping between different instances of a parameterized type, such as the desirable and legal subtyping between different read-only interfaces to collection types as in our `i_vector` example. Pizza also inherits several restrictions from its Java base, including that it does not allow contravariant method overriding. Pizza extends Java with first-class, lexically nested functions and with algebraic data types and pattern-matching. The authors justify introducing algebraic data types by claiming that classes allow new representations to be added easily but not new operations, while algebraic data types support the reverse. Cecil's multi-methods enable both new representations and new operations to be added easily, avoiding the need for new language constructs.

Bruce, Odersky, and Wadler [Bruce et al. 98] recently proposed to extend Pizza with special support for declaring families of mutually recursive classes. They argue that pure F-bounded polymorphism is too cumbersome for programmers to use in practice. We have not found pure F-bounded polymorphism to be untenable, however; the `model-view` example from section 4.4 illustrates our approach. Our experience may be better than theirs because our multi-method framework encourages us to treat each argument and parameter symmetrically and uniformly, while their model is complicated by the asymmetry between the implicit receiver and the explicit arguments. Nevertheless, we are working on syntactic sugars that would make the more sophisticated uses of F-bounded polymorphism simpler.

Agesen, Freund, and Mitchell propose a similar extension to Java [Agesen et al. 97]. It differs from Pizza and Cecil in being able to parameterize a class over its superclass. However, this feature cannot be typechecked when the abstraction is declared, but instead must be rechecked at each instantiation.

Haskell's type classes can be viewed as a kind of F-bounded polymorphism [Wadler & Blott 89]. Haskell automatically infers the most-general parameterization and constraints on functions that take polymorphic arguments, as well as automatically inferring instantiations on calls to such functions; Cecil requires polymorphic methods to explicitly declare type variables and constraints over these variables. (In some cases, Haskell cannot unambiguously infer instantiations.) However, Haskell is not truly object-oriented, in that after instantiation, no subtype polymorphism remains; values of different classes but a common supertype cannot be mixed together at run-time, preventing for instance lists of mixed integers and floats.

$ML_{\leq}$ is a powerful polymorphic object-oriented language supporting multi-methods [Bourdoncle & Merz 97]. $ML_{\leq}$ supports subtyping directly, but treats inheritance as a separate syntactic sugar (which must follow the subtyping relation). Similarly to Cecil, $ML_{\leq}$ constrains type variables using sets of potentially recursive subtype constraints, supports inference of type parameters to methods, and supports least-upper-bound type expressions (although not greatest-lower-bound type expressions). $ML_{\leq}$ also supports parameterization over type constructors, while in Cecil type constructors must be instantiated before use. $ML_{\leq}$ supports explicit declarations of co- and contravariant type parameters of type constructors, while Cecil uses polymorphic subtype declarations to achieve more general effects. $ML_{\leq}$ only allows subtyping between types in the same type constructor "class," however, which for instance restricts subtyping to be between types with the same number of type parameters with the same variance properties, and $ML_{\leq}$ does not support other forms of constrained subtyping, conformance, or inheritance. Cecil supports multiple polymorphic signature declarations for the same message, while $ML_{\leq}$ allows only a single signature declaration per message. $ML_{\leq}$ is purely functional and side-effect-free.

4.5.2 Languages Based on `SelfType` or Matching

Some languages provide only restricted forms of F-bounded polymorphism. In TOOPLE [Bruce et al. 93] and Strongtalk [Bracha & Griswold 93], a special type `SelfType` is introduced, which can be used as the type of method arguments, results, or variables; roughly speaking, a class *C* with references to `SelfType` can be modeled with the F-bounded declaration

```
forall SelfType where SelfType <= C[SelfType]:
 template object C[SelfType];
```

`SelfType` supports binary methods like <= and methods like `copy` that return values of exactly the same type as their receiver, but it does not support other kinds of F-bounded parameterization. Other languages provide a related notion called matching, which allows a kind of F-bounded polymorphism where a single type variable is bounded by a function of itself (but of no other type variables); languages with matching include PolyTOIL [Bruce et al. 95b] and LOOM [Bruce et al. 97]. The key advantage of `SelfType` and matching is convenient syntactic support for a common idiom, but it is less powerful than F-bounded polymorphism. Additionally, the LOOM language drops subtyping altogether in favor of matching, which costs it the ability to support run-time mixing of values of different classes but common supertypes, such as performing binary operations on the elements of a list of mixed integers and floats. `SelfType` and matching also are weaker than F-bounded polymorphism in that they force subclasses to continually track the more specific type; they cannot stop narrowing at some subclass and switch to normal subtyping below that point. For example, with F-bounded polymorphism, the parameterized `ordered` type can have its type parameter "narrowed" and then fixed (say at `ordered[num]`), allowing subtypes of the fixed type (such as `int` and `float`) to be freely mixed. This open/closed distinction for recursive references to a type was noted previously by Eifrig *et al.* [Eifrig et al. 94].

4.5.3 Languages Based on Signature Constraints and Implicit Structural Subtyping

Some languages use collections of signatures to constrain polymorphism, where any type which supports the required signatures can instantiate the parameterized declaration. These systems can be viewed as treating the signature constraints as defining "protocol" types and then inferring a structural subtyping relation over user-defined and protocol types. This inference is in contrast to the systems described earlier which require that the protocol types be declared explicitly, and that legal instantiations of the protocols be declared as explicit subtypes. Implicit structural subtyping can be more convenient, easier to understand, more adaptable to program evolution, and better suited to combining separately written code without change, while explicit by-name subtyping avoids inferring subtyping relations that ignore behavioral specifications, and may interact better with inheriting default implementations of protocol types. Neither is clearly better than the other; Cecil supports both easily. In addition, Cecil allows new supertypes to be added to previously declared types, avoiding one limitation of explicit subtyping when adding new explicit protocol types and adapting previously written objects to conform to them.

Strongtalk is a type system for Smalltalk where programmers define protocol types explicitly, use protocols to declare the types of arguments, results, and variables, and let the system infer subtype and conformance relations between protocols and classes; like Cecil, subtyping and inheritance are separated. Precise details of the type system are not provided, but it appears that Strongtalk supports explicit parameterization (but without constrained polymorphism) for protocols and classes, a kind of parametric typing with dependent types and type inference for methods, least-upper-bound type expressions, and a form of `SelfType`. To avoid accidental subtyping, a class may be branded with one or more protocols. Like Cecil, type declarations and typechecking are optional in Strongtalk.

Interestingly, a later version of Strongtalk appears to have dropped inferred structural subtyping and brands in favor of explicit by-name subtyping [Bracha 96]. This later version also introduces the ability to declare that different instantiations of a parameterized type are subtype-related either co- or contravariantly with respect to its parameter types. Both Strongtalk systems are subsets of Cecil's type system.

