Object-Oriented Multi-Methods in Cecil

Craig Chambers

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195
chambers@cs.washington.edu

Abstract

Multiple dispatching provides increased expressive power over single dispatching by guiding method lookup using the values of all arguments instead of only the receiver. However, existing languages with multiple dispatching do not encourage the data-abstraction-oriented programming style that is encouraged by traditional single-dispatching languages; instead existing multiple-dispatching languages tend to foster a function-oriented programming style organized around generic functions. We propose an alternative view of multiple dispatching that is intended to promote a data-abstraction-oriented programming style. Instead of viewing a multi-method as "outside" of all objects, we view a multi-method as "inside" the objects for which the multi-method applies (on which it dispatches). Because objects are closely connected to the multi-methods implementing their operations, the internals of an object can be encapsulated by being accessible only to the closely-connected multi-methods. We are exploring this object-oriented view of multi-methods in the context of a new programming language named Cecil.

1 Motivation

1.1 Single Dispatching

In most object-oriented languages, a message is sent to a distinguished receiver object, and the run-time "type" of the receiver determines the method that is invoked by the message. Other arguments of the message are passed on to the invoked method but do not participate in method lookup. This style of object-oriented language is termed a *single-dispatching language*, since method lookup (a.k.a. dispatching) is performed only with respect to the single receiver argument.

Single dispatching works well for many kinds of messages, especially those in which the first argument is more "interesting" that the others, in the sense that the first argument alone determines what code should be run to carry out the operation. However, for some kinds of messages, several arguments may be "interesting," with no clear reason to prefer one argument over another. For example, for most standard binary arithmetic messages such as `+`, both arguments are equally interesting, and dispatching on only one is unnatural.

If this problem were confined to pre-defined abstractions like numbers and arithmetic, then perhaps it could be overlooked as an irksome complication of arithmetic, solved once and for all by the system implementors and ignored from then on. However, the asymmetry of single dispatching complicates other kinds of messages. For example:

- Most data types define an equality testing binary operation, and many kinds of objects define ordering relations. The same issues arise as with the standard arithmetic messages.
- The `pairDo(c1, c2, block)` message takes two collections and a closure block and iterates through the two collections in parallel. Neither collection is more important than the other, yet in single-dispatching languages the programmer must favor one collection over the other when selecting the initial implementation of `pairDo`.

To be published in *ECOOP '92 Conference Proceedings*, Utrecht, the Netherlands, July, 1992.

- The `displayOn(shape, device)` message displays a shape on an output device. The particular implementation for `displayOn` depends on both the kind of shape and the kind of output device: rectangles are displayed differently from splines, and a fancy graphics accelerator uses a different rendering strategy for filled polygons than does an X window. Neither the shape nor the output device is more important, and so neither should be treated differently than the other.

Similar examples arise in other programming tasks. The problems of single dispatching thus extend beyond the realm of system implementors to that of the everyday programmer.

1.2 Double Dispatching

In a single-dispatching language, the best solution to the asymmetry problem is *double dispatching* [Ingalls 86]. With double dispatching, the programmer can apply single dispatching to each interesting argument in turn, hand-simulating the effect of dispatching on all interesting arguments. For example, if the programmer wanted to write an equality operation for a pair of two-dimensional points, the programmer could write the following:

```
-- in Point:
    self = aPoint {
        return self.x = aPoint.x && self.y = aPoint.y; }
```

However, if the programmer wanted to be able to compare points against arbitrary objects, this code would be insufficient: the code additionally needs to dispatch on the argument to make sure it's also a two-dimensional point. Using double dispatching, the programmer could rewrite this method as follows:

```
-- in Point:
    self = aPoint { return aPoint.equalsPoint(self); }
    self.equalsPoint(originalSelf) {
        return originalSelf.x = self.x && originalSelf.y = self.y; }
-- in Object:
    self.equalsPoint(originalSelf) { return false; }
```

In general, with double dispatching, for each original method, the programmer must add at least two methods for each additional dispatched argument: one that starts the double dispatching by resending the message to the argument with the type of the receiver encoded in the new name, and one that does the default action for arguments of other types. Maintaining this double dispatching code can be difficult and error-prone.

