The Cecil Language
Specification and Rationale

Craig Chambers

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195 USA

Technical Report 93-03-05
March 1993

chambers@cs.washington.edu
(206) 685-2094; fax: (206) 543-2969

Abstract

Cecil is a new purely object-oriented language intended to support rapid construction of high-quality, extensible software. Cecil combines multi-methods with a classless object model, object-based encapsulation, and optional static type checking. 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, optimizing the common case where the two graphs are parallel. Cecil includes a fairly flexible form of parameterization, including both explicitly parameterized objects, types, and methods and implicitly parameterized methods related to the polymorphic functions commonly found in functional languages. By making type declarations optional, Cecil aims to support mixed exploratory and production programming styles.

This document describes the design of the Cecil language as of March, 1993. It 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 initial design of Cecil, an object-oriented language intended to support the rapid construction of high-quality, reusable, extensible software systems [Chambers 92b]. Cecil is unusual in combining a pure, classless (prototype-based) object model, multiple dispatching (multi-methods), and mixed static and dynamic type checking. Cecil was inspired primarily 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].

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 shows signs of diminishing [Chambers *et al.* 89, Chambers & Ungar 91, Hölzle *et al.* 91b, Chambers 92a].

  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 combing multi-methods with traditional object-oriented language concepts, such as encapsulation and static type checking, not found in other multiple dispatching languages.

- *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].

  Another important kind of static declaration distinguishes the external interface of an object from its internal implementation. Cecil includes a mechanism for encapsulating internal details of objects and enforcing this boundary. This encapsulation mechanism works despite Cecil's multiple dispatching base; we are not aware of any other language combining object-based encapsulation and multiple dispatching.

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 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 whose intended uses are more restrictive.

Finally, Cecil does not include certain complex language features that often 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 covariant method redefinition in a single-dispatching language.

Additionally, the environment can 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 the environment providing 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, support for generic arithmetic, a robust implementation with mandatory error checking for messages and primitives, 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. Section 4 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 objects with no knowledge of their underlying representations or implementations.

At the top level, a Cecil program is a collection of object, method, field, and variable declarations and an expression to evaluate to run the program. The syntax of the overall structure is as follows:

```
program        ::=  [ decl_block ] expr [“;”]
decl_block     ::=  decl { decl }
decl           ::=  object_decl | obj_extension |
                    method_decl | field_decl | var_decl
```

The next three subsections describe objects, methods, and fields. Subsection 2.4 describes encapsulation. Subsections 2.5 and 2.6 detail the semantics of message passing in Cecil.

2.1 Objects and Inheritance

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

```
object number;
-- generic number operations here
object int inherits number; -- behavior for integers
-- integer operations here
object small_int inherits int, prim_int; -- fixed-precision integers
-- fixed-precision operations here
object big_int inherits int; -- arbitrary-precision integers
-- arbitrary-precision operations here
object zero inherits int; -- special zero object behavior
-- zero operations here
```

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

```
object_decl    ::=  “object” name { relation } [ field_inits ]
relation       ::=  “inherits” parents
parents        ::=  object { “,” object }
object         ::=  name
```

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 elegance and simplicity but also to avoid introducing a meta-hierarchy. Additionally, 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.

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 parents. The inheritance graph must be acyclic.

Like most object-oriented languages, in Cecil the inheritance graph is static. An object cannot change its ancestry after it has been created, other than as described below with object extension declarations, nor can an object inherit from another anonymous object created at run-time. Each of an object's parents must be an object declared and named using an object declaration (not an object constructor expression). These restrictions unfortunately preclude some interesting language features traditionally associated with prototype-based languages, such as dynamic inheritance as in SELF and delegation to run-time objects as in Actra [Lieberman 86], but it simplifies other parts of the language such as the type system and guarantees a certain amount of program structure to readers of a program.

The inheritance structure of an object may be augmented after the object is created through an object extension declaration:

```
obj_extension  ::= name { relation } ";"
```

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`, `array`, and `closure` 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, an application may need specialized tab-to-space conversion to be provided by strings and other collections of characters. 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. Section 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.

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. In the initial version of Cecil we are using a relatively simple inheritance semantics, but we will consider Eiffel-like extensions after gaining experience with Cecil.

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.

2.2 Methods

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

```
object int inherits number; -- behavior for integers
method is_zero(x@int) { x = 0 }
method factorial(x@int) {
   -- if invokes a user-defined method, with different definitions for the true and the false unique objects
   if(x <= 1,
       { 1 },
       { x * factorial(x - 1) }) }
method for(from@int, to@int, block) {
   var i := from; -- declare and initialize a new local variable
   while({ i <= to }, {
      eval(block, i); -- invoke the block with an argument
      i := i + 1; });
   }
method +(@int, @int) { abstract } -- all concrete children must provide an implementation for +

object small_int inherits int, prim_int; -- fixed-precision integers
   -- prim_int provides primitive integer arithmetic operations
method +(x@small_int, y@small_int) {
   -- prim_add performs primitive arithmetic of (children of) prim_int
   -- prim_add takes a failure block which is invoked if an error (e.g., overflow) occurs
   prim_add(x, y, &(error_code){ (-- code to handle failure (e.g., retry as big_ints) --) }) }
method +(x@small_int, y@big_int) { as_big_int(x) + y }
method as_big_int(x@small_int) {
   (-- code to create an arbitrary-precision integer from a fixed-precision integer --) }

object big_int inherits int; -- arbitrary-precision integers
method +(x@big_int, y@big_int) { (-- code to add arbitrary-precision integers --) }
method +(x@big_int, y@small_int) { x + as_big_int(y) }

object zero inherits int; -- special zero object behavior
method +(@zero, x) { x } -- zero plus anything is that thing
method +(x, @zero) { x }
method +(z@zero, @zero) { z } -- resolve the ambiguity between the previous two methods
method is_zero(@zero) { true }
```

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

```
method_decl    ::=  [privacy] "method" method_name function [";"]
method_name    ::=  name | infix_name
function       ::=  "(" [formals] ")" function_body
formals        ::=  formal { "," formal }
formal         ::=  [name] specializer
specializer    ::=  [location]
location       ::=  "@" object
function_body  ::=  "{" body "}"
               |    "{" "abstract" [";"] "}"
```

Encapsulation and privacy declarations are discussed in section 2.4.

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 `x@int` notation specializes the `x` formal argument on the `int` object, implying that the method is intended to work correctly with any actual argument object that is equal to or a descendant of the `int` object as the `x` formal. An unspecialized formal argument (one lacking a `@...` suffix) is treated as being specialized on a `top` object that is implicitly an ancestor of all other objects; consequently an unspecialized formal can accept any argument object.

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 places no constraints on how those operations are implemented. Type declarations have no effect on method lookup.

Zero, one, or several of a method's arguments may be (explicitly) specialized, thus enabling Cecil methods to emulate normal undispatched functions, singly-dispatched methods, and true multi-methods, respectively. Callers which send a particular message to a group of arguments need not be 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.

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

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, in some sense. When sending a message of a particular name with a certain number of arguments, the method lookup system (described in section 2.5) will resolve the overloaded methods to a single most appropriate method based on the dynamic values of the actual argument objects and the corresponding formal argument specializers of the methods. Cecil multi-methods can simulate normal undispatched functions (by leaving all formals unspecialized) and singly-dispatched methods (by specializing only the first argument). Statically-overloaded functions and functions declared via certain kinds of pattern-matching also are subsumed by multi-methods.

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 and CLOS's mechanisms differ 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.

As mentioned in the previous section, 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.

2.2.2 Statements and Expressions

The syntax of the body of a method is as follows:

```
body           ::=  {stmt} result
result         ::=  empty | expr | “^” [“;”] | “^” expr
stmt           ::=  decl_block
               |    assignment “;”
               |    effect_expr “;”
assignment     ::=  name “:=” expr
               |    expr “.” name “:=” expr
               |    expr infix_name expr “:=” expr
expr           ::=  literal | simple_expr | effect_expr
literal        ::=  integer | float | character | string
simple_expr    ::=  name | object | object_expr | array_expr | closure_expr
effect_expr    ::=  message | resend | “(” body “)”
```

The body of a method is a (possibly empty) sequence of statements and an optional result expression. With the possible exception of method declarations, every statement is terminated by a semicolon. A statement is a declaration block, an assignment to an assignable variable, or an expression that may have side-effects. The bodies of local methods are lexically-scoped within the containing scope. The interactions among nested scopes, method lookup, and other language features is described in more detail in section 2.5.7.

An expression is either a literal, a reference to a variable or a named object, an object constructor expression, an array constructor expression, a closure constructor expression, a message, a resend, or a parenthetical subexpression. The parenthesized “subexpression” has the same syntax as the body of a method; in particular, it introduces a new nested scope and may contain statements and local declarations.

A declaration block is an unbroken sequence of declarations. 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.

Variable declarations have the following syntax:

```
var_decl       ::=  “var” name initializer “;”
initializer    ::=  “=” expr | “:=” expr
```

If the variable is initialized using the `=` symbol, then it is a constant binding. If the `:=` symbol is used, then the variable may be assigned a new value using an assignment statement.* 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 later variables within the same declaration block are considered undefined.[*] This avoids potential misunderstandings about the meaning of apparently self-referential or mutually recursive initializers, 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 is less well defined. At present, the existing textual ordering of variable declarations is used to define an ordering for evaluating variable initializers. We would prefer a semantics which was independent of the "order" of variable declarations at the top level. Possible alternative semantics under consideration that have this property are to restrict variable initialization expressions to be `literal` or `simple_expr` expressions without side-effects (thereby making the issue of evaluation order unimportant) or to eliminate variable declarations at the top level entirely.

The syntax of a message send is as follows:

```
message       ::=  name "(" [exprs] ")"
              |    expr infix_name expr
              |    expr "." name
exprs         ::=  expr { "," expr }
```

A message is written either in prefix form, with the name of the message followed by a parenthesized list of expressions (again, all arguments to the message must be listed explicitly) or in infix form, with the message name in between a pair of argument subexpressions. A message with a name beginning with a letter must be written in prefix form, while a message with a name beginning with a punctuation symbol or an underscore must be written in infix form.[†] Syntactic sugar exists for three common cases:

- *p*.*x* is syntactic sugar for *x*(*p*), for any expression *p* and prefix name *x*,
- *p*.*x* := *q* is sugar for set_*x*(*p*, *q*), for any expressions *p* and *q* and prefix name *x*, and
- *p* * *q* := *r* is sugar for set_*(*p*, *q*, *r*), for any expressions *p*, *q*, and *r* and infix name *.

The latter two sugars are instances of the `assignment` statement form. The semantics of method lookup is described in section 2.5. Resends, a special kind of message send, are described in section 2.6.

At present, the precedence and associativities of infix messages is unspecified. Combinations of infix operators must be explicitly parenthesized. A mechanism that would allow programmers to declare the relative precedences and associativities of infix operators has been designed but not yet incorporated into the language.

[*] It might be preferable to use a different leading keyword, such as `const`, for constant declarations instead of depending on the subtle distinction between = and :=.

[*] Variables thus constitute an exception to the rule that names introduced in a declaration block are visible throughout the declaration block. It would be nice to develop a more consistent scoping rule that works at the top-level as well as within a method.

[†] As described above, prefix form is always used for method declarations.

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.

An array constructor is written as follows:

```
array_expr     ::= “[” [exprs] “]”
```

The result of evaluating an array constructor is a new object that inherits from the predefined `array` object and is initialized with the corresponding elements.

The syntax of a closure object constructor is as follows:

```
closure_expr   ::= “&” function | function_body
```

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 implicit 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 2.2.5.

2.2.3 Method and Closure Results

A method or closure produces a result if and only if a final result expression is present; the result produced is the result of this expression. If absent, the method or closure does not return a usable result; the method or closure can be thought of as returning the special `void` object. The system will report an error if such a method or closure is invoked in a context that expects a result, i.e., if `void` is stored in a variable or is passed as an argument to another method. It is not an error to return a result in a context where none is needed. Note that the presence or absence of a terminating semicolon is significant: the body of a method or closure that does not return a result must either be empty or be terminated with a semicolon, while the body of a method or closure that returns a result must not end in a semicolon. Whether this syntactic distinction is too subtle remains an open issue.

A closure `eval` method may force a *non-local return* by prefixing the result expression with the `^` symbol. 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] or 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.

In Cecil, a method specifies explicitly whether or not it returns a result. In some other languages, including Smalltalk, SELF, Lisp, and most expression-oriented languages, methods (functions) always return results. In cases where a method does not have an obvious result, some conventional value, such as `self` or the value of a particular argument, is returned instead. However, programs which rely on these conventions can break if the conventions are not always observed, and the system frequently cannot check whether the conventions are followed. In Cecil, a method can explicitly elect not return a result, and the system can verify that no caller expects a result, thereby avoiding any possible confusion for methods whose result is not obvious.

Distinguishing function-like methods from procedure-like methods is an experiment. While it seems to solve a problem observed with some other languages, it may be the case that static type checking would solve the problem just as well; each of the languages mentioned above is dynamically-typed. Alternatively, the `void` object could be promoted to first-class status, so that methods that do not return a result would explicitly return the `void` object instead. This approach may be preferable for some methods, such as the `if` method, that can either return a result or not, depending on whether some argument closure returns a result or not.

2.2.4 Abstract Methods

Normally, a method specifies a sequence of Cecil statements and expressions to execute to implement the corresponding message for certain argument representations. Alternatively, a method may be implemented primitively (as described in section 2.2.5), may provide access to an instance variable (as described in section 2.3), or may simply be marked as `abstract`. An abstract method documents that the associated argument specializers expect such a method to be implemented for arguments that inherit from the specializers, but that a default implementation cannot or should not be specified with the specializing objects. An abstract method cannot be invoked but instead must be overridden with a real method implementation. Abstract methods are important when deriving a type from an object as described in section 3.4, and when guaranteeing privileged access to methods implemented in children as described in section 2.4. As described in section 3.7, the static type checker can ensure that no abstract method is invoked at run-time.