Theta [Day et al. 95, Liskov et al. 94] and PolyJ [Myers et al. 97] support signature constraints called `where` clauses. Unlike Cecil, only explicit type variables are supported, and clients must provide instantiations of all type variables when using a parameterized abstraction. No subtype relation holds between different instantiations of the same parameterized type, preventing idioms such as the covariantly related read-only collection interfaces.

Recursively constrained types are the heart of a very sophisticated type system [Eifrig et al. 95]. In this system, type variables and sets of constraints over them are automatically inferred by the system. Subtyping is inferred structurally, viewing objects as records and using standard record subtyping rules. Technically, the constraints on type variables are (mutually recursive) subtype constraints, but anonymous types may be introduced as part of the subtype constraints, providing a kind of signature constraint. Instead of instantiating polymorphic entities and inferring ground types for expressions, their system simply checks whether the inferred constraints over the whole program are satisfiable, without ever solving the constraints. For example, when computing the type of the result of a message, their system may return a partially constrained type variable, while Cecil must infer a unique, most-specific ground type. As a result, their system can typecheck programs Cecil cannot. On the other hand, because Cecil computes named types for all subexpressions, it can give simpler type error messages for incorrect programs; recursively constrained types can provide only the constraint system that was unsatisfiable as the error message, and this constraint system may be as large as the program source code itself. Their system limits syntactically where least-upper-bound and greatest-lower-bound subtype constraints can appear to ensure that such constraints can always be solved, while Cecil places no syntactic limits but may report a type error due to incompleteness of the particular deterministic algorithm used by the typechecker.

4.5.4 Languages Based on Instantiation-Time Checking

Some languages, including C++ [Stroustrup 86] and Modula-3 [SRC], dispense with specifying constraints on type variables entirely, relying instead on checking each instantiation separately. These languages are very flexible in what sort of parameterized declarations and clients can be written, as the only constraints that need be met are that the individual instantiations made in some program typecheck, and they are simple for programmers to use. (C++ also allows constant values as parameters in addition to types.) However, dropping explicit constraints on instantiating type variables loses the ability to check a parameterized declaration for type correctness once and for all separately from its (potentially unknown) clients, loses the specification benefit to programmers about how parameterized declarations should be used, and forces the source code of parameterized entities to be made available to clients in order for them to typecheck instantiations.

4.5.5 Languages Based on Covariant Redefinition

Some languages support bounded polymorphic classes through covariant redefinition of types or operations: a polymorphic class is defined as a regular class that has an "anchor" type member initialized to the upper bound of the type parameter, and instances are made by defining subclasses that redefine some anchor types to selected subtypes. Instances may themselves be further subclassed and their anchor types narrowed. Eiffel supports covariant overriding of methods and instance variables, and uses the `like` construct to refer to anchors [Meyer 92]; Eiffel also supports unbounded parameterized classes as well. Beta supports virtual patterns as anchor classes [Madsen & Møller-Pedersen 89, Madsen et al. 93], and Thorup adapted this idea in his proposed virtual types extension to Java [Thorup 97]. While all of these mechanisms seem natural to programmers in many cases and are syntactically concise, they suffer from a loss of static type safety. In contrast, Cecil can directly support all of the standard examples used to justify such mechanisms (including binary methods and the `model-view` example), for instance using one or more mutually recursive F-bounded type parameters, without sacrificing static type safety. We are working on syntactic support for the general pattern of mutually recursive F-bounded type parameters, in hopes of achieving the same syntactic conciseness and programmer comprehensibility as well.

4.5.6 Languages Offering Local Type Inference

The work on local type inference in an extension of $F_{\leq}$ [Pierce & Turner 98], especially the "local type argument synthesis," is very similar to inference of instantiating types in Cecil: they address a similar problem and use a similar inference algorithm. Their setting is different from Cecil's: they work within an impredicative type system whereas Cecil's is essentially predicative. In contrast with their system, Cecil handles F-bounded quantification, signature constraints, by-name subtyping, and overloading (with multiple dispatch). An earlier work on type inference in $F_{\leq}$ [Cardelli 93] presents a faster algorithm which is more restrictive in some cases due to asymmetric treatment of method arguments.

A similar kind of type inference is also offered by GJ, a language that adds parameterized types to Java [Bracha et al. 98]. Compared to its predecessor, Pizza, in GJ the type of an expression does not depend on its context, and the type inference supports subsumption and empty collections (which may be considered as having multiple incomparable collection types). GJ only provides non-variant type parameters whereas in Cecil covariant or contravariant type parameters can be expressed using polymorphic subtype declarations and are supported by type inference. Type inference in GJ seeks to find the smallest instantiating types for type variables, whereas the goal of type inference in Cecil is to infer the most specific type of an expression (which may be achieved, for example, with the biggest instantiating type for a contravariant type parameter). GJ supports F-bounded polymorphism, but does not provide other advanced language constructs, such as signature constraints, independently parameterized subtype declarations, and multi-methods. The authors of GJ report on the positive experience with their 20,000-line GJ compiler (written in GJ, too) which extensively uses parameterization for container classes and the Visitor pattern. The 125,000-line Vortex compiler written in Cecil [Dean et al. 96] also uses parameterization extensively for container classes as well as in heavily parameterized optimization and interprocedural analysis frameworks [Litvinov 98]. Since Cecil allows additions of new multi-methods and new branches of multi-methods to the existing code, there is no need to use the Visitor pattern in Vortex.

5 Modules

Object-oriented methods encourage programmers to develop reusable libraries of code. However, multi-methods can pose obstacles to smoothly integrating code that was developed independently. Unlike with singly-dispatched systems, if two classes that subclass a common class are included into a program, it is possible for incompleteness or inconsistency to result. The additional expressiveness and flexibility of multi-methods creates new pitfalls for integration.

Encapsulation and modularity of multi-methods is a related problem. To enable easier program reuse and maintenance, it is often desirable to encapsulate a data structure's implementation. However, in a multiply-dispatched language achieving this encapsulation is less straightforward than it would be in either an abstract data type based language, such as CLU, or a singly dispatched object-oriented language, such as C++ or Smalltalk. In ADT-based or singly-dispatched languages, direct access to an object's representation can be limited to a statically-determined region of the program. An earlier approach to encapsulation in Cecil suffered from the problem that privileged access could always be gained by writing methods that specialized on the desired data structures [Chambers 92b].