1.3 Multiple Dispatching

To surmount the limitations of the asymmetric messages of single-dispatching languages, some object-oriented languages include a more powerful form of message passing in which *multiple* arguments to a message can participate in method lookup. These languages are called *multiple-dispatching* languages; methods in a multiple-dispatching language are called *multi-methods*. Perhaps the best-known multiple-dispatching language is CLOS [Bobrow *et al.* 88]; CommonLoops [Bobrow *et al.* 86], one of CLOS's predecessors, pioneered support for multi-methods.

In a multiple-dispatching language, the programmer can handle several "interesting" arguments by writing multi-methods that dispatch on each interesting argument. For example, one + multi-method would be specialized for the case where *both* arguments are fixed-precision integers, another + multi-method for the case where *both* arguments are floating-point numbers, and two more multi-methods for the mixed-representation addition cases. Similarly, multiple dispatching would simplify the programming of the other troublesome cases described earlier: object equality, pairwise iteration, and displaying shapes on output devices. For example, point equality could be written as follows:

```
-- the default implementation of equality returns false (don't dispatch on either argument):
x = y { return false; }

-- implementation of equality for a pair of points
-- (v@obj means dispatch on argument v, and match only for actuals that are equal to or inherit from obj):
p1@Point = p2@Point {
   return p1.x = p2.x && p1.y = p2.y; }
```

Supporting multiple dispatching at the language level avoids the need for double dispatching. Since only methods with useful code bodies need to be written, programmers can spend more time writing productive code.

Each multi-method can be written with the knowledge of the implementations of all dispatched arguments, thus streamlining method bodies and improving both coding speed and readability. Programmers can see more easily those argument combinations that have methods defined for them; under double dispatching, programmers would need to trace through lots of extraneous code to determine which argument type combinations had implementations defined for them. Finally, requiring programmers to implement their own argument dispatching manually opens the door for programming errors that may be hard to locate. Multi-methods specify method lookup *declaratively*, while double dispatching specifies method lookup *procedurally*. In this particular case, where dispatching is highly stylized and idiomatic, declarative specification is superior to procedural specification.

1.4 But is it Object-Oriented?

Since multiple dispatching appears to be more expressive, more natural, more readable, and less error-prone than single dispatching, why do so few object-oriented languages support multi-methods? In fact, the apparent advantages of multiple-dispatching languages are far from universally acknowledged. Some reasons are that multiple-dispatching languages have been more complex that competing single-dispatching languages, and their implementations have not been as efficient. However, even ignoring these disadvantages, many practicing programmers used to single-dispatching object-oriented languages complain that multi-methods "just don't feel object-oriented." This common feeling reflects a basic difference in the programming styles encouraged by single- and multiple-dispatching languages.

When using a single-dispatching language, the programmer associates methods with the data types (or classes, or objects) for which they are implemented. The programmer's mental model is one of defining abstract data types, with their associated state (i.e., instance variables) and operations (i.e., singly-dispatched methods). These abstract data types are organized into inheritance hierarchies based on implementation and/or interface similarity. A whole design and implementation methodology has been developed around this *data-abstraction-oriented programming style* fostered by single-dispatching languages.

In contrast, existing multiple-dispatching languages do not provide much linguistic or programming environment support for a data-abstraction-oriented programming style. Instead of defining methods as part of abstract data types, multi-methods are defined externally to objects. In multiple-dispatching languages such as CLOS, multi-methods with the same name are grouped into *generic functions*. Generic functions have a decentralized, case-analysis-style implementation similar to pattern matching as found in functional programming languages, but this implementation is hidden from clients: a generic function can be invoked by clients as if it were a simple function.* To a large extent, this approach to multi-methods integrates the function-oriented and object-oriented programming styles; merging Lisp and object-oriented programming was an explicit goal for CLOS [Gabriel *et al.* 91].

However, the extant generic-function-based approach to multiple-dispatching object-oriented languages tends to encourage a function-oriented programming style at the expense of a data-abstraction-oriented programming style. Since multi-methods cannot be viewed as completely contained within any single data type, the generic-function-based approach treats multi-methods as outside of all objects. However, the data-abstraction-oriented programming view seems to depend on methods being contained within some data type (or class, or object). Consequently, encouraging a data-abstraction-oriented programming style in the presence of multi-methods appears problematic. For example, in the future research section of his introductory paper on Flavors, Moon mentions that Flavors could be extended to support "multiadic operations," but expresses the concern that first "a coherent and useful framework for organizing programs needs to be developed" [Moon 86]. Before multi-methods will feel object-oriented to programmers used to single-dispatching languages, the programming methodology must support data-abstraction-oriented programming.