2.2.5 Primitive Objects and Methods

Some objects are predefined at system start-up. These include objects that are the shared parents of integer, float, character, string, array, and closure objects. Low-level functionality is provided through special predefined primitively-implemented methods defined on the primitive objects, such as the `prim_add` method defined on the `prim_int` object in the earlier number example.

* Smalltalk-80 is a trademark of ParcPlace Systems.

These primitive methods exploit multiple dispatching to perform type checking of arguments through appropriate argument specializers. For example, the `prim_add` method specializes its first two arguments on the `prim_int` object (its third argument is a closure that is invoked to handle failures such as overflow, thereby providing the necessary hooks to support generic arithmetic at user level). To gain access to primitive behavior, some user-defined object must inherit from the primitive object, as does the `small_int` object in the example, and implement "wrapper" functions that provide a convenient external interface to the primitive behavior, such as the `+` method in `small_int`. Any state needed by these primitive methods, such as the actual integer value in `prim_int` or the elements of the array in the `prim_array` primitive object, is defined internally to the primitive object and automatically copied-down to inheriting user-level objects. This enables any object to inherit sensibly from a primitive object, including such low-level objects as `prim_int` and `prim_array`, unlike other object-oriented languages which do not allow certain low-level classes to be subclassed.

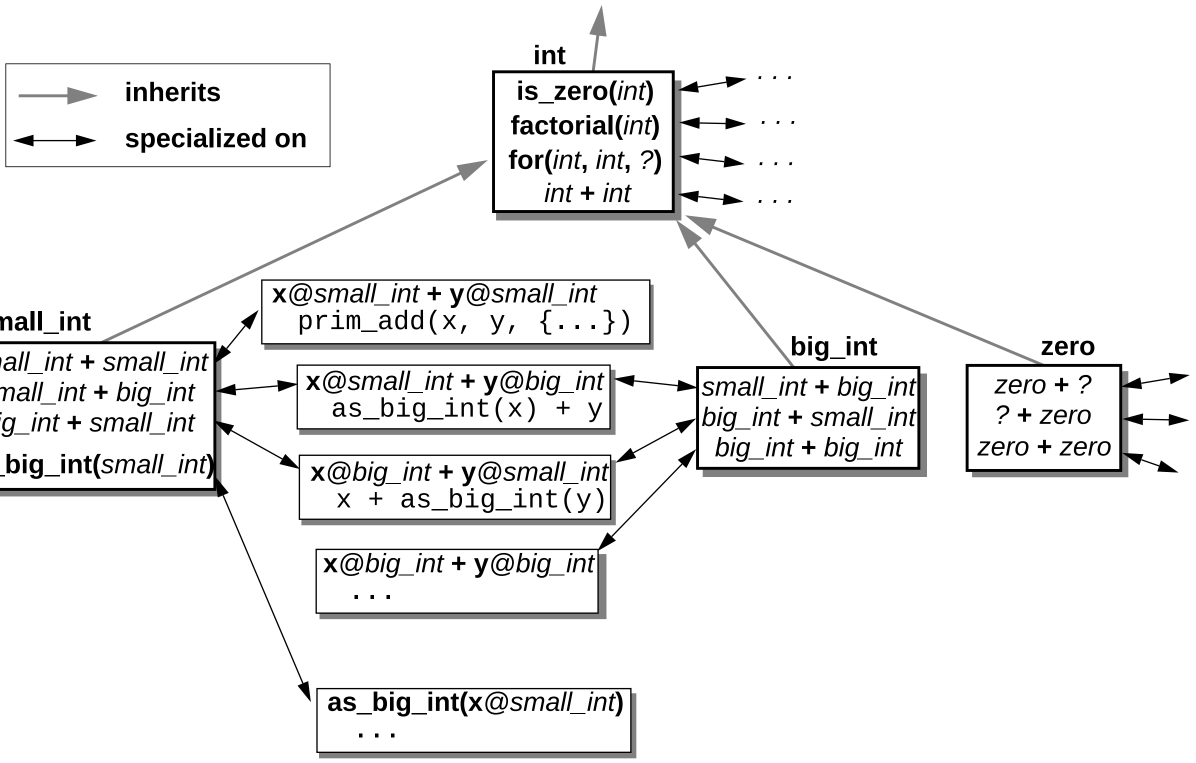
Looping primitive behavior is provided by the `loop` primitive method specialized on the `closure` predefined object. This method repeatedly invokes its argument closure until some closure performs a non-local return to break out of the loop. Other languages such as Scheme [Rees & Clinger 86] avoid the 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 informed to perform tail-recursion elimination.

2.2.6 Programming Environment Support

Cecil supports multi-methods because they are more expressive than traditional singly-dispatched receiver-based methods. Multiple dispatching automates much of the machinery normally implemented by hand using double-dispatching [Ingalls 86]. However, multiple dispatching can alter the normal programming style from a data-abstraction-oriented style to a function-oriented style [Chambers 92b]. With Cecil, we rely on language and programming environment support and tutorial documentation of the intended programming methodology to foster a data-abstraction-oriented view of multi-methods.

Methods and objects are connected through the methods' argument specializers. The methods in the system with the same name have no explicit connection or relationship beyond the programmer's intentions. This contrasts sharply with the traditional approach of linking all multi-methods with the same name into a single generic function object. In Cecil, methods are closely associated with their specializing objects (i.e., the objects whose implementations include the methods), and only weakly associated with each other.

This approach to multi-methods allows programmers to view objects and their connected methods as a unit which implements a data abstraction; the methods defined for a particular object are always directly accessible from the object. This mental image depends heavily on non-hierarchical relationships among objects and methods. Traditional programming environments are text-based, and text is particularly bad at showing non-hierarchical relationships. Consequently, Cecil and similar languages require a graphical interactive programming environment that can display directly non-hierarchical relationships and dynamically-varying views of the relationships. We imagine this environment to show objects on the screen, with their associated multi-methods “contained” within the objects. The user could view the same multi-method from each of its specializing objects; the identity of the multi-method would be illustrated graphically by showing the various “views” of the multi-method as linked to the same object. This interface might look something like the following:



Programmers would design, code, and debug Cecil programs entirely within such an environment; programmers would never need to look at a flat textual form of a Cecil program. The prototype SELF user interface [Chang & Ungar 90] could provide a good starting point for the design of the Cecil user interface, since it is graphical, interactive, and good at displaying non-hierarchical relationships among objects and at reflecting the identity of shared objects.

Of course, the data-abstraction-oriented view of the program is not the only view that may be of interest to the programmer. The programming environment also should be able to present an alternative view of the program in which all methods that together implement some algorithm are displayed simultaneously on the screen (thus capturing some of the programmer’s intentions that link methods). This algorithm-oriented view is more general than the generic function view, since related methods with different names may be displayed simultaneously and unrelated methods that happen to have the same name won’t be displayed.

In many high-productivity programming environments, a single user manipulates a program, frequently using normal object editing operations to edit a heap-based representation of the program. While these environments are powerful, they have historically been difficult to use to manage multi-person cooperative programming. For example, in Smalltalk-80, the primary representation of the program is as objects in the Smalltalk-80 heap [Goldberg 84]. Sharing a program with another programmer requires saving part of this heap to a text file and then loading this text file into the other programmer's system. This process may fail, for example because the two programmers have incompatible naming choices or other local customizations to their systems, or perhaps because the first programmer did not save enough of his extensions to the text file and so the other programmer did not get a complete and consistent update. The SELF system currently relies on text files as the primary representation of SELF programs, but these files can get out of synch with the duplicate representation of the program as objects in the SELF heap if edits are made to the heap-based program without reflecting the change to the backing text files.

Cecil is intended to support multi-person closely-cooperative programming teams as well as single programmers. The primary representation of a Cecil program is a web of interconnected objects and methods, possibly extended with past and future versions of objects and methods. All cooperating programmers see the same program structure, independent of the effect of individual executions of the program; execution cannot change the program. Object and method definitions are declarative statements about the existence of part of the shared object/method web. We believe that high-productivity exploratory programming environments do not require the program to be able to edit itself in a way that would be incompatible with this declarative view, and therefore that this design will continue to support single-user exploratory programming. We also believe that this design will additionally support high-productivity collaborative production programming environments.

2.3 Fields

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

```
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 which provide the only access to the variable:

```
method x(v@obj) { <v.x> } -- the get accessor method
method set_x(v@obj, value) { <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; it does not return a result. 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 2.4 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 inherits ordered_collection;
method is_empty(l@list) { l.length = 0 }
method length(@list) { abstract } -- length must be defined in all concrete children
method prepend(x, l@list) { -- dispatch on second argument
   object inherits cons { head := x, tail := l } }

object nil inherits 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 inherits list; -- non-empty lists
field head(@cons); -- defines head(@cons) and set_head(@cons, ) accessor methods
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 full syntax of field declarations, excluding static typing aspects, is as follows:

```
field_decl     ::= [field_privacy] field_kind “field” name “(” formal “)”
                        field_body
field_kind     ::= empty | “shared” | “read_only” | “init_only”
field_body     ::= [initializer] “;” | “{” “abstract” “}” [“;”]
```

Encapsulation and field privacy is explained in section 2.4. The remaining parts of a field declaration are explained below.

2.3.1 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;
field vertices(@polygon);
method draw(p@polygon, d@output_device) {
   (-- draw the polygon on an output device, accessing vertices --) }
```

```
object rectangle inherits polygon;
field top(@rectangle);
field bottom(@rectangle);
field left(@rectangle);
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), 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 [Apple 92], a descendant of CLOS, 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.

2.3.2 Copy-Down vs. Shared Fields

By default, each object inheriting a field declaration receives its own space for the variable, and the field's accessor methods access the variable 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 copy of the memory location. 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, 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]. 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 which 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.

2.3.3 Field Initialization

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

```
field_inits    ::=  "{" field_init { "," field_init } "}"
field_init     ::=  name [location] initializer
```

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

```
   method prepend(e, l@list) { object inherits 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 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 2.6.

Fields may be declared to be immutable. A field declaration may be prefixed with the `read_only` keyword, indicating that no set accessor method is to be generated; a read-only field thus acts like a constant. An initializing expression for a read-only field must be supplied, and no object-specific initialization of a read-only field is allowed. The `shared` annotation on a read-only field would be redundant and is disallowed. Alternatively, a copy-down field declaration may be prefixed with the `init_only` keyword, also indicating that no set accessor method is to be generated but allowing new object-specific values to be specified for the field when a child object is created that inherits the field. The `shared` annotation conflicts with object-specific values, and so is disallowed for initialize-only fields. It is intended that the `:=` symbol be used to initialize a read/write field and the `=` symbol be used to initialize an immutable field, but this convention is not enforced by the language.

Many languages, including SELF and Eiffel, support distinguishing between assignable and constant variables, but few other imperative languages support initialize-only variables. Our experience with SELF leads us to believe that initialize-only fields would be quite useful; their documentation aspects alone will make programs clearer. Initialize-only fields support a functional (i.e., side-effect-free) programming style as well. 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` function can always modify a slot even if the set accessor is not generated.

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 3.8.

In Cecil, a field initialization expression is not evaluated until the field is first accessed. This supports functionality similar to `once` functions in Eiffel and other languages. It also avoids 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, i.e., no cyclic dependencies among field initializers are allowed.

Unfortunately, copy-down fields interact in some undesirable ways with lazily-evaluated field initializers. The current Cecil semantics is that the field initialization expression is evaluated at most once, with any inheriting objects sharing the same initial value. Since field initializers are evaluated lazily on demand, any inheriting object accessing the initial value of the field causes the initializer to be evaluated, and all other objects that have the default field value then use the initializer’s result expression. This semantics is similar to CLOS’s `:init-value` specifier. An alternative semantics under consideration would evaluate the initializing expression for each inheriting object; this would correspond roughly to CLOS’s `:init-function` specifier. A third alternative would use the current field value of the parent object(s) rather than the expression specified as part of the field declaration as the default initial value of the field in a child. Conflicts of field values from multiple parents could be reported. It is not clear, however, how to combine this semantics with lazy field initializer evaluation. For example, if two parents have field initializer expressions, when are they evaluated to detect whether they conflict? The exact semantics of field initialization is still being investigated and refined.

2.3.4 Abstract Fields

A field whose body is marked abstract is an abstract field. An abstract field is merely syntactic sugar for one or two (depending on whether or not the field is declared read-only or initialize-only) abstract method declarations corresponding to the field's get and set accessors. A concrete child object may implement an abstract field with a real field, one or two real methods, or some combination of the two. The `shared` annotation on an abstract field has no effect.

2.4 Encapsulation

To support the implementation of abstract data types, many languages include some mechanism whereby the external interface to an abstraction can be clearly defined and other internal implementation details can be hidden behind the abstraction boundary. This kind of encapsulation provides important benefits:

- Programmers can identify easily which operations are intended to be invoked by external clients. Encapsulation is a form of machine-checkable documentation.
- Implementors of an abstraction can isolate implementation choices that might change later, hiding these choices behind the abstraction boundary and thereby ensuring that the choices can be changed without affecting external clients. The implementor can easily identify those methods that might depend on the internal details and so may need to change.

Cecil allows fields and methods optionally to be prefixed with a privacy declaration of the following form:

```
privacy          ::= “public” | “private”
```

Those fields and methods forming the external interface of the abstraction can be identified by prefixing their declarations with the `public` keyword, while the internal implementation details of an abstraction can be hidden by prefixing internal fields and methods with the `private` keyword. Unannotated fields methods are treated by method lookup as if they were public, but with the caveat that the programmer has not made a commitment to continue to support the operation. Unannotated fields and methods would likely be common during exploratory programming, with privacy declarations added incrementally as the external interface to an abstraction becomes more refined. SELF pioneered this three-valued encapsulation mechanism.

Intuitively, a private method is internal to the data abstraction implementation(s) of which it is a part, and only other methods also within the same implementation(s) can invoke the private method (more precise semantics will be presented in the following subsections). For example, the earlier non-empty list abstraction might be rewritten to encapsulate its representation:

```
   object cons inherits list;
```

```
private field head(@cons);
private field tail(@cons);
public method length(c@cons) { 1 + length(c.tail) }
public method do(c@cons, block) {
   eval(block, c.head);
   do(c.tail, block); }
public method pair_do(c1@cons, c2@cons, block) {
   eval(block, c1.head, c2.head);
   pair_do(c1.tail, c2.tail, block); }
```

The `head`, `set_head`, `tail`, and `set_tail` accessor methods would be hidden from public view. These four methods are internal to the `cons` implementation, and so only other methods that also are part of the `cons` implementation could access these private operations.

For mutable fields which have two accessor methods, Cecil allows each accessor method to be given a different visibility, just as in SELF and Trellis, using the following extended syntax:

```
field_privacy  ::= privacy [ (“get” | “put”) [ privacy (“get” | “put”) ] ]
```

For example, if the `head` and `tail` operations were to be publicly-accessible but the `set_head` and `set_tail` operations were to be protected, the programmer could write those two field declarations as follows:

```
public get private put field head(@cons);
public get private put field tail(@cons);
```

Since this notation is somewhat cumbersome and verbose, alternative more concise notations are being explored. SELF certainly has a concise notation (the above privacy status would be described using the ^_ combination), but some find it too obscure. Some happy medium would be nice.

Note that a public field accessor method does not reveal that it is implemented as a field, nor does a public method reveal that it is implemented as a method. The programmer may always reimplement a public operation using an alternative implementation strategy without affecting clients. This is in contrast to some languages such as C++ where public data members reveal their implementation.

2.4.1 Granting Privileged Access

A sending method *S* is granted access to a private method *M* only if *S* is considered part of the implementation of which *M* is also a part. To specify the semantics of this rule precisely, the notion of “part of an implementation” must be defined. A method is considered part of the implementation of an object if at least one of its formals is specialized on the object. Additionally, since children should be allowed to invoke private methods which they inherit from their ancestors, a method is considered part of the implementation of an object if at least one of its formals is specialized on an ancestor of the object. A more subtle situation involves a child that needs to override a private method inherited from an ancestor while still allowing other methods inherited from the ancestor to invoke the overridden private method. To support this case, a method is considered part of the implementation of an object if at least one of its formals is specialized on a descendant of the object. In no other case is a method considered part of the implementation of an object. To summarize, a method is considered part of the implementation of an object if and only if the object is equal to, a descendant of, or an ancestor of one of the method's argument specializers.

The third case above may appear to grant nearly unlimited access to methods defined at the roots of the inheritance graph, thus greatly weakening encapsulation. However, in the presence of static type checking, methods defined near the roots of the inheritance graph will be limited to the set of messages that have been defined at the roots of the hierarchy, at least as abstract methods. Thus such methods will have a large number of objects to which they receive privileged access, but there won't be many messages that they will legally be able to send to these objects. During exploratory mode, no such controls are in place. If this becomes a problem, the encapsulation rule could be strengthened to verify that whenever a private method defined in a descendant is invoked from a method defined in an object, that the descendant's private method is overriding a private method defined in the object or one of its ancestors. This extra check would be compatible with the static type check and would correspond to the original reason privileged access is granted to children.

A method that dispatches on more than one argument is considered part of the implementation of all dispatched arguments, and so is granted privileged access to each of them. Granting privileged access to each of a method's dispatched arguments is often necessary from a practical standpoint. When writing a multi-method that is intended for arguments implemented in certain ways (indicated by the formal arguments' specializers), it is natural and often required to invoke operations internal to those arguments' implementations. For example, the `pair_do` method introduced earlier needs privileged access to its first two arguments in order to iterate through them in parallel:

```
public method pair_do(c1@cons, c2@cons, block) {
   eval(block, c1.head, c2.head);
   pair_do(c1.tail, c2.tail, block); }
```

Granting privileged access to operations that dispatch on the object also is reasonable from the standpoint of an implementor of an abstract data type. Encapsulation is useful in part to limit the potential dependencies on internal implementation details which might change, so that these potential dependencies can be found and updated whenever the implementation changes. Multi-methods with argument specializers can be just as easy to locate as are normal singly-dispatched methods, particularly in conjunction with programming environments that link multi-methods closely to their argument specializers. In the above list example, if the representation of non-empty lists is to be changed, it would be an easy matter to identify those methods that might need to be updated.

Encapsulation should be a static property verifiable from the program text, not a property verified dynamically as the program runs. This is to ensure that encapsulation will help in statically identifying those methods that might need to be updated as a result of a change to an internal implementation feature. Consequently, a sending method *S* is granted access to a private method *M* only if the specialized formals that enable privileged access are passed as arguments to the message; the actual names of the formals must be used, not just expressions that evaluate to the arguments. Thus, the collection of objects that the sending method is considered a part of is restricted by a particular message send to only those objects that the sender can be statically guaranteed to share with the invoked private method, namely specialized formal arguments. For example, the following method would be denied access to the `set_head` method even if `x` evaluates at run-time to another cons cell:

```
method copy_head(c@cons, x) {
   set_head(x, c.head); }
```

The restriction of allowing privileged access only to specialized formal parameters is checked by the static type system. It may turn out to be inappropriate in the dynamically-typed core, however, in which case the encapsulation rules may be weakened.

2.4.2 Private Multi-Methods

If a multi-method is marked private, then only callers who are part of the implementations of *all* of the private multi-method's argument specializers can access it. For example, the implementation of `display` for filled polygons on a fancy graphics device might rely on an internal private method:

```
public method display(shape@filled_polygon,
                      device@fancy_graphics_hardware) {
   set_up_display(shape, device);
   (-- rest of code --) }
private method set_up_display(shape@filled_polygon,
                              device@fancy_graphics_hardware) {
   (-- initialize graphics hardware for filled polygon displays --) }
```

The `set_up_display` method for filled polygons and fancy hardware is declared private, but the `display` method for filled polygons and fancy graphics hardware is granted access since it is a part of both the filled polygon implementation and the fancy hardware implementation. However, some other programmer might write the following method:

```
public method draw(shape@filled_polygon) {
   set_up_display(shape, standard_output());
   (-- more code --) }
```

This second method will be denied access, since the `draw` method is part of only one of the implementations that the `set_up_display` method belongs to. Since access privilege is determined statically, the `draw` method will be denied access even if at run-time `standard_output()` returns a child of `fancy_graphics_hardware`.

An alternative strategy would grant privileged access to a multi-method as long as the calling method and the called method had at least one argument specializer object in common. We have rejected this looser version of encapsulation since it would grant access too freely for our tastes. Since we are not aware of any other multiple dispatching language that attempts to support encapsulation of objects in this way, this encapsulation design should be considered an experiment whose outcome depends on experience gained from writing programs under this encapsulation model.

2.4.3 Other Issues with Encapsulation

With Cecil's current encapsulation rules, it is possible for an external client to gain privileged access to an object simply by defining a new multi-method, one of whose formal arguments is specialized on the target object. The new method would be considered part of the specializing object's implementation and so receive privileged access. If this "open-endedness" of objects turns out to be a problem, we may extend Cecil to allow an object declaration to specify those multi-methods which are granted privileged access. Other multi-methods would be allowed to dispatch on the object, but would not be able to access its private operations.

Alternatively, some methods which do not dispatch on an object may need to be granted private access. For example, an object creation method might need to allocate a new object and then invoke some sort of private initialization functions on the new object (field initialization as part of object creation may not always be sufficient or convenient for properly initializing the created object). The creation method should be granted privileged access even though it does not take an argument specialized on the necessary object. Conversely, an object may wish to require that all child objects are created by invoking one of the object's designated creation methods, making it illegal for unprivileged clients to create new children through object constructor expressions. Neither of these facilities is presently available in Cecil.

In Cecil, a public method cannot be overridden with a private method. Doing so might cause privilege violation errors for messages that appear to have an applicable public method. SELF uses an alternative rule which allows a private method to override a public method. A send which does not have privileged access simply ignores the private method, invoking the public method instead. While at first this semantics appears reasonable (inaccessible private methods are treated simply as being invisible to the caller), our experience has been that the difference between an accessible and an inaccessible send is too fine and too easy to change accidentally, leading to mysterious and *silent* changes in behavior as the target method of the send changes. C++ uses a rule similar to Cecil's, checking privilege after method lookup is complete.

Ideally, a private method should be able to override another private method. In particular, the child's private method should be allowed to dispatch on more arguments than the ancestor's private method. However, given the rules stated before, it must be possible to guarantee privileged access based on the static form of the message, using only the positions of specialized formals in the caller's argument list to verify that the message should be granted access to the private method. This static check rule could disallow privileged access to the overriding private method (since it could be private to a more restricted group of arguments than the caller is a part of), while the caller could have access to the overridden private method (since it could have fewer specialized formals). Consequently, the current combination of encapsulation rules appears to disallow a private method to override another private method with fewer specialized arguments. Clearly, this is undesirable. We are exploring several alternatives, such as relaxing the static checking part of encapsulation to determine access privileges dynamically or modifying the static checking rule to allow access to child private methods as long as access would be granted to an overridden private method, and we are likely to modify Cecil's encapsulation rules in some way in the future.

At present, one large global name space holds all object, method, and field declarations that are not nested within a method. In the future, Cecil will be extended with some sort of module system to divide up the single global name space. Modules might end up subsuming Cecil's existing encapsulation mechanism, resolving many of these outstanding issues in the process. We are actively exploring module designs.

2.4.4 Comparison with Other Languages

Cecil's encapsulation design was derived by extending the encapsulation rules common in single-dispatching object-oriented languages to handle multiple dispatching. For example, most single-dispatching languages grant privileged access only to messages sent to `self`. In Cecil terminology, a singly-dispatched method is one where the first formal argument, named `self`, is specialized on the object containing the method definition, and the traditional encapsulation rule then can be expressed as granting a message privileged access to a private method if the first argument to the message is the specialized formal argument of the sending message. Since in full Cecil any of the formal arguments may be specialized, i.e., there may be multiple `self`-like formal arguments, a multi-method is granted access to a private method if the message includes a `self`-like argument for each specialized formal parameter of the called private method.

Cecil's encapsulation rules are simpler in some ways than the corresponding rules for SELF [Chambers *et al.* 91]. Unlike Cecil, SELF grants privileged access to an object as long as the sending method is in an ancestor of the receiving object, with this relationship determined dynamically each time the message is sent. SELF's mechanism is intended to support module-based encapsulation in a classless language, where a method in a "class" (a traits object in SELF) has privileged access to all of its instances simultaneously, independent of which of its instances is currently the receiver. Cecil, like Trellis and Eiffel, supports a more restricted object-based encapsulation model, where a method has privileged access only to `self`-like arguments. For example, in SELF, a method in a traits object can access the private operations of an argument, as long as the argument turns out to inherit from the same traits object. This is particularly useful for methods like `=` which want to compare the private components of the receiver and the argument. Of course, should the argument turn out not to inherit from the traits object, some sort of run-time error is likely to result. In Cecil, the programmer instead exploits multiple dispatching to grant a method privileged access to multiple arguments in a safe way. In SELF, a copy method can assign to the private instance variables of the result of the `clone` message sent to `self`, since the clone will inherit from the traits object containing the copy method. Cecil supports direct initialization of fields as part of object constructor expressions.

Other languages support alternative encapsulation mechanisms. Several languages, including C++ and Trellis, distinguish between `private`, meaning access restricted to methods in the same class, and `protected` or `subtype_visible`, meaning also accessible to subclasses. Additionally, C++ supports the concept of a `friend` class or method which can be granted privileged access independent of its relation to the called method in the inheritance graph. The current Cecil design supports only protected-style encapsulation (called private in Cecil): parents are not encapsulated from their children, and subgroups of objects cannot form their own encapsulation module. This area merits further investigation.

Other languages supporting multiple dispatching do not appear to provide any encapsulation at the object or class level. CLOS provides packages, but these primarily organize the program's name space and are coarser and more limited than object-level encapsulation as in Cecil. For example, it is impossible (or at least awkward) in CLOS to define a private generic function and then write several multi-method versions of this generic function, where each version is placed in the file where the rest of the appropriate abstract data type implementation is written.

2.5 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.

2.5.1 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.

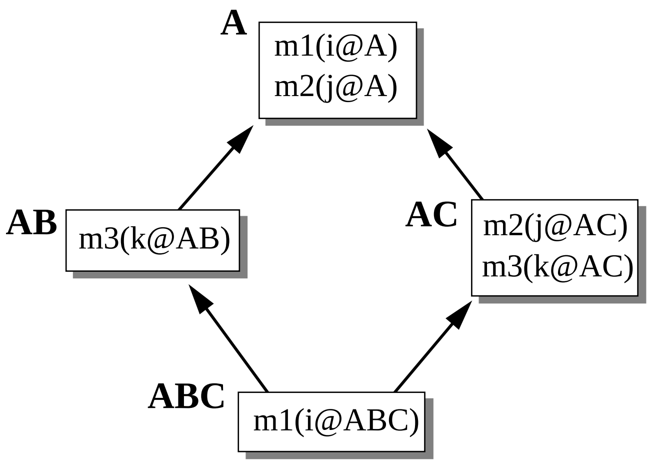
2.5.2 Semantics

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 is less than (more specific than) another in the partial order if and only if the first object is a proper descendant of the second object. 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 is less than (more specific than) another if and only if each of the argument specializers of the first method is equal to or less than (more specific than) the corresponding argument specializer of the second method. Since two methods cannot have the same argument specializers, at least one argument specializer of the first method must be strictly less than (more specific than) the corresponding specializer of the second method. For the purposes of this rule, an unspecialized argument is considered specialized on the "top" 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. Abstract methods are ignored when constructing this partial ordering. 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. 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.

2.5.3 Examples

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



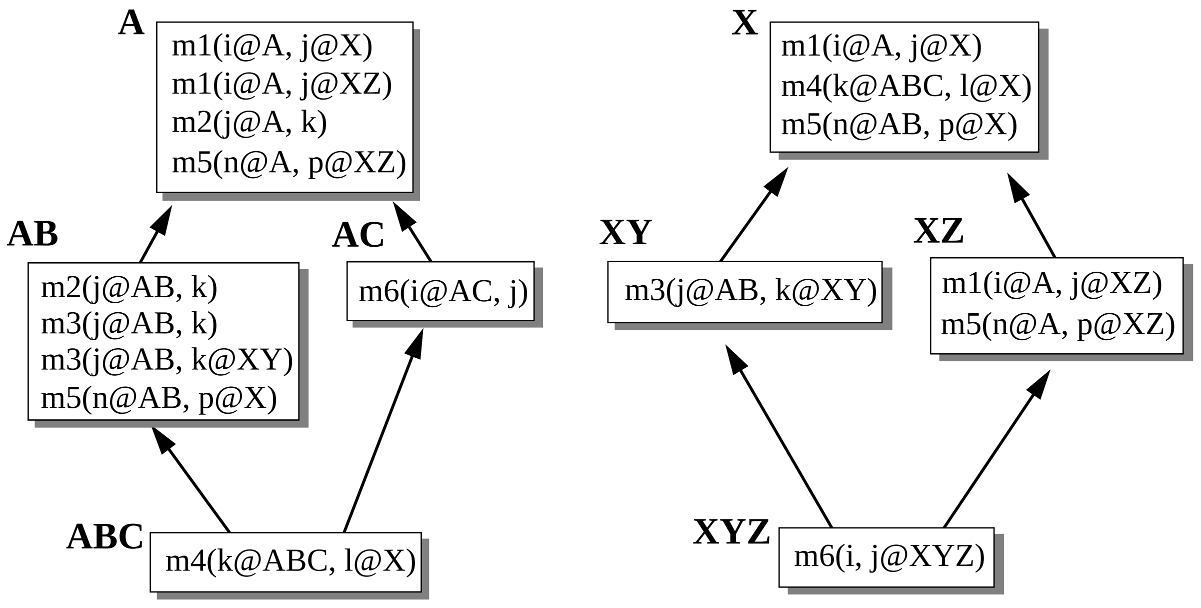
The partial ordering on objects in this graph defines ABC to be more specific than either AB or AC, and both AB and AC are more specific than A. Thus, methods defined for ABC will be more specific (will override) methods defined in A, AB, or AC, and methods defined in either AB or AC will be more specific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

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 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 2.6. 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

2.5.4 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 added enough

static type declarations to enable the system to detect the ambiguity at definition-time, then the warning will be delivered even earlier.

- 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.

2.5.5 Method Lookup and Encapsulation

Once method lookup has been performed, and a single target method selected, the system checks whether privileged access is needed and granted. If the target method is private, then privileged access is required to avoid an error. Access is granted if and only if for each specialized argument of the target private method, the corresponding actual argument is a specialized formal of the sending method, and the argument specializer of the formal in the sending method is equal to, an ancestor of, or a descendant of the argument specializer of the private method. If the target method is private but access is not granted, then the message results in a "message not privileged" error.

2.5.6 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 fields, 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 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 2.6) to access the contents of one or the other of the ambiguous fields.

2.5.7 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 will become more significant when some sort of module mechanism is added to Cecil.

The view of lexically-nested methods in Cecil is that nested methods extend the inheritance graph defined in the enclosing scope, rather than replace 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 from the enclosing scope plus any methods defined in the local scope. 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 respect primarily 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.

2.5.8 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, actuals are assigned to formals, 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 accessor method is executed), and the actual result of the method (if any) is returned to the caller. If the method does not return a result, but the caller expects a result, then the invocation terminates with a "result required" error.

2.6 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 over another.

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 | name "@" object
```