The Cecil module system has been designed to support integration of separately developed code, encapsulation, and modular design. This system can restrict access to parts of an implementation to a bounded region of program text while preserving the flexibility of multi-methods. Individual modules can be reasoned about and typechecked in isolation from modules not explicitly imported. Modules can *extend* existing modules with subclasses, subtypes, and augmenting multi-methods. If any conflicts arise between independent extensions, they are resolved through *resolving modules* that extend each of the conflicting modules. A simple check for the presence of the necessary resolving modules is all that is needed at link-time to guarantee type safety. Chambers and Leavens describe the Cecil module system in more detail [Chambers and Leavens 94].

The syntax of declarations is extended to support modules as follows:

```
decl            ::= module_decl
                  | import_decl
                  | let_decl
                  | tp_decl
                  | type_ext_decl
                  | object_decl
                  | obj_ext_decl
                  | predicate_decl
                  | disjoint_decl
                  | cover_decl
                  | divide_decl
                  | signature_decl
                  | method_decl
                  | field_sig_decl
                  | field_decl
                  | precedence_decl
                  | include_decl
                  | prim_decl
privacy         ::= "public" | "protected" | "private"
module_decl     ::= [privacy] "module" module_name [extension] "{"
                        {friendship | decl} "}" [";"]
```

```
extension       ::=  “extends” module_names
friendship      ::=  “friend” module_names “;”
module_names    ::=  module_name {“,” module_name}
module_name     ::=  name
import_decl     ::=  [privacy] “import” [“friend”] module_names “;”
```

Also, most declarations have an optional privacy annotation allowed.

[The precise semantics of modules is still under development.]

6 Related Work

Cecil builds upon much of the work done with the Self programming language [Ungar & Smith 87, Hölzle *et al.* 91a]. Self offers a simple, pure, classless object model with state accessed via message passing just like methods. Cecil extends Self with multi-methods, copy-down and initialize-only data slots, lexically-scoped local methods and fields, object extensions, static typing, and a module system. Cecil has simpler method lookup and encapsulation rules, at least when considering only the single dispatching case. Cecil's model of object creation is different than Self's. However, Cecil does not incorporate dynamic inheritance, one of the most interesting features of Self; predicate objects are Cecil's more structured but more restricted alternative to dynamic inheritance. Freeman-Benson independently developed a proposal for adding multi-methods to Self [Freeman-Benson 89].

Common Loops [Bobrow *et al.* 86] and CLOS [Bobrow *et al.* 88, Gabriel *et al.* 91] incorporate multi-methods in dynamically-typed class-based object-oriented extensions to Lisp. Method specializations (at least in CLOS) can be either on the class of the argument object or on its value. One significant difference between Cecil's design philosophy and that in CLOS and its predecessors is that Cecil's multiple inheritance and multiple dispatching rules are unordered and report any ambiguities in the source program as message errors, while in CLOS left-to-right linearization of the inheritance graph and left-to-right ordering of the argument dispatching serves to resolve all message ambiguities automatically, potentially masking real programming errors. We feel strongly that the programmer should be made aware of potential ambiguities since automatic resolution of these ambiguities can easily lead to obscure errors in programs. Cecil offers a simpler, purer object model, optional static type checking, and encapsulation. CLOS and its predecessors include extensive support for method combination rules and reflective operations [Kiczales *et al.* 91] not present in Cecil.

Dylan [Apple 92] is a new language which can be viewed as a slimmed-down CLOS, based in a Scheme-like language instead of Common Lisp. Dylan is similar to CLOS in most of the respects described above, except that Dylan always accesses state through messages. Dylan supports a form of type declarations, but these are not checked statically, cannot be parameterized, and are treated both as argument specializers and type declarations, unlike Cecil where argument specializers and argument type declarations are distinct.

Polyglot is a CLOS-like language with a static type system [Agrawal *et al.* 91]. However, the type system for Polyglot does not distinguish subtyping from code inheritance (classes are the same as types in Polyglot), does not support parameterized or parametrically polymorphic classes or methods, and does not support abstract methods or signatures. To check consistency among multi-methods within a generic function, at least the interfaces to all multi-methods of a generic function must be available at type-check-time. This requirement is similar to that of Cecil that the whole program be available at type-check-time to guarantee that two multi-methods are not mutually ambiguous for some set of argument objects.

Kea is a higher-order polymorphic functional language supporting multi-methods [Mugridge *et al.* 91]. Like Polyglot (and most other object-oriented languages), inheritance and subtyping in Kea are unified. Kea's type checking of multi-methods is similar to Cecil's in that multi-methods must be both complete and consistent. It appears that Kea has a notion of abstract methods as well.

Leavens describes a statically-typed applicative language NOAL that supports multi-methods using run-time overloading on the declared argument types of methods [Leavens 89, Leavens & Weihl 90]. NOAL was designed primarily as a vehicle for research on formal verification of programs with subtyping using behavioral specifications, and consequently omits theoretically unnecessary features that are important for practical programming, such as inheritance of implementation, mixed static and dynamic type checking, and mutable state. Other theoretical treatments of multi-methods have been pursued by Rouaix [Rouaix 90], Ghelli [Ghelli 91], Castagna [Castagna *et al.* 92, Castagna 95], and Pierce and Turner [Pierce & Turner 92, Pierce & Turner 93].

The RPDE[3] environment supports *subdivided methods* where the value of a parameter to the method or of a global variable helps select among alternative method implementations [Harrison & Ossher 90]. However, a method can be subdivided only for particular values of a parameter or global variable, not its class; this is much like supporting only CLOS's `eql` specializers.

A number of languages, including C++ [Stroustrup 86, Ellis & Stroustrup 90], Ada [Barnes 91], and Haskell [Hudak *et al.* 90], support static overloading on function arguments, but all overloading is resolved at compile-time based on the static types of the arguments (and results, in the case of Ada) rather than on their dynamic types as would be required for true multiple dispatching.

Trellis[*] supports an expressive, safe static type system [Schaffert *et al.* 85, Schaffert *et al.* 86]. Cecil's parameterized type system includes features not present in Trellis, such as implicitly-bound type variables and uniform treatment of constrained type variables. Trellis restricts the inheritance hierarchy to conform to the subtype hierarchy; it only supports `isa`-style superclasses.

POOL is a statically-typed object-oriented language that distinguishes inheritance of implementation from inheritance of interface [America & van der Linden 90]. POOL generates types automatically from all class declarations (Cecil allows the programmer to restrict which objects may be used as types). Subtyping is implicit (structural) in POOL: all possible legal subtype relationships are assumed to be in force. Programmers may add explicit subtype declarations as a documentation aid and to verify their expectations. One unusual aspect of POOL is that types and classes may be annotated with *properties*, which are simple identifiers that may be used to capture distinctions in behavior that would not otherwise be expressed by a purely syntactic interface. This ameliorates some of the drawbacks of implicit subtyping.