Furthermore, the view that multi-methods exist external to all objects impedes encapsulating the internal implementation decisions of data types (or classes, or objects). Single-dispatching languages commonly support object-level encapsulation, and the data-abstraction-oriented programming methodology relies on this facility. Encapsulation appears to hinge on the fact that private methods and instance variables are contained wholly within a data type, and only other methods also completely contained within the data type can access the private information; only the interfaces of public operations are exported beyond the boundaries of the encapsulated data type implementation. In the generic-function-based view, multi-methods are considered outside of all objects, thus precluding object-level encapsulation.†

1.5 Towards Object-Oriented Multi-Methods

We believe that for multiple-dispatching object-oriented languages to become more widely accepted and their benefits to become available to a larger number of programmers, they must support a data-abstraction-oriented programming methodology. This paper describes an alternative to the generic-function-based view of multi-methods that we believe is more compatible with a data-abstraction-oriented design and programming style. The central idea is that a multi-method is viewed as *part of* each data type (or class, or object) for which the multi-method dispatches, rather than outside of all data types. Since multi-methods are closely connected to the objects on which they dispatch, objects can be encapsulated: only the closely-connected multi-methods are granted privileged access to an object's private internals.

---

* Generic functions are not mere implementation devices; rather, they are visible to the CLOS programmer. In particular, all multi-methods with the same name must have *congruent lambda lists*, meaning that they must have the same number of required arguments, the same number of optional arguments, and similar `&key` and `&rest` argument declarations. This prevents individual multi-methods from being developed independently.

† Packages in CLOS can help organize the program's name space by hiding names within modules, but this provides a different kind of encapsulation unrelated to data types. Of course, CLOS's Lisp tradition has not emphasized encapsulation.

We are exploring these ideas in the context of a new purely object-oriented programming language named Cecil. Much of Cecil's design arose by identifying facilities that in single-dispatching languages are specific to the receiver argument and then extending these facilities so that any argument can receive special treatment. Several of Cecil's features are direct consequences of this design approach:

- In single-dispatching object-oriented languages, a message is sent to the distinguished receiver argument, which in turn determines the appropriate method for this message; other arguments do not participate in method selection. In Cecil, the special properties of the receiver are conferred on all arguments: a message is sent to *all* its arguments, and together these arguments select which method is appropriate to implement the message.

- In a single-dispatching language, methods are contained within a single monolithic data type (or class, or object) implementation. This can be modeled by making the receiver formal argument explicit and then specially *constraining* the receiver argument by the appropriate data type). The constraint link between the receiver formal argument and the data type represents the "contained within" or "part of" relationship between methods and data type implementations in single-dispatching languages. Once viewed in this way, however, the relationship between the receiver and the containing data type can be generalized for multiple-dispatching languages by allowing any argument to be constrained by a data type implementation. These argument constraints correspond to the *parameter specializers* of CLOS. Thus a multi-method can be viewed as "part of" several data types, specifically those for which it dispatches.

- In programming environments for single-dispatching languages, such as the Smalltalk-80[*] environment [Goldberg & Robson 83, Goldberg 84], browsers allow the programmer to view all the methods associated with a particular class. For Cecil, the programming environment should continue to allow the multi-methods associated with an object to be browsed from that object, but any particular multi-method may be browsed from any of its linked objects. The programming environment thus plays a crucial role in reinforcing a data-abstraction-oriented programming style.

- In most single-dispatching languages, methods are granted privileged access to the private features of their enclosing data type implementation, i.e., to the internals of the object of which the method is a part. This same encapsulation model is used in Cecil, with the extension that a multi-method is granted privileged access to all objects of which the multi-method is a part, i.e., to the objects that are the method's argument constraints.

1.6 Outline of this Paper

This rest of this paper describes in more detail Cecil's support for object-oriented multi-methods. The next section outlines Cecil's object and message passing model, focusing on the ways in which Cecil supports multi-methods while fostering data-abstraction-oriented programming. Section 3 describes Cecil's object-based encapsulation model. Section 4 reports on the current status of the Cecil language project. Section 5 discusses related research. Section 6 concludes and identifies some questions this work leaves unanswered.