To invoke an overridden method, the normal 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 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 inherits rectangle;
field color(@colored_rectangle);
method display(r@colored_rectangle, d@output_device) {
   d.color := color; -- set the right color for this rectangle; sugar for set_color(d, color)
   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:*

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

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

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

This model of undirected and directed resends is a simplification of the current 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.

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.

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

2.7 Predicate Objects

Cecil has been extended experimentally to support *predicate objects* [Chambers 93]. 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 for better factoring of code. Also, predicate objects and multi-methods allow a pattern-matching style to be used to implement cooperating methods.

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.
- Parameterized objects, types, and methods support flexible forms of parametric polymorphism, complementing the inclusion polymorphism supported through subtyping.
- 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.

This section describes Cecil's static type system. 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, 3.5, and 3.6 describe the important kinds of declarations provided by programmers that extend the base dynamically-typed core language described in section 2. Sections 3.7, 3.8, 3.9, and 3.10 describe the type-checking algorithm for the language excluding parameterization, and section 3.11 extends the previous description to handle parameterization. Section 3.12 describes how the language supports mixed statically- and dynamically-typed code. Section 3.13 concludes with a discussion of some open issues and future work.

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,” “abstract method invoked,” 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 will rely on advanced implementation techniques such as those used for the dynamically-typed language SELF [Chambers & Ungar 91, Hölzle *et al.* 91b, Chambers 92a].

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 strongly influenced by certain language features. Foremost of these is multi-methods, which in Cecil are unbiased, in that no argument position is more important than any other. 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 unconstrained. 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 which *conform* to the type must support the type’s interface and promise to satisfy the behavioral specification. One type may claim to be a *subtype* of another, in which case all objects which 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 below, additional type constructors plus a few special types expand the type graph to a full lattice.

A *signature* in Cecil is an abstraction of a method, specifying both an interface (a name, a sequence of argument types, and a result type) and an implied but uncheckable behavioral specification. A set of signatures forms the interface of a type.

For example, the following types and signatures describe the interface to lists of integers (this notation is not Cecil syntax, as indicated by the italicized keywords):

```
type list subtypes collection;
signature is_empty(list):bool;
signature length(list):int;
signature do(list, &(int):void):void;
signature pair_do(list, list, &(int,int):void):void;
signature prepend(int, list):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 both *complete* and *consistent*:

- 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 3.7.2.

In a singly-dispatched language, each type has an associated set of signatures that define 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 specializers.

3.3 Type Declarations

Variable declarations and formal arguments and results of methods, closures, and fields may be annotated with type declarations, as in the following examples:

```
   method is_empty(l@:list):bool { l.length = 0 }
   method length(l@:list):int { abstract }
   method do(l@:list, closure:&(int):void):void { abstract; }
   method pair_do(l1@:list, l2@:list, closure:&(int,int):void):void {
      abstract; }
   method prepend(x:int, l@:list):list {
      object inherits cons { head := x, tail := l } }
   method copy_reverse(l:list):list {
      var l2:list := nil;
      do(l, &(x:int):void{ l2 := prepend(x, l2); };
      l2 }
   field head(@:cons):int;
   field tail(@:cons):list;
```

The syntax for types, excluding features relating to parameterized types, is as follows:

```
type           ::=  name
               |    "&" "(" [types] ")" [type_decl]
               |    type "|" type
               |    type "&" type
               |    "(" type ")"
types          ::=  type { "," type }
```

A type beginning with a & symbol is a closure type. Types created with the | or the & symbols are least-upper-bound or greatest-lower-bound types, respectively. These three kinds of types are described in more detail in subsection 3.5.

The following syntactic forms are extended to allow type declarations:

```
var_decl       ::=  "var" name [type_decl] initializer ";"
function       ::=  "(" [formals] ")" [type_decl] function_body
field_decl     ::=  [field_privacy] field_kind "field" name
                        "(" formal ")" [type_decl] field_body
specializer    ::=  [location] [type_decl]
               |    "@" ":" object
type_decl      ::=  ":" type
```

A type declaration on a variable or a formal restricts the contents of the variable or formal to objects that conform to the specified type. If the formal parameter is both specialized to some object and declared to be of some type, the specializing object must conform to the declared type. The notation `@:`*name* is syntactic sugar for `@`*name*`:`*name*. A type declaration after the formal parameter list of a method declaration or closure constructor expression restricts the result of the method or closure to objects that conform to the specified type. A type declaration after the formal parameter list of a field declaration restricts the contents of the field to objects that conform to the specified type; the result of the generated get accessor method and the second argument of the generated set accessor method are the same as the type of the field contents.

3.4 Extracting Types and Signatures

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. 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 specifies both graphs simultaneously with a single set of object and method declarations. An object declaration specifies both a new object in the object inheritance graph and a new type in the type lattice. Similarly, a method implies the existence of a corresponding signature. 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.

3.4.1 Extracting Types from Object Declarations

The syntax of object declarations and object constructor expressions is extended to support specification of the inheritance and the subtyping graphs as follows:

```
object_decl    ::= [role] object_or_type name {relation} [ field_inits ] “;”
object_expr    ::= [role] object_or_type {relation} [ field_inits ]
object_or_type ::= “object” | “type”
relation       ::= “isa” parents
                |    “inherits” parents
                |    “subtypes” types
```

A single object declaration constructs both a node in the inheritance graph and a node in the type graph. Both nodes have the same name, but no ambiguity is possible, since the variable name space and the type name space are distinct and uses of a name reference a particular name space. The new node in the inheritance graph is a direct child of the objects named in the `inherits` and the `isa` clauses, if any. The new node in the type graph is a direct subtype of each of the types named in the `subtypes` and `isa` clauses. Finally, the new object is declared to conform to the new type. Object constructor expressions similarly generate both objects and types, both of which are anonymous. It is illegal to create a cycle in the subtyping graph. Subtyping is a reflexive and transitive relation, and if an object conforms to a type then it conforms to all of the type's supertypes.

Each of the names included in the `isa` clause is interpreted as both an object (when constructing the inheritance graph) and as a type (when constructing the type lattice), and so is syntactic sugar for including each name in both the `inherits` clause and the `subtypes` clause (with a small caveat described below). The `isa` sugar is 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 `isa` declarations will be used; `inherits` and `subtypes` declarations are intended for relatively rare cases where finer control over inheritance and subtyping are required.

Some objects may not be intended to correspond to first-class types used in variable declarations and the like. For example, neither the `nil` nor `cons` objects should be considered types, but the abstract `list` object should. To support this distinction, an object declaration that is intended to generate a first-class namable type should replace the `object` keyword with the `type` keyword. The following declarations illustrate this distinction:

```
type list isa collection;
object nil isa list;
object cons isa list;
```

Only `list` may be used to declare the type of a variable.

Both `object` and `type` declarations create both an object and a type, but the type created as part of an `object` declaration is an internal type that cannot normally be named by the program. All references to types by name, such as in type declarations and `subtypes` lists, must name types created with the `type` declaration. The exceptions to this rule are the type of a specialized formal parameter, which may name an internal type to which the specializing object conforms, and names included as part of an `isa` declaration, which may reference any named object whether or not the object was declared with an `object` or `type` declaration.

3.4.2 Extracting Signatures from Method Declarations

As described above, the formal parameters and results of method and field declarations may be annotated with type declarations. In addition to specifying the operations required of legal arguments and the operations provided by all results, these type declarations are used to extract signatures automatically from method and field declarations. A method declaration of the form:

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

where any of the @obj_i may be omitted or shortened to `@:`$type_i$, implies a new signature of the form:

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

A field declaration of the form:

field *name*(x_1@*obj*:*type*):$type_R$ ...;

implies new signatures of the form:

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

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

where the second signature is omitted if the field is immutable.

3.4.3 Discussion

At present, Cecil does not provide programmers the ability to define types or signatures separately from objects and methods. If only a type or signature is needed, then an `abstract type` object declaration or an `abstract` method will suffice. Even with an abstract type, however, default implementations for some of the operations are often convenient to provide along with the definition of the type. These default implementations are easy to provide in Cecil by always generating a corresponding object with the type and using the object as the repository for the default implementations. In other languages, such as Axiom (formerly Scratchpad II) [Watt *et al.*, Jenks & Sutor 92], default implementations are stored with the type (called the *category* in Axiom).

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/Owl 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 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.

3.5 Special Types and Type Constructors

The Cecil type system includes four special pre-defined 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 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 `no_return` is implicitly a subtype of all other types, thus defines the bottom of the type lattice. It is the result type of closures with non-local returns, which never return to their caller. It also is the type of the primitive loop method, which never returns normally. No object has type `no_return`, and `no_return` may only be used when declaring the result type of a method or closure.

- The type `dynamic` is used where no other type is suitable. Wherever type declarations are omitted, `dynamic` is implied. The `dynamic` type selectively disables static type checking, in support of exploratory programming, as described in section 3.12.

Additionally, Cecil include three type constructors:

- The type of a closure taking N arguments (in addition to the closure object itself) is notated as `&(`$type_1$`,`...`,`$type_N$`):`$type_R$. This closure type would be extracted from a closure constructor expression of the form `&(`x_1`:`$type_1$`,`...`,`x_N`:`$type_N$`):`$type_R$`{`...`}`. For each closure expression or closure type appearing in the program, the system extracts a corresponding signature of the form ***`signature`*** `eval(&(`$type_1$`,`...`,`$type_N$`):`$type_R$`,` $type_1$`,`...`,`$type_N$`):`$type_R$, abstracting the closure's `eval` method. Closure types are related by implicit subtyping rules that reflect standard contravariant subtyping: a closure type of the form `&(`t_1`,`...`,`t_N`):`t_R is a subtype of a closure type of the form `&(`s_1`,`...`,`s_N`):`s_R if and only if each t_i is a supertype of the corresponding s_i and t_R is a subtype of s_R.
- The least upper bound of two types in the type lattice is notated as $type_1$ `|` $type_2$. Such a type 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 later in section 3.11.
- The greatest lower bound of two types is notated as $type_1$ & $type_2$. Such a type is a subtype of both $type_1$ and $type_2$, and a supertype of all types that are subtypes of both $type_1$ and $type_2$. The greatest-lower-bound type constructor has higher precedence than the least-upper-bound type constructor.

These special types and type constructors extend the explicitly-declared type partial order generated from object declarations to a full lattice.

Unlike most other languages, including C++ and Eiffel, Cecil has no built-in `nil` object that can be used in place of any object. Such a nil object would introduce the potential for nil pointer dereferences. Instead, programmers can define their own application-specific "nils" with appropriate behavior and fitting into the application's type hierarchy. The `nil` object defined above as part of the `list` abstraction is an example of an application-specific nil object. Accordingly, no special subtyping rules are needed to handle a built-in `nil` object.

3.6 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, are manipulated at run-time and can appear as arguments to messages at run-time. In contrast, other 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. The part played by an object can be documented by prefixing an object declaration or object constructor expression with an object role annotation:

```
role          ::= "abstract" | "template" | "concrete"
```

Each of these role annotations appears in the list hierarchy:

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

Abstract objects are potentially incomplete objects designed to be inherited from and fleshed out by other objects. Abstract objects can have abstract methods associated with them and can have uninitialized fields. In return, abstract objects cannot be manipulated at run-time nor inherited from directly in an object constructor expression. The type checker verifies that any objects referenced in an expression or listed as a parent in an object constructor expression are not abstract objects. Abstract objects are analogous to abstract classes in class-based languages.

Template objects are complete objects suitable for direct "instantiation" by object constructor expressions. Any abstract methods inherited by a template object must be overridden by concrete implementations, but fields may remain uninitialized. In return, template objects cannot be named in an expression, but they can be listed as a parent in an object constructor expression. Template objects are analogous to concrete classes in class-based languages.

Concrete objects are complete, initialized objects that can be manipulated at run-time. Accordingly, no abstract methods can remain for a concrete object and all fields inherited by the object must have been initialized, either at the point of declaration or as part of a field initialization expression. Like other objects, named concrete objects can be inherited from as well.

If the object role annotation is omitted, the object is considered fully manipulable by programs but no checks for abstract methods or uninitialized fields are performed. Unannotated objects might be common in exploratory code.

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 express the degree of completeness of the object. To be manipulated at run-time, an object must be fully complete, with no abstract methods and no uninitialized fields. Since these requirements may be too restrictive for objects used solely as part of the program structure, abstract and template annotations mark objects as potentially incomplete. The eased restrictions on completeness comes at the cost of increased restrictions on how the object can be used.

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. (Abstract methods are described in section 2.2.4).

In an earlier version of Cecil, a fourth 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 concrete 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 or interfaces to be inherited. Unique objects were removed because it was felt that the extra language mechanism was not worthwhile. Similarly, the `template` annotation may be removed for a similar reason, since it is not strictly necessary for the type checker. The distinction between abstract objects and concrete objects, however, is crucial to be able to write realistic Cecil code.

3.7 Type Checking Messages

This section describes Cecil's type checking rules for message sends. Section 3.8 describes type checking for other kinds of expressions. Parameterized types are described in subsection 3.11.

In Cecil, all control structures, instance variable accesses, and basic operators are implemented via message passing, so messages is the primary kind of expression to type check. For a message to be type-correct, there must be a single most specific applicable non-abstract method 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 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 completeness and consistency independent of whether enough 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 it may report more informative error messages.

Subsection 3.7.1 describes checking messages against signatures, and subsection 3.7.2 describes checking signatures against implementing methods.

3.7.1 Checking Messages Against Signatures

Given a message of the form *name*`(`*expr*$_1$`, ..., `*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`, ..., `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 legal. 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`.

3.7.2 Checking Signatures Against Methods

The type checker ensures that, for every signature extracted from the program, all possible run-time messages declared type-safe by the signature in fact locate a single most specific method with appropriate argument and result type declarations. As mentioned earlier, this checking ensures that each signature is implemented completely and consistently. 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. Consequently, the Cecil type system is based in a straightforward but brute-force algorithm to checking implementations of signatures. Of course, real implementations of Cecil can and should optimize this brute force algorithm.

A signature represents a family of messages that might be sent at run-time. Each distinct kind of message is represented by a *message pattern*, comprised of a name and a sequence of argument *object patterns*. Each object pattern represents a family of run-time objects, all with identical inheritance structures and associated methods. An object pattern is derived from each static occurrence of an object declaration, an object constructor expression, or a closure constructor expression, as well as each of the four kinds of literal constants. Abstract objects are excluded from consideration, because they cannot be manipulated at run-time. Template objects are included, even though they also cannot be manipulated directly at run-time, because the programmer claimed that they were complete (with the possible exception of uninitialized fields) and signature checking will verify this claim. Object patterns are analogous to classes in class-based languages and maps in SELF.

For each signature, the type checker constructs all message patterns covered by the signature by enumerating all possible combinations of object patterns that conform to the corresponding argument types in the signature. For each message pattern, 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 “method not implemented” error. If only abstract methods are found, the type checker reports an “invoking abstract method” error. If multiple mutually ambiguous methods are found, the type checker reports a “message ambiguous” error. Otherwise, a single most specific method is found for the message. In this case, the type checker also verifies that the argument object patterns conform to the declared argument types of the located 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 pattern 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 patterns that conform to `collection` and all those that conform to `&(int,int):void`. For a small system, the collection-conforming object patterns might be the following:

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

The `list` and `collection` objects are not enumerated because they are `abstract`. The third pattern is extracted from the object constructor expression in the `prepend` method. The closure object patterns are derived from closure constructor expressions in the program with the appropriate types.

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

```
message_pattern pair_do(nil,nil,...);
message_pattern pair_do(nil,cons,...);
message_pattern pair_do(nil,object_pattern inherits cons,...);
message_pattern pair_do(nil,array,...);
message_pattern pair_do(cons,nil,...);
message_pattern pair_do(cons,cons,...);
...
message_pattern pair_do(array,object_pattern inherits cons,...);
message_pattern pair_do(array,array,...);
```

For each message pattern, 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.

3.7.3 Comparison with Type Systems for Singly-Dispatched Languages

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 nevertheless it does enforce the effect of contravariant redefinition for unspecialized arguments. When type-checking a signature, more than one method could be found as a result of the simulated method lookups. For each located method, the declared types of the arguments must be supertypes of the types of the argument object patterns in the message pattern. For an unspecialized argument, the type of the object pattern can be as general as that specified in the signature. For the method to correctly implement the message pattern, then, its unspecialized arguments must be declared to be supertypes of the corresponding types in the signature. Additionally, the result type of any method located by simulated method lookup must be a subtype of the result type specified in the signature. These constraints are exactly the contravariant rules, restricted to the unspecialized arguments.

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. As long as these objects conform to the declared argument type, type safety is preserved; if they do not, the error will be detected and reported when simulating lookup of that method and comparing declared argument types against the types of the object patterns. 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, it would be given special treatment and allowed to vary 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, then Cecil's type checking rules would reduce to those found in a traditional singly-dispatched language.

In Cecil, some checks of legal method implementation can be done incrementally, as each method is defined. For example, a method could be checked against all methods that it overrides, to ensure that its unspecialized arguments are supertypes of the corresponding types of the overridden methods and that its result type is a subtype of the result types of the overridden methods, much like in a singly-dispatched language. However, final checking still is needed once the whole program is assembled to ensure that no two multi-methods are mutually ambiguous.

3.7.4 Comparison with Type Systems for Multiply-Dispatched Languages

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. Both Polyglot and Kea check individual messages for consistent and complete implementation, rather than using signatures as does Cecil. This means that some errors can remain in the implementation of related methods, but they might not be detected because no message currently encounters the error.

3.8 Type Checking Expressions, Statements, and Declarations

Type checking an expression results in both an indication of whether the expression was type-correct, and if so the type of the result of the expression. Type checking a statement or a declaration simply checks for type correctness. All constructs are type-checked in a typing context which binds variable names to their current types and an indication of whether the variable is assignable or constant, and which binds object names to object patterns; a separate name-space binds type names to types. Declarations introduce new bindings into the typing context.

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 variable reference *name* is type-correct if and only if *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, and if an object then the bound object pattern is not an `abstract` or `template` object. The type of the result of a variable reference is the associated type in the typing context (i.e., the type specified in the variable's declaration).
- An object constructor expression of the general form

  role-annotation **`object`** **`inherits`** parent_1`, ..., `parent_K
  `subtypes` $\mathit{supertype}_1$`, ..., `$\mathit{supertype}_L$
  `isa` $\mathit{parent\text{-}and\text{-}supertype}_1$`, ..., `$\mathit{parent\text{-}and\text{-}supertype}_M$
  `{` field_1@obj_1 `:=` expr_1`, ..., `field_N@obj_N `:=` expr_N `}`

  is type-correct if and only if:

  - each parent_i name is bound to a non-abstract object in the typing context;
  - each $\mathit{supertype}_i$ is type-correct in the current typing context and not `void`;
  - each $\mathit{parent\text{-}and\text{-}supertype}_j$ name is bound to a non-abstract 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 neither `shared` nor `read_only`;
  - 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;
  - 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; and
  - *role-annotation* is neither `abstract` nor `template`.

  The `=` symbol may be used in place of the `:=` symbol in a field initialization clause without effect. The type of the result of an object constructor expression is the type extracted from the object constructor expression, as described in section 3.4.1.
- A closure constructor expression of the general form

  `&(`x_1@obj_1`:`type_1`, ..., `x_N@obj_N`:`type_N`):`type_R `{` *body* `}`

  is type-correct if and only if:

- the x_i, when provided, are distinct;
- each of the $type_i$ are type-correct in the current typing context and not `void`, with the extra allowance that if @obj_i is provided then $type_i$ may name an internal type (a type extracted from an `object` declaration as opposed to a `type` declaration);
- if @obj_i is present, then obj_i conforms to $type_i$;
- $type_R$ is type-correct;
- *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

`&(`$type_1$`, ..., `$type_N$`):`$type_R$`.`

Closure constructor expressions also generate corresponding `eval` signatures.

- An array constructor expression of the general form `[`$expr_1$`,...,`$expr_N$`]` type-correct if and only if each of the $expr_i$ is type-correct. If the static type of each $expr_i$ is T_i, then the type of the result of an array constructor expression is the predefined `array` parameterized type instantiated with the least upper bound of the T_i.
- A message expression of the general form *name*($expr_1$`, ..., `$expr_N$) is type-correct if and only if:
  - each of the $expr_i$ is type-correct, with static type T_i, and T_i is not `void`;[*] and
  - the maximal set $S = \{S_1, ..., S_M\}$ of applicable signatures is non-empty, where S is drawn from the set of signatures extracted from method declarations, field declarations, and closure types and constructor expressions, and each signature has the form S_i = ***`signature`*** *name*`(`t_{i1}`, ..., `t_{iN}`):`t_{iR} where T_i is a subtype of t_i.

  The type of the result of a message is the greatest lower bound of each of result types t_{iR} of the applicable signatures. Correctness of the implementation of signatures is checked separately.
- A resend expression of the general form

  `resend``(..., `x_i@$parent_i$`, ..., `$expr_j$`, ...)`

  is type-correct if and only if:
  - each of the arguments x_i or $expr_i$ is type-correct, with static type T_i, and T_i 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;
  - 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 @$parent_j$;
  - when method lookup is simulated with a message name the same as M and with N arguments, where argument i is either the top object (if $formal_i$ of M is unspecialized),

[*] 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`.

parent$_i$ (if the argument of the resend is directed using the `@`*parent*$_i$ suffix notation), or *object*$_i$ (otherwise), and where the resending method M is removed from the set of applicable methods, 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 if and only if *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 if and only if:
    - *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}.
- A declaration block is type-correct if and only if its declarations are type-correct, when evaluated in a typing context where all names in the declaration block are available throughout the declaration block.
- An expression statement is type-correct if and only if the expression is type-correct. The type of the result of the expression is ignored.

Result expressions, the optional last part of a method, closure, or parenthetical expression, are type-checked as follows:

- If no result expression is provided, then the result expression is type-correct and of type `void`.
- A normal result expression of the form *expr* is type-correct if and only if *expr* is type-correct, with static type T. The type of the result of a normal result expression is T.
- A non-local void return, of the form `^` or `^;`, is type-correct if and only if:
    - the expression is nested textually inside the body of a method M; and
    - the declared result type of M is `void`.

  The type of the (local) result of a non-local void return is `no_return`.
- A non-local return, of the form `^` *expr*, is type-correct if and only if:
    - *expr* is type-correct, with static type T;
    - the expression 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 `no_return`.

A normal body of a method, closure, or parenthetical expression is type-correct if and only if its statements are type-correct and its result expression is type-correct, with static type T. The type of the result of a body is the type T. An abstract body of a method is always type-correct.

The following rules define type-correctness of declarations:

- A variable declaration of the form **`var`** *name*`:`*type equal-sym expr* is type-correct if and only if:
    - *name* is not otherwise defined in the same scope;

  - *type* is type-correct and not `void`; and
  - *expr* is type-correct in a typing context where *name* is unbound, with static type *T*, and *T* is a subtype of *type*.

  Subsequent constructs are evaluated in a typing context where *name* is bound to the type *type* and is assignable if *equal-sym* is `:=` and is constant otherwise.

- An object declaration of the form

  ```
  role-annotation object 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 are allowed in `inherits` and `isa` clauses;
  - the `abstract` and `template` role annotations are allowed; and
  - no cycles are allowed in the inheritance and subtyping graphs.