Emerald is another classless object-oriented language with a static type system [Black *et al.* 86, Hutchinson 87, Hutchinson *et al.* 87, Black & Hutchinson 90]. Emerald is not based on multiple dispatching and in fact does not include support for inheritance of implementation. Types in Emerald are arranged in a subtype lattice, however.

[*] Formerly known as Owl and Trellis/Owl.

Rapide [Mitchell *et al.* 91] is an extension of Standard ML modules [Milner *et al.* 90] with subtyping and inheritance. Although Rapide does not support multi-methods and relies on implicit subtyping, many other design goals for Rapide are similar to those for Cecil.

Some more recent languages support some means for distinguishing subtyping from inheritance. These languages include Theta [Day *et al.* 95], Java [Sun 95], and Sather [Omohundro 93]. Theta additionally supports an enhanced CLU-like where-clause mechanism that provides an alternative to F-bounded polymorphism. C++'s private inheritance supports a kind of inheritance without subtyping.

Several languages support some form of mixed static and dynamic type checking. For example, CLU [Liskov *et al.* 77, Liskov *et al.* 81] allows variables to be declared to be of type `any`. Any expression may be assigned to a variable of type `any`, but any assignments of an expression of type `any` to an expression of another type must be explicitly coerced using the parameterized `force` procedure. Cedar supports a similar mechanism through its `REF ANY` type [Teitelman 84]. Modula-3 retains the `REFANY` type and includes several operations including `NARROW` and `TYPECASE` that can produce a more precisely-typed value from a `REFANY` type [Nelson 91, Harbison 92]. Cecil provides better support for exploratory programming than these other languages since there is no source code "overhead" for using dynamic typing: variable type declarations are simply omitted, and coercions between dynamically-typed expressions and statically-typed variables are implicit. On the other hand, in Cecil it sometimes can be subtle whether some expression is statically-typed or dynamically-typed.

7 Conclusion

Cecil is a pure object-oriented language intended to support the rapid construction of reliable, extensible systems. It incorporates a relatively simple object model which is based on multiple dispatching. Cecil compliments this object model with a static type system that describes the interfaces to objects instead of their representations and a module system to group and encapsulate objects and methods. Cecil's type system distinguishes subtyping from code inheritance, but uses notation that strives to minimize the burden on the programmer of maintaining these separate object and type relationships. The type system supports explicitly and implicitly parameterized types and methods to precisely capture the relationships among argument types and result types in a convenient and concise way. Cecil supports both an exploratory programming style and a production programming style, in part by allowing a program to mature incrementally from a dynamically-typed system to a statically-typed system. Some areas of Cecil's design are the subject of current work, including the details of the parameterization mechanism in the static type system, the precise semantics of the module system, and a formal specification of the static and dynamic semantics of the language.

Acknowledgments

The Cecil language design and the presentation in this document have benefitted greatly from discussions with members of the Self group including David Ungar, Urs Hölzle, Bay-Wei Chang, Ole Agesen, Randy Smith, John Maloney, and Lars Bak, with members of the Kaleidoscope group including Alan Borning, Bjorn Freeman-Benson, Michael Sannella, Gus Lopez, and Denise Draper, with the Cecil group including Claudia Chiang, Jeff Dean, Charles Garrett, David Grove, Vassily Litvinov, Vitaly Shmatikov, and Stuart Williams, and others including Peter Deutsch, Eliot Moss, John Mitchell, Jens Palsberg, Doug Lea, Rick Mugridge, John Chapin, Barbara Lerner, and Christine Ahrens. Gary Leavens collaborated with the author to refine the static type system, devise the module system, and develop an efficient typechecking algorithm. Claudia Chiang implemented the first version of the Cecil interpreter, in Self. Stuart Williams augmented this interpreter with a type checker for the monomorphic subset of the Cecil type system. Jeff Dean, Greg DeFouw, Charles Garrett, David Grove, MaryAnn Joy, Vassily Litvinov, Phiem Huynh Ngoc, Vitaly Shmatikov, Ben Teitelbaum, and Tina Wong have worked on various aspects of the Vortex optimizing compiler for object-oriented languages, a.k.a. the UW Cecil implementation. A conversation with Danny Bobrow and David Ungar at OOPSLA '89 provided the original inspiration for the Cecil language design effort.

This research has been supported by a National Science Foundation Research Initiation Award (contract number CCR-9210990), a NSF Young Investigator Award (contract number CCR-945767), a University of Washington Graduate School Research Fund grant, a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM Canada, Xerox PARC, Edison Design Group, and Pure Software.

More information on the Cecil language and Vortex optimizing compiler projects are available via `http://www.cs.washington.edu/research/projects/cecil` and via anonymous ftp from `cs.washington.edu:pub/chambers`.

References

[Agesen et al. 97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings OOPSLA '97*, Atlanta, GA, October 1997.

[Agrawal *et al.* 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In *OOPSLA '91 Conference Proceedings*, pp. 113-128, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(11)*, November, 1991.

[America & van der Linden 90] Pierre America and Frank van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *OOPSLA/ECOOP '90 Conference Proceedings*, pp. 161-168, Ottawa, Canada, October, 1990. Published as *SIGPLAN Notices 25(10)*, October, 1990.

[Andersen & Reenskaug 92] Egil P. Andersen and Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In *ECOOP '92 Conference Proceedings*, pp. 133-152, Utrecht, the Netherlands, June/July 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[Apple 92] *Dylan, an Object-Oriented Dynamic Language*. Apple Computer, April, 1992.

[Barnes 91] J. G. P. Barnes. *Programming in Ada, 3rd Edition.* Addison-Wesley, Wokingham, England, 1991.

[Black et al. 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *OOPSLA '86 Conference Proceedings*, pp. 78-86, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Black & Hutchinson 90] Andrew P. Black and Norman C. Hutchinson. Typechecking Polymorphism in Emerald. Technical report TR 90-34, Department of Computer Science, University of Arizona, December, 1990.

[Bobrow *et al.* 86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *OOPSLA '86 Conference Proceedings*, pp. 17-29, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Bobrow *et al.* 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. In *SIGPLAN Notices 23(Special Issue)*, September, 1988.

[Borning 86] A. H. Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the 1986 Fall Joint Computer Conference*, pp. 36-40, Dallas, TX, November, 1986.

[Bourdoncle & Merz 97] François Bourdoncle and Stephan Merz. Type Checking Higher-Order Polymorphic Multi-Methods. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, January 1997.

[Bracha & Griswold 93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA '93 Conference Proceedings*, pp. 215-230, Washington, D.C., September 1993. Published as *SIGPLAN Notices 28(10)*, October 1993.

[Bracha 96] Gilad Bracha. The Strongtalk Type System for Smalltalk, 1996. OOPSLA '96 Workshop on Extending the Smalltalk Language, available from http://java.sun.com/people/gbracha/nwst.html.