---

[*] Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

2 The Cecil Object Model

A Cecil program consists of a collection of object and multi-method definitions, plus an expression that is evaluated when the program is invoked. Objects define implementations of data abstractions. These data abstractions provide a set of operations (some externally visible, others hidden), linguistically specified as methods defined on the object implementing the data abstraction.

2.1 Objects and Inheritance

Cecil has a classless (prototype-based) object model: self-sufficient objects implement data abstractions, and objects inherit directly from other objects to share code. Several other prototype-based models have been proposed [Borning 86, Lieberman 86, LaLonde *et al.* 86, Ungar & Smith 87, Lieberman *et al.* 87]. Cecil uses a classless model primarily because of its elegance and simplicity. Our approach to object-oriented multi-methods, however, does not hinge on this decision; our approach also could be adopted in a class-based multiple-dispatching language.

For example, the following declarations define a simple hierarchy for integers:

```
int = object inherits number -- behavior for integers
-- integer operations here


smallInt = object inherits int -- fixed-precision integers
-- fixed-precision operations here


bigInt = object inherits int -- arbitrary-precision integers
-- arbitrary-precision operations here


zero = object inherits int -- special zero object behavior
-- zero operations here
```

In Cecil, inheritance of code as in this example is distinct from *subtyping* (inheritance of interface or of specification). A special `type` annotation on an object definition declares that the object also specifies a type (a set of method signatures), and `subtypes` declarations separate from the `inherits` declarations describe an object's relationship to types in the subtyping lattice. This distinction enables an object to be a subtype of another without being forced to inherit any code, and enables an object to inherit code without being restricted to be a legal subtype of the parent object. Other researchers also have argued the benefits of distinguishing between inheritance of implementation and inheritance of interface or specification [Snyder 86, Halbert & O'Brien 86, Cook *et al.* 90].

Additionally, object declarations can be annotated with the role of the object in executing programs, in support of static type checking. For example, the `int` object above could be annotated as `abstract` implying that it will only be used as a packet of behavior and/or specification and not as a run-time manipulable object, the `smallInt` and `bigInt` objects could be annotated as `template` objects implying that they will only be used as patterns for new objects created from them at run-time, and the `zero` object could be annotated as `unique` implying that it is a one-of-a-kind object not to be inherited or instantiated from but otherwise fully manipulable. The type checker will verify that programs observe these annotations and consequently guarantee that certain potential run-time errors cannot occur.

Further details of Cecil's static type system are beyond the scope of this paper.

2.2 Methods

In Cecil, multi-methods specify the kinds of arguments for which their code is designed to work. For each formal argument of a method, the programmer may specify that the method should apply 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, called an *argument constraint*. Formal arguments with such restrictions are called *constrained arguments*. An unconstrained formal argument can accept any actual argument. Any number of arguments may be constrained, supporting three idioms:

- If zero arguments are constrained, the multi-method acts like a normal procedure. This sort of method can be useful as a default case, overridden by other more specific multi-methods with constrained arguments.
- If only the first argument is constrained, the multi-method acts like a normal singly-dispatched method. The semantics of such a one-constraint multi-method is intended to exactly mimic the semantics of a method in a single-dispatching object-oriented language.
- If several arguments are constrained, the method is a true multi-method.

Callers are unaware of implementation decisions made in terms of which arguments, if any, are constrained and how many different multi-methods are cooperating to implement the behavior.

To illustrate, the following method implements addition for objects represented as `smallInt` objects:

```
x@smallInt + y@smallInt {
    -- primAdd performs primitive arithmetic of (children of) primInt
    -- primAdd takes a failure block which is invoked if an error (e.g., overflow) occurs
    ↑ primAdd(x, y, { &errorCode | -- code to handle failure (e.g., retry as bigInts) -- } ) }
```

Details of the syntax are as follows:

- The `x@smallInt` notation constrains the `x` formal argument by the `smallInt` object; formals without `@`... suffixes are unconstrained. There is no implicit `self` formal; all formals are listed explicitly.
- The body of a method is a sequence of expressions. These expressions can be constants, variable references, assignments to local variables, or messages. Usually, Algol-like syntax is used to specify a message send; again, all actual arguments are listed explicitly. Syntactic sugar exists for two common cases: `p.x` is syntactic sugar for `x(p)`, and `p.x := y` is sugar for `set_x(p, y)`.
- A sequence of expressions enclosed within braces and nested inside of a method describes a lexically-nested closure object. If the closure takes arguments, they are listed just inside the closure and prefixed with a `&` symbol (intended to be reminiscent of the λ symbol) and separated from the closure's body by a | symbol. All control structures in Cecil are implemented at user level using messages and closures.* The body of a closure is invoked by sending the closure object the `eval` message.
- The last expression in the body of a method or closure may be prefixed with a ↑ symbol to indicate that the result of the last expression is returned as the result of the method or closure; a method or closure without a ↑ does not return a result. Nested closures may