  Subsequent constructs are evaluated in a typing context where *name* is bound in the variable name space to an object pattern extracted from the object declaration, and *name* is bound in the type name space to the associated type generated from the object declaration.

- An object extension declaration of the form

  ```
  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 if and only if:

  - *name* is bound in the typing context to an object pattern;
  - the object declaration created by extending the named object pattern with the parents, types, and field initializations in the `inherits`, `subtypes`, `isa`, and field initialization clauses would be type-correct; and
  - none of the $field_i@obj_i$ initialize fields already specialized on or inherited by the object pattern.

  Subsequent constructs are evaluated in a typing context where *name* is bound in the variable name space to the extended object pattern and *name* is bound in the type name space to the associated type generated from the extended object declaration.

- A method declaration of the general form

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

  is type-correct if and only if:

  - the x_i, when provided, are distinct;
  - each of the $type_i$ are type-correct in the current typing context and not `void`, with the extra allowance that if $@obj_i$ is provided then $type_i$ may name an internal type;
  - if $@obj_i$ is present, then obj_i conforms to $type_i$;
  - $type_R$ is type-correct;

  - *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 $type_R$.

  Method declarations also generate corresponding signatures.

- A field declaration of the general form

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

  is type-correct if and only if:

  - *type* is type-correct in the current typing context and not `void`, with the extra allowance that if `@`*obj* is provided then *type* may name an internal type;
  - if `@`*obj* is present, then *obj* conforms to *type*;
  - $type_R$ is type-correct;
  - if `:=` *expr* is provided, then *expr* is type-correct, with static type *T*, and *T* is a subtype of $type_R$; and
  - if *kind* is `shared` or `read_only`, then `:=` *expr* is provided.

  The `=` symbol may be used in place of the `:=` symbol without effect. Field declarations also generate corresponding signatures.

A type expression is type-correct if and only if all of the types referenced by name are bound in the type name space of the typing context, and none of these names references an internal type. In the above descriptions, a type expression is often used in place of a type, with the implication that the type used is the one constructed by computing the type specified by the type expression. Closure types also generate corresponding `eval` signatures.

3.9 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 3.7.2, is sufficient to detect and report the error, so no additional checking is required. When enumerating and checking message patterns 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.

3.10 Type Checking Encapsulation

The type system needs to check statically whether a message might invoke a private method, and if so, whether this access is allowed. Several extensions to the described type-checking rules are required to accomplish this. The type of an object is divided into two halves: a public type and a private type. An object declaration conforms to its private type, and the private type is a subtype of its public type. Publicly-accessible operations will be part of the public type, while private operations will be restricted to the private type.

References to types by name resolve to the public version. In two cases, references to objects by name also correspond to analogous references to types, and in these two cases the private type can be used. The private version of a type is used when extracting the corresponding type from an object reference in an `isa` clause. This implies that one private type is a subtype of another private type exactly when the object declaration defining the first private type references the object declaration defining the second private type in its `isa` clause. For example, consider the following object declarations:

```
object point;
object color_point isa point;
object polar_point subtypes point;
```

Each of the `point`, `color_point`, and `polar_point` declarations define an object, a public type, and a private type. Each object conforms to its private type, and each private type is a subtype of the corresponding public type. The public `color_point` and `polar_point` types are subtypes of the public `point` type, as expected. Additionally, the private `color_point` type is a subtype of the private `point` type (because the declaration was part of an `isa` clause). The private `polar_point` type is not a subtype of the private `point` type, however, since it only uses the `subtypes` clause and does not inherit any code from `point`.

The private type also comes into play when a private method has a `@:` formal parameter specializer. In this case, the signature extracted from the private method uses the private type for any parameters specialized with the `@:` notation, and uses the public type for all other formal parameters. For example, the following method:

```
private method set_x(p@:point, new_x:int):void { ... }
```

generates the following signature:

signature set_x($\mathtt{point}_{\mathtt{private}}$, $\mathtt{int}_{\mathtt{public}}$):$\mathtt{void}_{\mathtt{public}}$;

When type-checking the body of a method, the type of a formal argument declared using the `@:` specializer notation is the private version of its type; other variables use the public version of the named type as before. Because a private type is a subtype of the corresponding public type, a formal parameter specialized using `@:` has access to all the public methods of the type, but it also has access to the private methods of its type. For example, in the following method, the `set_x` message locates the above signature and so is type-correct:

```
method move_right(p@:point, delta:int):void {
   set_x(p, p.x + delta);
}
```

The variable `p` has type $\mathtt{point}_{\mathtt{private}}$ and the expression `p.x+delta` has type $\mathtt{int}_{\mathtt{public}}$, which are subtypes of the argument types in the signature.

To complete type-checking of encapsulation, type-checking of the implementations of signatures must be extended to handle the distinction between private and public types and to handle the case when a private method is located during simulated method lookup. Enumerating object patterns that conform to a type, whether public or private, is no different than before, given than an object conforms to its corresponding private type, which is a subtype of the corresponding public type.

More interesting is what to do if simulated method lookup determines that a message pattern would invoke a private method. In this case, the system needs to verify that the calling method is part of the same implementation as the target method, as described in section 2.4. The implementation that the target method is part of is simply the union of its argument specializers. The signature itself contains a lower bound on the set of implementations that the sending method is a part of, computed from the set of private types used as arguments in the signature. For each private argument type $T_{private}$ of the signature, a type-correct sender must be passing an argument known to be a subtype of $T_{private}$. The only expressions that would have this type are formal parameters specialized using the `@:` notation to an object O whose private type is a subtype of $T_{private}$. The only such objects are the object $O_{Tprivate}$ from which $T_{private}$ is derived and objects which inherit via `isa` clauses from $O_{Tprivate}$. Consequently, the sender can be considered to be a member of the implementation defined by the set of $O_{Tprivate}$ objects. Access to a private method is then checked by verifying that the set of $O_{Tprivate}$ objects includes ancestors or descendants of each of the argument specializers of the private target method.

3.11 Parameterized Objects, Types, and Methods

Practical statically-typed languages need some mechanism for parameterizing objects and methods by types. This is particularly important for “container classes” like `list` and `array`.

3.11.1 Explicit Parameterization

Cecil allows both object, method, and field declarations to be parameterized by a sequence of types, as the following examples illustrate:

```
abstract type collection[T];

abstract type list[T] isa collection[T];
concrete object nil[T] isa list[T];
template object cons[T] isa list[T];
field head[T](@:cons[T]):T;
field tail[T](@:cons[T]):list[T] := nil[T];
method prepend[T](h:T, t:list[T]):list[T] {
   object isa cons[T] { head := h, tail := t } }

abstract type table[Key,Value] isa collection[Value];

template type array[T] isa table[int,T];
method new_array[T](size:int, initial_value:T):array[T] {
   object isa array[T] { size := size, initial_value := initial_value } }


template type printable_array[T <= printable] isa array[T];
```

The syntax of object, method, and field declarations is extended to allow explicit parameterization as follows:

```
object_decl    ::= [role] object_or_type name [formal_params]
                       {relation} [ field_inits ] “;”
method_decl    ::= [privacy] “method” method_name [formal_params]
                       function [“;”]
field_decl     ::= [field_privacy] field_kind “field” name [formal_params]
                       “(” formal “)” [type_decl] field_body
formal_params  ::= “[” formal_param { “,” formal_param } “]”
formal_param   ::= [“‘”] name [ “<=” type ]
```

The formal type parameter of the form `‘`*name* is universally quantified over all non-`void` types, while a formal type parameter of the form `‘`*name* `<=` *type* is quantified over all types that are subtypes of *type*; *type* must be a non-`void` type. The upper bound of a formal type parameter is *type*, if the `<=` *type* suffix is provided, or `any`, otherwise. Similar facilities appear under the name of bounded quantification [Cardelli & Wegner 85] and constrained genericity [Meyer 86].

Type parameters are scoped over the whole object, method, or field declaration; type parameters must have distinct names. Within its scope, a type parameter may be used in a type declaration or as an instantiating type for some other parameterized type or method; a type parameter cannot be used in a `subtypes` clause, as this context requires a statically-known type. When used, a type parameter is treated as some unknown type that is a subtype of is upper bound type. Cycles are not allowed in the dependency graph of formal type parameters and their upper bound types, e.g., `[‘A <= B, ‘B <= A]` is illegal, but no other orderings are required, e.g., `[‘A <= B, ‘B <= int]` is legal, and the first occurrence of `B` refers to the instantiating value of the second type parameter.

To use a parameterized object or method, the client must first instantiate it with actual types for each of its parameters, at which point the instantiated object or method can be used as an equivalent unparameterized object or method. The syntax of object references, type references, messages, and object extension declarations is extended as follows to allow instantiating parameters to be provided:

```
object         ::= name [params]
type           ::= name [params] | ...
message        ::= name [params] “(” [exprs] “)” | ...
obj_extension  ::= name [params] {relation} “;”
params         ::= “[” types “]”
```

Legal instantiating type parameters must be subtypes of the upper bounds of the corresponding formal type parameters.

3.11.2 Implicit Parameterization

While explicit parameterization and instantiation is sufficient for programming parameterized objects and types, it is often inconvenient. For example, consider the implementation of an explicitly-parameterized `pair_do` method:

```
method pair_do[T1,T2](c1@:cons[T1], c2@:cons[T2],
                      closure:&(T1,T2):void):void {
   eval(closure, head[T1](c1), head[T2](c2));
   pair_do(tail[T1](c1), tail[T2](c2), closure);
}
```

Singly-dispatched languages do not face this verbosity, because methods are defined within a class and within the scope of the parameterized class's type parameters. Additionally, invocations of methods on a parameterized object, such as the `head` message above, would not need to specify an instantiating parameter because the it can be derived from the instantiating parameter of the distinguished receiver object.

To regain much of the conciseness of singly-dispatched methods in a parameterized class while still supporting multi-methods, Cecil allows *implicit type parameter bindings* to be present in the type declarations of formal arguments of a method or field. These implicit type parameters are instantiated automatically with the corresponding type of the actual argument in each call site. The syntax of types is extended to allow implicit type parameter bindings as follows:

```
type           ::= ... | “‘” name [“<=” type]
```

The operations on parameterized `cons` objects can be rewritten with implicit type parameters as follows:

```
   template object cons[T] isa list[T];
   field head(@:cons[‘T]):T;
   field tail(@:cons[‘T]):list[T] := nil[T];
   method pair_do(c1@:cons[‘T1], c2@:cons[‘T2],
                  closure:&(T1,T2):void):void {
      eval(closure, head(c1), head(c2));
      pair_do(tail(c1), tail(c2), closure);
   }
   method prepend(h:T, t:list[‘T]):list[T] {
      object isa cons[T] { head := h, tail := t } }
```

Implicit type parameter bindings can only appear as the declared type of a formal parameter or variable, as the upper bound type of another type parameter, or as the instantiating type of a parameterized type; all other occurrences of implicit type parameter bindings are illegal. Like explicit formal type parameters, an implicit formal type parameter may be bounded from above by some type using the `<=` *type* notation, and implicit formal parameters are quantified over all types that are subtypes of its upper bound (where `any` is used as the default upper bound). Like explicit type parameters, implicit type parameters are scoped over the entire declaration. Implicit type parameters must have names distinct from any explicit type parameters. Like explicit type parameters, implicit type parameters may be used in the type declarations of earlier formal arguments, as in the `prepend` method above, as long as no cyclic dependencies result. Implicit type parameters are akin to polymorphic type variables in languages like ML [Milner *et al.* 90].

Implicit type parameters are useful not only for parameterized types but also for performing simple calculations on argument types to compute appropriate result types. For example, the following method describes its result type in terms of its argument types:*

```
   method min(x1:‘T1 <= comparable, x2:‘T2 <= comparable):T1|T2 {
      if (x1 < x2, { x1 }, { x2 })}
```

* Section 3.13.4 will mention F-bounded polymorphism as an area of future work needed to make a general-purpose `comparable` type work well.

User-defined control structures often compute the types of their results from the types of their arguments:

```
method if(condition:bool,   true_case:&():‘T1, false_case:&():‘T2):T1|T2 {
   abstract }
method if(condition@:true,  true_case:&():‘T1, false_case:&():‘T2):T1|T2 {
   eval(true_case) }
method if(condition@:false, true_case:&():‘T1, false_case:&():‘T2):T1|T2 {
   eval(false_case) }
```

As illustrated by the above example, least-upper-bound types over implicit type parameters are relatively common. To avoid the need for many of these disjunctions, Cecil allows multiple implicit type parameter bindings of the same type name. This is semantically identical to multiple distinctly named implicit type parameter bindings and replacing all occurrences of the original type name with the disjunction of all the new type names. This mechanism allows the `min` example to be rewritten as follows:

```
method min(x1:‘T <= comparable, x2:‘T):T {
   if (x1 < x2, { x1 }, { x2 })}
```

The system automatically supplies the instantiating type for an invocation of an implicitly parameterized method. This instantiating type is derived at run-time for each call from the dynamic type of the corresponding argument. The declared formal parameter is treated as a form of type pattern that needs to be matched against the dynamic type of the actual parameter to bind the implicit type parameters. The most general form of the type declaration for a formal parameter is the following:

‘S <= *type*[‘T_1 <= $type_1$, ..., ‘T_N <= $type_N$]

If the ‘S <= prefix is omitted, then a fresh type variable is supplied to represent the dynamic type of the argument. If the *type*[...] is omitted, it defaults to `any`. Zero or more parameters may be provided for *type*. If any of the <= $type_i$ suffixes are omitted, they default to `any`. None of the *type* or $type_i$ can contain an implicit type parameter binding (i.e., no ‘ in them). Any of the ‘T_i <= prefixes may also be omitted.

3.11.3 Instantiating Implicit Type Parameters

When invoking a method with a formal parameter declared with a type of the above general form, the S type variable is bound to the dynamic type of the corresponding actual argument object. To bind any T_i type parameters, the dynamic type of the actual is matched against the type pattern of the form *type*[‘T_1 <= $type_1$, ..., ‘T_N <= $type_N$]. This matching is performed by searching the supertype lattice of the dynamic type for the single most specific type of the form *type*[t_1, ..., t_N], where each t_i matches the corresponding ‘T_i <= $type_i$ pattern using the following rules:

- if the ‘T_i <= prefix is omitted, then t_i must be the same as $type_i$;
- if the ‘T_i <= prefix is included, then t_i must be a subtype of $type_i$, and T_i is bound to t_i.

For example, consider the following code:

```
abstract type printable;
method print(@:printable):void { abstract; }
```

```
abstract type number isa printable;

abstract type collection[T];
method do(c@:collection['T], closure:&(T):void):void { abstract; }
method print(c@:collection['T <= printable]):void {
   "[ ".print;
   do(c, &(x:T){ x.print; " ".print; });
   "]".print;
}
method expand_tabs(c@:'T <= collection[char]):T {
   -- return a copy of c, where tab characters have been replaced with spaces
}

abstract type list[T] isa collection[T];
concrete object nil[T] isa list[T];
template object cons[T] isa list[T];

abstract type table[Key,Value] isa collection[Value];
abstract type indexed[T] isa table[int,T];
template type array[T] isa indexed[T];
template type string isa indexed[char];
```

If the message `print` is sent to an object of dynamic type `cons[number]`, then the `print` method defined on `collection` will be found. Then the dynamic type `cons[number]` will be matched against the pattern `collection['T <= printable]` to bind the implicit type parameter `T`. The supertype graph of `cons[number]` will be searched for a type of the form `collection`[*something*], where *something* is a subtype of `printable`. This search will locate the type `collection[number]`, and `T` will be bound to the type `number` for the duration of the execution of the `print` method.

If, on the other hand, the message `expand_tabs` is sent to an object of dynamic type `string`, the method defined for `collection[char]` will be found. The dynamic type `string` will be matched against the static formal argument type `'T <= collection[char]`. This match will succeed, since `string` is declared as a subtype of `collection[char]`, and the implicit type parameter `T` will be bound to `string`.

Note that the type declaration `collection[char]` is different than the type declaration `collection['T <= char]`. The former will match any collection that is declared to be a subtype of a collection of characters, i.e., that supports all the operations of collections of characters and is substitutable wherever a collection of characters appears. The latter type declaration matches any collection of items which are subtypes of characters. The type `collection[letter_char]` would match this latter type declaration, assuming that `letter_char` was a subtype of `char` restricted to letters, but it would not match the former type declaration, since a collection of letters is not a subtype of a collection of generic characters; in particular, the store operation for a collection of letters takes a more specific argument than does the store operation for collections of generic characters. Deciding the exact form of a parameterized type declaration can be rather subtle, and we need to gather experience with the language to assess how well programmers are able to pick an appropriate type declaration.

The dynamic type of the actual argument is used to compute the instantiation of any implicitly-bound type parameters. The static type is not used because, with mixed statically- and dynamically-typed code as described in section 3.12, the caller may not have a static type available to provide as the instantiating value. Usually, the distinction between the dynamic and the static type is unimportant. For example, with the simple `min` method defined above, the caller will know that the type of the result is a subtype of the least-upper-bound of the dynamic types of the two arguments. Given the static knowledge that both arguments are of some dynamic type that is a subtype of a particular static type, the caller can infer the static knowledge that the result is some subtype of that static type. Static type information already implies only that the dynamic type of some expression is some subtype of the static type, so calculating static approximations to implicitly-bound type variables is what the type checker has been doing all along.

In two circumstances, however, the distinction between instantiating a type parameter with a dynamic type versus a static type is important. If a implicitly-bound type parameter is used as a normal type for another declaration, i.e., as an upper bound type, then legal actual parameters must be known to be equal to or subtypes of the implicitly bound type variable. For example, if `min` were rewritten as follows:

```
method min(x1:‘T <= comparable, x2:T):T {
   if (x1 < x2, { x1 }, { x2 })}
```

the second argument would be required to be a subtype of the *dynamic* type of the first argument. This requirement could be quite difficult to guarantee statically and is probably not what the programmer meant. Type parameters are usually used directly as type declarations when they are bound to the instantiating parameter of a parameterized type, as in the following method:

```
method store(a:array[‘T], index:int, value:T):void {
   -- store value as the indexth element of the array a
}
```

Here, `T` will be bound to the type of the elements of the array, and typically the `value` argument will be known to be a subtype of that type at the call site.

The distinction between dynamic types and static types for instantiation also appears when instantiating a parameterized object. For example, one way to write the `new_array` method might be the following:

```
method new_array(size:int, initial_value:‘T):array[T] {
   object isa array[T] { size := size, initial_value := initial_value } }
```

Given an initial value of dynamic type `T`, an array is returned with the type `T` as the instantiating value. Because of the fetch and store operations defined on arrays, this array will only be able to contain elements that are subtypes of the *dynamic* type of its initial value. Usually, this would be too restrictive. Accordingly, the real method to create a new array is explicitly parameterized with the type of the elements:

```
method new_array[T](size:int, initial_value:T):array[T] {
   object isa array[T] { size := size, initial_value := initial_value } }
```

Instantiations of parameterized objects record their instantiating types as part of their dynamic run-time state. The instantiating types are used to determine the subtyping relation of the object and when matching the parameterized object's type against a type declaration of the form *type*`[`...`,` `'`T_i `<=` *type*$_i$`,` ...`]`.

The process for matching a dynamic type against a static type declaration containing implicit type parameter bindings depends on locating a single most specific binding type. This may not always be possible without additional constraints. For example, in the following declaration:

```
concrete object strange isa collection[int], collection[string];
```

if the `strange` object is sent the `do` message, its type will be matched against the type declaration `collection[T]`. Both `collection[int]` and `collection[string]` will match, but neither is a subtype of the other. Binding `T` to `int&string` might seem reasonable, but then a type error will result, because `strange` is not a subtype of `collection[int&string]` (such a relationship would have to be explicitly declared). To avoid this sort of problem at method invocation time, objects like `strange` are disallowed. For an object declaration to be legal, there must be at most one most specific instantiation for any of its parameterized supertypes. This check is done when type-checking an object declaration.

3.11.4 Method Lookup

Method lookup is extended to include the number of explicit parameters of candidate methods as part of the method selection process. A message of the form *name*`[`$type_1$`,` ...`,` $type_M$`](`$expr_1$`,` ...`,` $expr_N$`)`, with *M* and *N* zero or greater, will only match methods named *name* with *M* explicit formal type parameters and *N* formal arguments. Method lookup does not depend on the constraints placed on legal instantiating types of explicit formal type parameters. For example,

```
method foo[T <= integer]():void { ... }
```

does not override

```
method foo[T <= number]():void { ... }
```

In fact, these two methods could not legally be defined in the same system, since they have the same name, same number of explicit type parameters, same number of arguments, and same argument constraints.

Once method lookup based on the above pieces of information plus the argument constraints of the candidate methods has located a single most specific target method, the type variables listed as explicit formal type parameters are bound to the corresponding instantiating types passed in the message, and implicitly bound type parameters are bound to the corresponding types derived from the dynamic types of the corresponding actual arguments. Then the constraints expressed by the upper bound type declarations of the formal type parameters and the argument type declarations are checked for consistency.

3.11.5 Parameterized Types and Signatures

When extracting the type of a parameterized object, a parameterized type with the same formal type parameters is created. Similarly, when extracting the signature from a method with explicit and/or implicit type parameters, the signature created has the same explicit type parameters and the same argument type declarations as the method, including implicit type parameter binding information. For example, the declarations

```
abstract type collection[T];

method do(c@:collection[‘T], closure:&(T):void):void { abstract; }

method print(c@:collection[‘T <= printable]):void {
   "[ ".print;
   do(c, &(x:T){ x.print; " ".print; });
   "]".print;
}

method expand_tabs(c@:‘T <= collection[char]):T {
   -- return a copy of c, where tab characters have been replaced with spaces
}


abstract type table[Key,Value] isa collection[Value];

abstract type indexed[T] isa table[int,T];

template type array[T] isa indexed[T];

method new_array[T](size:int, initial_value:T):array[T] {
   object isa array[T] { size := size, initial_value := initial_value } }
```

generates the following types and signatures:

```
type collection[T];

signature do(collection[‘T], &(T):void):void;

signature print(collection[‘T <= printable]):void;

signature expand_tabs(‘T <= collection[char]):T;


type table[Key,Value] subtypes collection[Value];

type indexed[T] subtypes table[int,T];

type array[T] subtypes indexed[T];

signature new_array[T](int, T):array[T];
```

A message send is type-correct if and only if a signature has been extracted that has the same name, number of formal type parameters, and number of actual arguments as the message and whose upper-bound type constraints on actual explicit type parameters and on actual argument objects are obeyed. As before, the type of the result of a message is the greatest lower bound of the result types of matching signatures.

When type-checking the implementation of signatures, message patterns are extended to have the same number of actual type parameters as the signature. Actual type parameters are enumerated in the same manner as are actual object patterns. As before, the upper bound types associated with the formal argument type declarations are used to locate object patterns to enumerate. Only those message patterns whose argument type parameters and argument objects obey the appropriate upper bound type constraints are considered. When simulating method lookup, after finding a target method, the type checker verifies that any properties of and relationships between the explicit and implicit type parameters are satisfied by the message pattern.

3.12 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; all uses of an undeclared variable will be checked for consistency dynamically. Programs without explicit type declarations are thus 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. Where type declarations are present, the system verifies (either statically or dynamically) that only objects that conform to the declared type of a variable are assigned to the variable. Where one statically-typed expression is assigned to a statically-typed variable, formal parameter, or method result, the system signals the user if the type-correctness of the assignment cannot be verified at program-definition time.

The type system models the types of undeclared variables with a special type `dynamic`; `dynamic` may also be specified explicitly as the type of some 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.

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; programmers should not be constrained by the inherent limitations of static type checking. The Cecil type system has been designed to be as flexible and expressive as we knew how to reasonably make it (in particular by supporting multi-methods, separating the subtype and code inheritance graphs and by supporting explicit and implicit parameterization) so that as many reasonable programs as possible 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.

One complication arises when combining dynamic typing with parameterized types. In Cecil, a parameterized object such as `array` can be used without explicit instantiation to indicate a dynamically type-checked version of the parameterized object, as if the parameterized type had been implicitly instantiated with the type `dynamic`. Assignments of an object of `dynamic` type into a statically-declared variable must be dynamically checked to see if the object conforms to the statically-declared type. With scalar objects, this checking is relatively easy, but with parameterized types, the checking may take a significant amount of time. For example, if a dynamically-typed array were assigned to a variable declared to be an array of numbers, then each element of the array object must be checked to verify that the element is a number. Furthermore, after the assignment, the array object can never be mutated to hold objects that are not numbers, since there may still be a reference to the array object (an alias) that is statically declared to be an array of numbers. Since manipulating parameterized types in a dynamically type-checked manner is crucial for exploratory programming, this feature cannot be sacrificed.

When building a new object in exploratory programming mode, declaring the new object's immediate supertypes should not be required. After all, the interface to the object is likely to be in flux. However, if the object is passed as an argument to some method with a static type declaration, then the supertypes of the object are needed to verify that the new object is a subtype of the declared type of the formal parameter. We imagine that a programming environment tool that infers the supertypes of objects given their current interfaces might help in this case. Alternatively, we may change the semantics of mixed statically- and dynamically-typed code so that assigning a dynamically-typed expression to a statically-typed variable does not cause an immediate run-time type check. Only run-time checks for method lookup errors would be included (and these function fine for exploratory objects without well-defined types). This approach also would eliminate the need to compute the types of parameterized objects implicitly instantiated with type `dynamic`. However, it does have the disadvantage that statically-typed code could fail at run-time with a message lookup error, if an erroneous dynamically-typed value was passed into the statically-typed code. In any case, programming environment support to identify where dynamically-typed code remains in an application would be helpful for verifying that no type errors could occur at run-time.

3.13 Open Issues and Future Work

This section discusses some issues relating to the Cecil type system and describes some areas of current and future work.

3.13.1 Efficient Implementation of Type Checking

While simple and accurate, the brute-force enumeration-based signature implementation checking algorithm described in section 3.7.2 is too inefficient to use directly in a practical implementation. Fortunately, the type checking algorithm can be optimized in several ways:

- For many signatures, some argument positions will not be specialized by any implementing method. This was the case for the closure argument of the `do` and `pair_do` methods, for example. For such argument positions, no enumeration of concrete implementing objects is necessary. The type checking algorithm simply checks that the argument types of the unspecialized formals are supertypes of the corresponding types in the signature.

- An object pattern may have the same behavior as another with respect to simulated method lookup. For example, if an object pattern defines no methods of its own (or at least none that are covered by the signature in question) and has only a single parent, then simulating method lookup with the child as the argument will give the same result as those with the parent as the argument, so the child does not need to be checked in addition to the parent. Nearly all object patterns derived from object constructor expressions will have this property.
- The type checking algorithm could be incremental. If a new method is defined, only its covering signatures need be rechecked. If a new object is defined, only those signatures with argument types to which the new object conforms need be rechecked. The previous two optimizations can reduce the set of signatures needing checking greatly; we expect that for most object definitions very few signatures will need rechecking.

In general, the structure of the method specificity graphs and the object inheritance graphs should be used to guide an efficient and direct type checking algorithm. Pursuing these and other optimizations is an important area of current work.

3.13.2 Incremental Type Checking and Modules

The Cecil type checker assumes that the whole program is available for type checking. In some environments, particularly those using external code libraries, this assumption may seem infeasible. However, all code is not strictly necessary for type checking. Instead, only the interfaces to methods are required at final whole-program checking time. The bodies of methods can be checked against a database of signatures incrementally and independently.

A related potential problem is that some errors may not be detectable until the whole program is combined together at what traditionally would be called link-time. These errors occur when two multi-methods are legal separately but become mutually ambiguous when combined. This problem can only appear in the presence of multi-methods, however, and so is qualitatively different than the kind of link-time type errors that can arise with system-level type-checking in Eiffel [Meyer 92].

Extending Cecil with a module facility might solve both of these problems. With modules, programs could be broken up into components which could be completely type-checked and partially compiled in isolation. Combining separate modules together at link-time would not introduce new type errors if they did not interact directly, because the previously ambiguous methods would be in unrelated modules and so invisible to each other. Whole program checking would be reduced to whole module checking, where “whole module” includes imported and textually-enclosing modules.

3.13.3 Dynamic Inheritance

Cecil can be statically type-checked despite its classless object model. This is because Cecil’s object model is restricted compared to some other prototype-based languages such as SELF and Actra where an object can inherit from some other run-time-computed object. Additionally, in SELF, an object can change its parents at run-time. Type-checking such constructs would require program declarations about the implementations of these parent objects in addition to their interfaces. Cecil avoids the need for such new kinds of types by restricting parents and supertypes of object declarations and object constructor expressions to be statically-known, named objects. This allows the type checker to reason about the inheritance graph statically. One could argue that it is desirable for programmers to be able to reason statically about the inheritance graph as well.

However, it is sometimes convenient to be able to compute the kind of object that should be created, passing in as a run-time value the template object to be instantiated. In current Cecil, such an operation could be emulated at user level, passing around explicit “factory” objects. A factory object would implement a `new` message which creates an object of a particular statically-known kind.

3.13.4 F-Bounded Polymorphism

In section 3.11.2, the `min` method was defined as follows:

```
method min(x1:‘T <= comparable, x2:‘T):T {
   if (x1 < x2, { x1 }, { x2 })}
```

The type `comparable` might be defined as follows:

```
abstract type comparable;
method = (x@:comparable, y@:comparable):bool { abstract }
method !=(x@:comparable, y@:comparable):bool { not(x = y) }
method < (x@:comparable, y@:comparable):bool { abstract }
method <=(x@:comparable, y@:comparable):bool { x = y | x < y }
method >=(x@:comparable, y@:comparable):bool { x = y | x > y }
method > (x@:comparable, y@:comparable):bool { y < x }
```

Numbers could be declared to be comparable as follows:

```
number isa comparable;
```

With this declaration, any pair of numbers could be used as arguments to the `min` method. We would also like to state that collections of comparable things are also comparable:

```
collection[‘T <= comparable] isa comparable;
```

Unfortunately, both these declarations cannot both appear in the same Cecil program, because this would require that numbers could be compared against collections of numbers. Subtyping as used in the declaration `‘T <= comparable` in the `min` method only constrains a single object. What we need to do for this case is to be able to describe that two objects come from related types, e.g., that both arguments to `min` are subtypes of `number` or that both are subtypes of the collection type instantiated with related types.

F-bounded polymorphism [Canning *et al.* 89, Cook *et al.* 90] is a different kind of subtyping relation that can describe that two objects come from the same type. One way of describing F-bounded polymorphism, adapted from Black and Hutchinson’s version of F-bounded polymorphism in Emerald [Black & Hutchinson 90] is that a type definition is treated as a *type generator*, a function from a type to a type. The result type of the generator is the type being defined, with the change that what would be self-references in the type are replaced with references to the argument type of the generator. To convert the type generator into the real type, its fixpoint is taken. Type generators can be further extended in “subtypes” with new operations. When the fixpoint is taken of the extended type, what would have been references to the original type are in fact references to the extended type. Unlike a normal subtype, the extended type is not necessarily substitutable wherever its base type might appear. Alternative versions of this idea arise in the categories of Axiom (formerly Scratchpad II) [Watt *et al.*, Jenks & Sutor 92] and the metaclasses of k-bench [Santas 93].

To illustrate these ideas, we will rewrite the above `min` example with some language extensions under consideration. The main extensions support a new kind of subtyping and inheritance relationship, notated by suffixing the `isa`, `subtypes`, or `inherits` keywords with `_pattern` to indicate F-bounded subtyping and/or inheritance rather than normal subtyping and/or inheritance, and a new kind of upper-bound type constraint, notated with `<~` in place of `<=`. Using these features, the `min` example could be written as follows:

```
method min(x1:‘T <~ comparable, x2:‘T):T {
   if (x1 < x2, { x1 }, { x2 })}
number isa_pattern comparable;
collection[‘T <~ comparable] isa_pattern comparable;
```

The definition of the `comparable` type would stay the same. The `isa_pattern` clause in the `number` and `collection` object extension declarations reuse the signatures and the methods of the `comparable` type and object, but with all references to `comparable` replaced with `number` or `collection[T]`. Neither `number` nor `collection[T]` is a (normal) subtype of `comparable`. When `min` is invoked, the arguments must be subtypes of some type that is `comparable` or has been declared to be patterned after `comparable` with an `isa_pattern` or `subtypes_pattern` clause. Consequently, `min` may be invoked on a pair of numbers or on a pair of collections of numbers, but without requiring a number to be comparable to a collection of numbers.

4 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, and static typing. 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 has yet to incorporate dynamic inheritance, one of the most interesting features of SELF. 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 address abstract methods. 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 *et al.* 92], 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] 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 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) and also allows the programmer to define explicit types separate from class declarations (a feature Cecil does not provide). 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.

The only other classless object-oriented language with a static type system of which we are aware is Emerald [Black *et al.* 86, Hutchinson 87, Hutchinson *et al.* 87]. 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.

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.

5 Conclusion

Cecil is a new 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 but still supports a form of encapsulation and an abstract-data-type-based programming style. Cecil compliments this object model with a static type system that describes the interfaces to objects instead of their representation. 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 merit further work, including the semantics of field initialization, the details of the encapsulation mechanism, and future extensions to support modules, predicate objects, and F-bounded polymorphism.

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 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. Claudia Chiang implemented the first version of the Cecil interpreter, in SELF. Stuart Williams implemented the type checker for the monomorphic subset of the Cecil type system. A conversation with Danny Bobrow and David Ungar at OOPSLA '89 provided the original inspiration for this work.

This research has been generously supported by a National Science Foundation Research Initiation Award (contract number CCR-9210990), a University of Washington Graduate School Research Fund grant, and several gifts from Sun Microsystems, Inc.

References

[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.

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

[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.

[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.

[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.

[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*, Utrecht, The Netherlands, July, 1992.

[Chambers 93] Craig Chambers. Predicate Classes. To appear in *ECOOP '93 Conference Proceedings*, Kaiserslautern, Germany, July, 1993.

[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.

[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.

[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.

[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.

[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.

[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.

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

[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.

[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.

[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.

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

[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.

[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

A.1 Grammar

a program is a set of declarations and an expression to evaluate

```
program        ::=  [ decl_block ] expr [“;”]
```

a declaration block is an unbroken sequence of declarations where names are available throughout

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

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

```
decl           ::=  var_decl
               |    object_decl
               |    obj_extension
               |    field_decl
               |    method_decl
```

variable declarations bind names to objects

```
var_decl       ::=  “var” name [type_decl] initializer “;”
initializer    ::=  “=” expr                  the initialized thing is constant
               |    “:=” expr                 the initialized thing is assignable
```

object declarations create new objects and sometimes new types

```
object_decl    ::=  [role] object_or_type name [formal_params]
                        {relation} [ field_inits ] “;”
object_or_type ::=  “object”                  builds an object and a private type
               |    “type”                    builds an object and a public type
role           ::=  “abstract”                only inherited from by non-concrete objects
               |    “template”                only inherited from, but no abstract methods
allowed
               |    “concrete”                completely usable, but no abstract methods or
                                                  uninitialized fields allowed
relation       ::=  “isa” parents
               |    “inherits” parents
               |    “subtypes” types
parents        ::=  object { “,” object }
field_inits    ::=  “{” field_init { “,” field_init } “}”
field_init     ::=  name [location] initializer
location       ::=  “@” object
```

object extensions adjust the declaration of an existing object

```
obj_extension  ::=  name [params] {relation} “;”
```

field declarations define methods manipulating shared state

```
field_decl     ::=  [field_privacy] field_kind “field” name [formal_params]
                        “(” formal “)” [type_decl] field_body
field_privacy  ::=  privacy [ (“get” | “put”) [ privacy (“get” | “put”) ] ]
field_kind     ::=  empty                     copy-down, mutable field, like an instance var
               |    “shared”                  a single memory location, like a class variable
               |    “read_only”               implicitly shared, no set_
               |    “init_only”               implicitly unshared, no set_
```

```
field_body      ::=  initializer “;”              the field is initialized
                |    “;”                          the field is uninitialized
                |    “{” “abstract” “}” [“;”]     field decl is sugar for a pair of abstract methods
```

method declarations define new methods

```
method_decl     ::=  [privacy] “method” method_name [formal_params]
                         function [“;”]
method_name     ::=  name | infix_name
privacy         ::=  “public” | “private”
```

same form used for methods and closures; body is lexically-scoped within enclosing module, method, or closure body

```
function        ::=  “(” [formals] “)” [type_decl] function_body
formals         ::=  formal { “,” formal }
formal          ::=  [name] specializer
specializer     ::=  location [type_decl]         specialized formal
                |    [type_decl]                  unspecialized formal
                |    “@” “:” object               sugar for @object :object

function_body   ::=  “{” body “}”
                |    “{” “abstract” [“;”] “}”

body            ::=  {stmt} result
result          ::=  empty                        do not return a result
                |    expr                         return a result
                |    “^” [“;”]                    do a non-local return, but do not return a result
                |    “^” expr                     do a non-local return, returning a result
```

result of a statementt is ignored, so don’t allow expressions w/o side-effects

```
stmt            ::=  decl_block
                |    assignment “;”
                |    effect_expr “;”
```

assignment only allowed if name is assignable

```
assignment      ::=  name “:=” expr
                |    expr “.” name “:=” expr      sugar for set_name(expr, expr)
                |    expr infix_name expr “:=” expr   set_infix_name(expr, expr, expr)
```

three classes of expression

```
expr            ::=  literal
                |    simple_expr
                |    effect_expr

literal         ::=  integer
                |    float
                |    character
                |    string
simple_expr     ::=  name                         reference a local or global variable
                |    object                       reference a named object
                |    array_expr                   construct an array
                |    closure_expr                 construct a closure
                |    object_expr                  construct an anonymous object
```

```
effect_expr    ::= message
                |    resend
                |    “(” body “)”                introduces a nested scope
```

build an array

```
array_expr     ::= “[” [exprs] “]”
```

build a closure

```
closure_expr   ::= “&” function
                |    function_body               shortcut for zero-arg closure
```

build a new object

```
object_expr    ::= [role] object_or_type {relation} [ field_inits ]
```

send a message

```
message        ::= name [params] “(” [exprs] “)”
                |    expr infix_name expr
                |    expr “.” name               sugar for name(expr)
exprs          ::= expr { “,” expr }
```

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)
```

name an object

```
object         ::= name [params]
```

syntax of types

```
type           ::= name [params]
                |    “&” “(” [types] “)” [type_decl]  type of closure
                |    type “|” type               anonymous least-upper-bound type
                |    type “&” type               anonymous greatest-lower-bound type
                |    “‘” name [“<=” type]        also bind a name to the type
                |    “(” type “)”
types          ::= type { “,” type }

type_decl      ::= “:” type
```

formal parameters for objects and methods

```
formal_params  ::= “[” formal_param { “,” formal_param } “]”
formal_param   ::= [“‘”] name [ “<=” type ]
```

actual parameters for objects and methods

```
params         ::= “[” types “]”
```

A.2 Tokens

```
name           ::= letter {letter | digit} [id_cont]
infix_name     ::= punct {punct} [id_cont] | id_cont
```

```
id_cont         ::=  “_” (name | infix_name)

integer         ::=  [“-”] [radix] hex_digits
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 “\”’nrtvba”
                |    [“o”] digit [digit [digit]]
                |    “x” hex_digit [hex_digit]
line_break      ::=  “\” {whitespace} new_line {whitespace} “\”

letter          ::=  one of “a..zA..Z”
digit           ::=  one of “0..9”
punct           ::=  one of “!#$%^&*-+=<>/?‘~\|”
```

A.3 White Space

```
whitespace      ::=  space | tab | new_line | comment
comment         ::=  “--” {any} new_line      comment to end of line
                |    “(--” {any} “--)”        bracketed comment, can be nested
```