[Bracha et al. 98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA '98 Conference Proceedings*, Vancouver, B.C., October, 1998.

[Bruce et al. 93] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Proceedings OOPSLA '93*, pages 29–46, October 1993. Published as ACM SIGPLAN Notices, volume 28, number 10.

[Bruce et al. 95b] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyToil: A Type-Safe Polymorphic Object-Oriented Language. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Bruce et al. 97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "Match" for object-oriented languages. In *Proceedings ECOOP '97*. Springer-Verlag, June 1997.

[Bruce et al. 98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A Statically Safe Alternative to Virtual Types. In *Proceedings ECOOP '98*, Brussels, Belgium, July 1998. Springer-Verlag.

[Canning *et al.* 89] Peter S. Canning, William R. Cook, Walter L. Hill, John C. Mitchell, and William Olthoff. F-Bounded Quantification for Object-Oriented Programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1989.

[Cardelli & Wegner 85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. In *Computing Surveys 17(4)*, pp. 471-522, December, 1985.

[Cardelli 93] Luca Cardelli. An implementation of Fsub. Research Report 97, Digital Equipment Corporation Systems Research Center, 1993.

[Castagna *et al.* 92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 182-192, San Francisco, June, 1992. Published as *Lisp Pointers 5(1)*, January-March, 1992.

[Castagna 95] Giuseppe Castagna. Covariance and Contravariance: Conflict without a Cause. In *ACM Transactions on Programming Languages and Systems 17(3)*, pp. 431-447, May 1995.

[Chambers *et al.* 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, pp. 49-70, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Chambers *et al.* 91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in Self. In *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Chambers & Ungar 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Conference Proceedings*, pp. 1-15, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(10)*, October, 1991.

[Chambers 92a] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. thesis, Department of Computer Science, Stanford University, March, 1992.

[Chambers 92b] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92 Conference Proceedings*, pp. 33-56, Utrecht, the Netherlands, June/July, 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[Chambers 93a] Craig Chambers. The Cecil Language: Specification and Rationale. Technical report #93-03-05, Department of Computer Science and Engineering, University of Washington, March, 1993.

[Chambers 93b] Craig Chambers. Predicate Classes. In *ECOOP '93 Conference Proceedings*, pp. 268-296, Kaiserslautern, Germany, July, 1993. Published as *Lecture Notes in Computer Science 707*, Springer-Verlag, Berlin, 1993.

[Chambers & Leavens 94] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. In *OOPSLA '94 Conference Proceedings*, pp. 1-15, Portland, OR, October 1994. Published as *SIGPLAN Notices 29(10)*, October 1994. An expanded and revised version to appear in *ACM Transactions on Programming Languages and Systems*.

[Chang & Ungar 90] Bay-Wei Chang and David Ungar. Experiencing Self Objects: An Object-Based Artificial Reality. Unpublished manuscript, 1990.

[Cook 89] W. R. Cook. A Proposal for Making Eiffel Type-Safe. In *ECOOP '89 Conference Proceedings*, pp. 57-70, Cambridge University Press, July, 1989.

[Cook *et al.* 90] William Cook, Walter Hill, and Peter Canning. Inheritance is not Subtyping. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January, 1990.

[Cook 92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In In *OOPSLA '92 Conference Proceedings*, pp. 1-15, Vancouver, Canada, October, 1992. Published as *SIGPLAN Notices 27(10)*, October, 1992.

[Day *et al.* 95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp. 156-168, Austin, TX, October 1995.

[Dean & Chambers 94] Jeffrey Dean and Craig Chambers. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 273-282, Orlando, FL, June 1994. Published as *Lisp Pointers 7(3)*, July-September 1994.

[Dean *et al.* 95a] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization in Object-Oriented Languages. In *Proceedings of the 1995 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95),* La Jolla, CA, June 1995.

[Dean *et al.* 95b] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95)*, Århus, Denmark, August 1995.

[Dean et al. 96] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Eifrig et al. 95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound Polymorphic Type Inference for Objects. In *OOPSLA'95 Conference Proceedings*, pages 169–184, Austin, TX, October 1995.

[Ellis & Stroustrup 90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[Freeman-Benson 89] Bjorn N. Freeman-Benson. A Proposal for Multi-Methods in Self. Unpublished manuscript, December, 1989.

[Gabriel *et al.* 91] Richard P. Gabriel, Jon L White, and Daniel G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. In *Communications of the ACM 34(9)*, pp. 28-38, September, 1991.

[Ghelli 91] Giorgio Ghelli. A Static Type System for Message Passing. In *OOPSLA '91 Conference Proceedings*, pp. 129-145, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(11)*, November, 1991.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Goldberg 84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[Grove *et al.* 95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, Austin, TX, October 1995.

[Grove 95] David Grove. The Impact of Interprocedural Class Analysis on Optimization. In *Proceedings of CASCON '95*, pp. 195-203, Toronto, Canada, November 1995.

[Halbert & O'Brien 86] Daniel C. Halbert and Patrick D. O'Brien. Using Types and Inheritance in Object-Oriented Languages. Technical report DEC-TR-437, Digital Equipment Corp., April, 1986.

[Harbison 92] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Harrison & Ossher 90] William Harrison and Harold Ossher. Subdivided Procedures: A Language Extension Supporting Extensible Programming. In *Proceedings of the 1990 International Conference on Computer Languages*, pp. 190-197, New Orleans, LA, March, 1990.

[Harrison & Ossher 93] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *OOPSLA '93 Conference Proceedings*, pp. 411-428, Washington, D.C., September 1993. Published as *SIGPLAN Notices 28(10)*, October 1993.

[Hölzle *et al.* 91a] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Agesen, and David Ungar. *The Self Manual, Version 1.1*. Unpublished manual, February, 1991.

[Hölzle *et al.* 91b] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In *ECOOP '91 Conference Proceedings*, pp. 21-38, Geneva, Switzerland, July, 1991.

[Hölzle *et al.* 92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. To appear in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June, 1992.

[Hölzle 93] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *ECOOP '93 Conference Proceedings*, pp. 36-56, Kaiserslautern, Germany, July 1993. Published as *Lecture Notes in Computer Science 707*, Springer-Verlag, Berlin, 1993.

[Hudak *et al.* 90] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, Jonathan Young. *Report on the Programming Language Haskell, Version 1.0*. Unpublished manual, April, 1990.

[Hutchinson 87] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* Ph.D. thesis, University of Washington, January, 1987.

[Hutchinson *et al.* 87] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programming Language Report. Technical Report 87-10-07, Department of Computer Science, University of Washington, October, 1987.

[Ingalls 86] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *OOPSLA '86 Conference Proceedings*, pp. 347-349, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Jenks & Sutor 92] Richard D. Jenks and Robert S. Sutor. *Axiom: the Scientific Computing System*. Springer-Verlag. 1992.