* The one exception is the predefined `loop` method inherited by all closures that repeatedly invokes its argument closure over and over, until some closure performs a non-local return out of the loop. We use the `loop` method instead of user-defined recursion and required tail-recursion elimination as in Scheme [Rees & Clinger 86] because the latter precludes complete source-level debugging [Chambers 92, Hölzle *et al.* 92].

force a *non-local return* (a return to the caller of the outermost lexically-enclosing method rather than to the caller of the closure, much like a `return` statement in C) by using the ↑↑ symbol instead of the ↑ symbol.

The following declarations extend the ongoing integer hierarchy example with additional methods:

```
int = object inherits number -- behavior for integers

isZero(x@int) { ↑ false } -- the default case for integers; overridden below for zero

factorial(x@int) {
    -- if invokes a user-defined method, with different definitions for the true and the false objects
    if(x <= 1,
        { ↑↑ 1 }, -- return from the factorial method, not from the closure
        { ↑↑ x * factorial(x - 1) } ) }

for(from@int, to@int, block) {
    i ::= from -- declare and initialize a new local variable
    while({ i <= to }, {
        eval(block, i) -- invoke the block (closure) with an argument
        i := i + 1 } )
    -- do not return a result (no ↑)
    }


smallInt = object inherits int, primInt -- fixed-precision integers

x@smallInt + y@smallInt {
    -- primAdd performs primitive arithmetic of (children of) primInt
    -- primAdd takes a failure block which is invoked if an error (e.g., overflow) occurs
    ↑ primAdd(x, y, { &errorCode | -- code to handle failure (e.g., retry as bigInts) -- } ) }

x@smallInt + y@bigInt { ↑ asBigInt(x) + y } -- support mixed-representation arithmetic

asBigInt(x@smallInt) {
    -- code to create an arbitrary-precision integer from a fixed-precision integer -- }


bigInt = object inherits int -- arbitrary-precision integers

x@bigInt + y@bigInt { -- code to add arbitrary-precision integers -- }

x@bigInt + y@smallInt { ↑ x + asBigInt(y) } -- support mixed-representation arithmetic


zero = object inherits int -- special zero object behavior

z@zero + x { ↑ x } -- zero plus anything is that thing

x + z@zero { ↑ x }

z1@zero + z2@zero { ↑ z1 } -- resolve the ambiguity between the previous two methods

isZero(z@zero) { ↑ true } -- override the default method above
```

Cecil also allows methods to be defined whose body is only the keyword `abstract`. Such *abstract methods* serve as placeholders for real implementations that will be defined later in children and as specifications of expected interfaces. An abstract method cannot itself be invoked but instead should be overridden with a non-abstract method for all concrete children. Section 3.3 will describe how abstract methods help control object encapsulation.

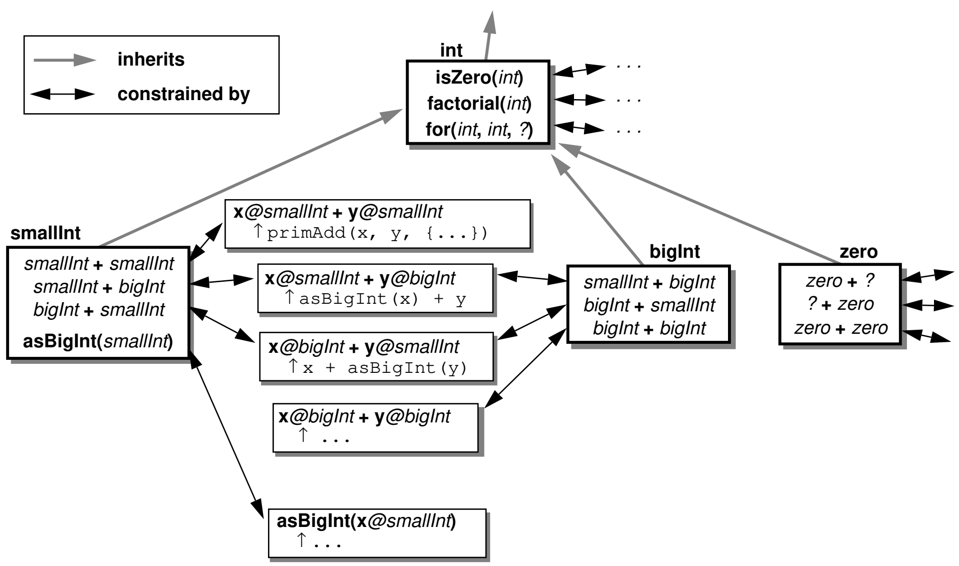
Cecil's classless object model combines with its definition of argument constraints to support something similar to CLOS's `eql` specializers without additional mechanism. In CLOS, an argument to a multi-method in a generic function may be constrained to apply only to a particular object by annotating the argument constraint with the `eql` keyword. Cecil can achieve a similar effect, since methods already are constrained by objects. In Cecil, a method constrained by an object will also apply to an object's children, if any.