[Kiczales *et al.* 91] Gregor Kiczales, James des Rivières, and Daniel G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge, MA, 1991.

[Kristensen *et al.* 87] B. B. Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA Programming Language. In *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, MA, 1987.

[LaLonde *et al.* 86] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings,* pp. 322-330, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Leavens 89] Gary Todd Leavens. *Verifying Object-Oriented Programs that use Subtypes*. Ph.D. thesis, MIT, 1989.

[Leavens & Weihl 90] Gary T. Leavens and William E. Weihl. Reasoning about Object-Oriented Programs that use Subtypes. In *OOPSLA/ECOOP '90 Conference Proceedings*, pp. 212-223, Ottawa, Canada, October, 1990. Published as *SIGPLAN Notices 25(10)*, October, 1990.

[Lieberman 86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*, pp. 214-223, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Lieberman *et al.* 87] Henry Lieberman, Lynn Andrea Stein, and David Ungar. The Treaty of Orlando. In *Addendum to the OOPSLA '87 Conference Proceedings*, pp. 43-44, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 23(5)*, May, 1988.

[Liskov *et al.* 77] Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction Mechanisms in CLU. In *Communications of the ACM 20(8)*, pp. 564-576, August, 1977.

[Liskov *et al.* 81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, Berlin, 1981.

[Liskov et al. 94] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawhat, Robert Gruber, Paul Johnson, and Andrew C. Myers. Theta Reference Manual. Technical Report Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, February 1994.

[Litvinov 98] Vassily Litvinov. Constraint-Based Polymorphism in Cecil:Towards a Practical and Static Type System. In *OOPSLA '98 Conference Proceedings*, Vancouver, B.C., October, 1998.

[Madsen & Møller-Pedersen 89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings OOPSLA '89*, pages 397–406, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[Madsen et al. 93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Krysten Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, MA, 1993.

[Meyer 86] Bertrand Meyer. Genericity versus Inheritance. In *OOPSLA '86 Conference Proceedings*, pp. 391-405, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.

[Meyer 92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[Milner *et al.* 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Mitchell *et al.* 91] John Mitchell, Sigurd Meldal, and Neel Hadhav. An Extension of Standard ML Modules with Subtyping and Inheritance. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, Williamsburg, VA, January, 1991.

[Moon 86] David A. Moon. Object-Oriented Programming with Flavors. In *OOPSLA '86 Conference Proceedings,* pp. 1-8, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Mugridge *et al.* 91] W. B. Mugridge, J. G. Hosking, and J. Hamer. Multi-Methods in a Statically-Typed Programming Language. Technical report #50, Department of Computer Science, University of Auckland, 1991. Also in *ECOOP '91 Conference Proceedings*, Geneva, Switzerland, July, 1991.

[Myers et al. 97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized Types for Java. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

[Nelson 91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Odersky & Wadler 97] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[Omohundro 93] Stephen Omohundro. *The Sather 1.0 Specification*. Unpublished manual, June 1993.

[Pierce & Turner 92] Benjamin C. Pierce and David N. Turner. Statically Typed Multi-Methods via Partially Abstract Types. Unpublished manuscript, October, 1992.

[Pierce & Turner 93] Benjamin C. Pierce and David N. Turner. Object-Oriented Programming Without Recursive Types. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, January, 1993.

[Pierce & Turner 98] Benjamin C. Pierce and David N. Turner. Local Type Inference. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, January 1998.

[Rees & Clinger 86] Jonathan Rees and William Clinger, editors. *Revised^3 Report on the Algorithmic Language Scheme*. In *SIGPLAN Notices 21(12)*, December, 1986.

[Rouaix 90] Francois Rouaix. Safe Run-Time Overloading. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp. 355-366, San Francisco, CA, January, 1990.

[SRC] DEC SRC Modula-3 Implementation. Digital Equipment Corporation Systems Research Center. http://www.research.digital.com/SRC/modula-3/html/home.html.

[Santas 93] Philip S. Santas. A Type System for Computer Algebra. In *International Symposium on Symbolic and Algebraic Computation*. 1993.

[Schaffert *et al.* 85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment, Language Reference Manual. Technical report DEC-TR-372, November, 1985.

[Schaffert *et al.* 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*, pp. 9-16, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Snyder 86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86 Conference Proceedings*, pp. 38-45, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Steele 84] Guy L. Steele Jr. *Common LISP*. Digital Press, 1984.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[Sun 95] Sun Microsystems. *The Java Language Specification.* Unpublished manual, May 1995.

[Teitelman 84] Warren Teitelman. *The Cedar Programming Environment: A Midterm Report and Examination*. Xerox PARC technical report CSL-83-11, June, 1984.

[Thorup 97] Kresten Krab Thorup. Genericity in Java with Virtual Types. In *Proceedings ECOOP '97*, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[Touretzky 86] D. Touretzky. *The Mathematics of Inheritance Systems.* Morgan-Kaufmann, 1986.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Ungar *et al.* 91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing Programs without Classes. In *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Ungar 95] David Ungar. Annotating Objects for Transport to Other Worlds. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp. 73-87, Austin, TX, October 1995.

[Wadler & Blott 89] Philip Wadler and Stephen Blott. How to Make *ad-hoc* Polymorphism Less *ad-hoc*. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.

[Watt *et al.* 88] Steven M. Watt, Richard D. Jenks, Robert S. Sutor, and Barry M. Trager. The Scratchpad II Type System: Domains and Subdomains. In *Proceedings of the International Workshop on Scientific Computation*, Capri, Italy, 1988. Published in *Computing Tools for Scientific Problem Solving*, A. M. Miola, ed., Academic Press, 1990.

Appendix A Annotated Cecil Syntax

In our EBNF notation, vertical bars (|) are used to separate alternatives. Braces ({ ... }) surround strings that can be repeated zero or more times. Brackets ([...]) surround an optional string. Parentheses are used for grouping. Literal tokens are included in quotation marks (" ... ").

A.1 Grammar

a program is a sequence of declaration blocks and statements

```
program        ::= file_body
file_body      ::= { top_decl_block | stmt }
```

a declaration block is an unbroken sequence of declarations where names are available throughout; declaration blocks at the top level can be interspersed with pragmas

```
top_decl_block ::= { decl | pragma }
decl_block     ::= decl { decl }
```

a declaration is a variable, a field, or a method declaration

```
decl           ::= module_decl
                 | import_decl
                 | let_decl
                 | tp_decl
                 | type_ext_decl
                 | object_decl
                 | obj_ext_decl
                 | predicate_decl
                 | disjoint_decl
                 | cover_decl
                 | divide_decl
                 | signature_decl
                 | method_decl
                 | field_sig_decl
                 | field_decl
                 | precedence_decl
                 | include_decl
                 | prim_decl
```

privacy of a declaration defaults to public

```
privacy        ::= "public" | "protected" | "private"
```

modules package up independent subsystems

```
module_decl    ::= [privacy] "module" module_name [extension] "{"
                       {friendship | decl} "}" [";"]
extension      ::= "extends" module_names
friendship     ::= "friend" module_names ";"
module_names   ::= module_name {"," module_name}
module_name    ::= name
```

import declarations specify used modules

```
import_decl    ::= [privacy] "import" ["friend"] module_names ";"
```

variable declarations bind names to objects; if "var" is present then variable is assignable

```
let_decl        ::= [privacy] "let" ["var"] name [type_decl] {pragma}
                        ":=" expr ";"
```

type, representation, and object declarations create new implementations and/or types

```
tp_decl         ::= [type_cxt] [privacy] "type" name [formal_params]
                        {type_relation} [type_cons] ";"     declares an object type
object_decl     ::= [type_cxt] [privacy] rep_role rep_kind name [formal_params]
                        {relation} [type_cons] [field_inits] ";"
rep_role        ::= "abstract"                 only inherited from by named objects;
                                                   allowed to be incomplete
                  |  "template"                only inherited from or instantiated;
                                                   uninitialized fields allowed
                  |  "concrete"                completely usable;
                                                   must be complete and initialized
                  |  ["dynamic"]               completely usable; accesses checked dynamically
rep_kind        ::= "representation"           declares an object implementation
                  |  "object"                  declares an object type and implementation
type_relation   ::= "subtypes" type_patterns
relation        ::= type_relation              type subtypes from type, or impl conforms to type
                  |  "inherits" parents        impl inherits from impl
                  |  "isa" parents             impl inherits from impl, type subtypes from type
parents         ::= named_object_p { "," named_object_p }
field_inits     ::= "{" field_init { "," field_init } "}"
field_init      ::= msg_name [location] ":=" expr
location        ::= "@" named_object
```

predicate object declaration

```
predicate_decl ::= [type_cxt] [privacy] "predicate" name [formal_params]
                        {relation} [type_cons] [field_inits] ["when" expr] ";"
```

declarations of the relationships among predicate objects

```
disjoint_decl   ::= [privacy] "disjoint" named_objects ";"
cover_decl      ::= [privacy] "cover" named_object "by" named_objects ";"
divide_decl     ::= [privacy] "divide" named_object "into" named_objects ";"
named_objects   ::= named_object { "," named_object }
```

extensions adjust the declaration of an existing object and/or type

```
type_ext_decl   ::= [type_cxt] [privacy] "extend" "type" named_type_p
                        [type_cons] {type_relation} ";"
obj_ext_decl    ::= [type_cxt] [privacy] "extend" extend_kind named_object_p
                        {relation} [type_cons] [field_inits] ";"
extend_kind     ::= "representation"           extend representation
                  |  ["object"]                extend both type and representation
```

signature declarations declare method signatures

```
signature_decl ::= [type_cxt] [privacy] "signature" method_name
                        "(" [arg_type_ps] ")" [type_decl_p] [type_cons] ";"
arg_type_ps     ::= arg_type_p { "," arg_type_p }
arg_type_p      ::= [[name] ":"] type_pattern
method_name     ::= msg_name [formal_params] | op_name
msg_name        ::= name
```

implementation declarations define new method implementations; method decls define signatures, too

```
method_decl     ::=  [type_cxt] [privacy] impl_kind method_name
                         "(" [formals] ")" [type_decl_p] [type_cons] {pragma}
                         "{" (body | prim_body) "}" [";"]
impl_kind       ::=  ["method"] "implementation"  declares a method implementation
                |    "method"                     declares a method signature and implementation
formals         ::=  formal { "," formal }
formal          ::=  [name] specializer           formal names are optional, if never referenced
specializer     ::=  location [type_decl_p]       specialized formal
                |    [type_decl_p]                unspecialized formal
                |    "@" ":" named_object_p       sugar for @named_obj_p :named_obj_p
```

field declarations declare accessor method signatures and/or implementations

```
field_sig_decl  ::=  [type_cxt] [field_privacy] ["var"] "field" "signature"
                         msg_name [formal_params] "(" arg_type_p ")"
                         [type_decl_p] [type_cons] ";"
field_decl      ::=  [type_cxt] [field_privacy] ["shared"] ["var"] "field"
                         field_kind msg_name [formal_params] "(" formal ")"
                         [type_decl_p] [type_cons] {pragma} [":=" expr] ";"
field_kind      ::=  empty                        declare accessor method impl(s) and sig(s)
                |    "implementation"             declare just accessor method implementation(s)
field_privacy   ::=  privacy [ ("get" [ privacy "set" ] | "set") ]
```

precedence declarations control the precedence and associativity of binary operators

```
prec_decl       ::= [privacy] "precedence" op_list
                         [associativity] {precedence} ";"
associativity   ::= "left_associative" | "right_associative" | "non_associative"
precedence      ::= "below" op_list | "above" op_list | "with" op_list
op_list         ::= op_name { "," op_name }
```

include declarations control textual file inclusions (implementation specific)

```
include_decl    ::=  "include" file_name ";"
file_name       ::=  string
```

primitive body declarations include an arbitrary piece of code in the compiled file (implementation specific)

```
prim_decl       ::=  prim_body ";"
```

primitive method bodies support access to code written in other languages (implementation specific)

```
prim_body       ::=  "prim" { language_binding }
language_binding::= language ":" code_string
                |    language "{" code_chars "}"
language        ::=  name                         currently recognize rtl and c_++
code_string     ::=  string
code_chars      ::=  brace_balanced_chars         any characters, with balanced use of "{" and "}"
```

body of a method or closure

```
body            ::=  {stmt} result
                |    empty                        return void
stmt            ::=  decl_block
                |    assignment ";"
                |    expr ";"
```

```
result          ::=  normal_return                return an expression
                |    non_local_rtn                return from the lexically-enclosing method
normal_return   ::=  decl_block                   return void
                |    assignment [";"]             return void
                |    expr [";"]                   return result of expression
non_local_rtn   ::=  "^" [";"]                    do a non-local return, returning void
                |    "^" expr [";"]               do a non-local return, returning a result
```

assignment only allowed if name is assignable; returns `void`

```
assignment      ::=  qualified_name ":=" expr
                |    assign_msg                   assignment-like syntax for messages
assign_msg      ::=  lvalue_msg ":=" expr         sugar for set_msg (exprs..., expr)
lvalue_msg      ::=  message
                |    dot_msg
                |    unop_msg
                |    binop_msg
```

expressions

```
expr            ::=  binop_expr
```

binary msgs have lowest precedence

```
binop_expr      ::=  binop_msg | unop_expr
binop_msg       ::=  binop_expr op_name binop_expr
                                                  predecence and associativity as declared
```

unary msgs have second-lowest precedence

```
unop_expr       ::=  unop_msg | dot_expr
unop_msg        ::=  op_name unop_expr            & and ^ are not allowed as unary operators
```

dotted messages have second-highest precedence

```
dot_expr        ::=  dot_msg | simple_expr
dot_msg         ::=  dot_expr "." msg_name [params] ["(" [exprs] ")"]
                               sugar for msg_name[params](dot_expr,exprs...)
```

simple messages have highest precedence

```
simple_expr     ::=  literal
                |    ref_expr
                |    vector_expr
                |    closure_expr
                |    object_expr
                |    message
                |    resend
                |    paren_expr
```

literal constants

```
literal         ::=  integer
                |    float
                |    character
                |    string
```

reference a variable or a named object implementation

```
ref_expr        ::=  qualified_name               reference a local or global variable
                |    named_object                 reference a named object
```

build a vector

```
vector_expr    ::=  "[" [exprs] "]"
exprs          ::=  expr { "," expr }
```

build a closure

```
closure_expr   ::=  [ "&" "(" [closure_formals] ")" [type_decl] ] "{" body "}"
closure_formals::=  closure_formal { "," closure_formal }
closure_formal ::=  [name] [type_decl]        formal names are optional, if never referenced
```

build a new object

```
object_expr    ::=  rep_role rep_kind {relation} [field_inits]
```

send a message

```
message        ::=  msg_name [params] "(" [exprs] ")"
```

resend the message

```
resend         ::=  "resend" [ "(" resend_args ")" ]
resend_args    ::=  resend_arg { "," resend_arg }
resend_arg     ::=  expr                      corresponding formal of sender must be
                                                  unspecialized
                 |  name                      undirected resend (name is a specialized formal)
                 |  name location             directed resend (name is a specialized formal)
```

introduce a new nested scope

```
paren_expr     ::=  "(" body ")"
```

name something perhaps in another module

```
qualified_name ::=  [module_name "$"] name
```

name an object

```
named_object   ::=  qualified_name [params]
named_object_p ::=  qualified_name [param_patterns]
```

type contexts and constraints

```
type_cxt       ::=  "forall" formal_param { "," formal_param }
                        [type_cons] ":"
type_cons      ::= "where" type_constraint { "," type_constraint }
type_constraint::=  sub_constraint | sig_constraint
sub_constraint ::=  type_pattern ("<=" | ">=") type_pattern
sig_constraint ::=  "signature" (msg_name [param_patterns] | op_name)
                        "(" [arg_type_ps] ")" type_decl_p
```

syntax of types

```
types          ::= type { “,” type }
type           ::= named_type
                 |  closure_type
                 |  lub_type
                 |  glb_type
                 |  paren_type
named_type     ::= qualified_name [params]
closure_type   ::= “&” “(” [arg_types] “)” [type_decl]
arg_types      ::= arg_type { “,” arg_type }
arg_type       ::= [[name] “:”] type
lub_type       ::= type “|” type
glb_type       ::= type “&” type
paren_type     ::= “(” type “)”
```

type patterns are types that can contain binding occurrences of implicit type parameters

```
type_patterns  ::= type_pattern { “,” type_pattern }
type_pattern   ::= binding_type_p
                 |  named_type_p
                 |  closure_type_p
                 |  lub_type_p
                 |  glb_type_p
                 |  paren_type_p
binding_type_p ::= “`” name_binding
named_type_p   ::= qualified_name [param_patterns]
closure_type_p ::= “&” “(” [arg_type_ps] “)” [type_decl_p]
lub_type_p     ::= type_p “|” type_p
glb_type_p     ::= type_p “&” type_p
paren_type_p   ::= “(” type_p “)”
```

name_binding** introduces a type variable called **name

```
name_binding   ::= name [“>=” type_pattern] [“<=” type_pattern]

type_decl      ::= “:” type
type_decl_p    ::= “:” type_pattern
```

formal type parameters for objects and methods

```
formal_params  ::= “[” formal_param { “,” formal_param } “]”
formal_param   ::= [“`”] name_binding
```

actual type parameters for objects and methods

```
params         ::= “[” types “]”
```

actual type parameters for types that may contain binding occurrences of implicit type variables

```
param_patterns ::= “[” type_patterns “]”
```

pragmas can be added at various points in a program to provide implementation-specific hints/commands

```
pragma         ::= “(**” exprs “**)”
```

A.2 Tokens

Bold-faced non-terminals in this grammar are the tokens in the full grammar of A.1. As usual, tokens are defined as the longest possible sequence of characters that are in the language defined by the grammar given below. The meta-notations “`one of “...”`”, “`any but` *x*,” and “*x*`..`*y*” are used to concisely list a range of alternative characters. `space`, `tab`, and `newline` stand for the corresponding characters.

```
name              ::=  letter {letter | digit} [id_cont]
                  |    “_” {“_”} op_name          the first underscore is not part of the msg name
op_name           ::=  punct {punct} [id_cont]
                  |    “_” {“_”} name             the first underscore is not part of the msg name
id_cont           ::=  “_” {“_”} [name | op_name]

integer           ::=  [radix] hex_digits         a leading “-” is considered a unary operator
radix             ::=  digits “_”
hex_digits        ::=  hex_digit {hex_digit}
hex_digit         ::=  digit | one of “a..fA..F”

float             ::=  integer “.” hex_digits [exponent]
                  |    integer exponent
exponent          ::=  “^” [“+” | “-”] digits
digits            ::=  digit {digit}

character         ::=  “'” char “'”
string            ::=  “"” { char | line_break } “"”
char              ::=  any | “\” escape_char
escape_char       ::=  one of “abfnrtv'"\?0”
                  |    [“o”|“d”] digit [digit [digit]]
                  |    [“o”] digit [digit [digit]]
                  |    “x” hex_digit [hex_digit]
line_break        ::=  “\” {whitespace} new_line {whitespace} “\”
                                                  characters between \'s are not part of the string
brace_balanced_chars ::=
                       {any but “{”} [“{” brace_balanced_chars “}” {any but “}”}]

letter            ::=  one of “a..zA..Z”
digit             ::=  one of “0..9”
punct             ::=  one of “!#$%^&*-+=<>/?~\|”
```

A.3 White Space

Whitespace is allowed between any pair of tokens in the grammar in A.1.

```
whitespace        ::=  space | tab | newline | comment
comment           ::=  “--” {any but newline} newline    comment to end of line
                  |    “(--” {any} “--)”          bracketed comment; can be nested
```