Argument constraints guide method lookup.* Informally, when a message is sent to the argument objects, the system finds all methods with the same name and number of arguments as the message. The system then weeds out those methods whose argument constraints are too restrictive to apply to the actuals passed in the call. (If after this step no methods are applicable, a "message not understood" error is reported.) Of the remaining applicable methods, the system locates the most specific one (the one whose argument constraints are most specific), and invokes that method to implement the message. (If no single method is most specific, a "message ambiguous" error is reported.) Section 2.6 describes method lookup more precisely.

2.3 Programming Environment Support

Methods and objects are connected through the argument constraints of methods. Programmers are intended to view objects and their connected methods as a unit which implements a data type; the methods defined for a particular object should always be directly accessible from the object. This is the essence of data-abstraction-oriented programming.

The mental model of the program as a collection of objects exerting "spheres of influence" over connected methods 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, achieving our goal of data-abstraction-oriented programming in the presence of multi-methods depends on support from a graphical interactive programming environment that can display 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. The user could view the same multi-method from each of its constraining 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:



---

* Multi-methods dispatch on particular object implementations (i.e., those objects that inherit from the argument constraints) rather than on particular types. Undispatched arguments may be annotated with type declarations, but these type declarations have no impact on message lookup.

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, an object-based 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 alternative views of the program where all the methods that together implement some algorithm are displayed simultaneously on the screen. This algorithm-based view is subtly different than a generic-function-based view, since the algorithm-based view could contain related methods with different names and could exclude unrelated methods that happen to have the same name.

2.4 Mutable State

In existing multiple-dispatching languages such as CLOS, instance variables (*slots* in CLOS terminology) are defined with the objects, while methods are defined externally in generic functions. The instance variables are accessed directly using special linguistic constructs, while multi-methods are invoked through generic functions. Such distinctions between methods and instance variables reduce expressiveness. If different rules apply to variables and methods, programmers of abstractions cannot easily change their minds about what is stored and what is computed. In particular, the restriction in most object-oriented languages that instance variables cannot be overridden limits the reusability of code to only those abstractions which wish to use the same representation. Other object oriented languages such as SELF [Ungar & Smith 87, Hölzle *et al.* 91a] and Trellis [Schaffert *et al.* 85, Schaffert *et al.* 86] have demonstrated the advantages of accessing instance variables solely through special *get* and *set accessor methods*.

In the traditional generic-function-oriented view of multi-methods, however, accessing instance variables solely through multi-methods appears problematic. Since these methods are defined “outside” of any particular object, how can each object have its own local state? How can the accessor multi-methods be connected to the object “containing” the instance variable? Cecil’s alternative data-abstraction-oriented view of multi-methods provides a way of resolving this dilemma. Instance variable accessor methods are constrained by the object “containing” the instance variable, thus establishing the link between the accessor methods and the object, and the accessor methods accordingly will be considered part of the object’s implementation. Section 3 describes how in Cecil these accessor methods can be encapsulated within the data abstraction implementation and protected from external manipulation.

In Cecil, a method declaration whose body is the keyword `field` defines a pair of accessor methods which share hidden mutable state. The get accessor method (whose name is the same as the declared field) takes a single argument, constrained by the object “containing” the field, and returns the contents of the field. The set accessor method (whose name is `set_` followed by the declared field name) takes two arguments: one constrained by the object containing the instance variable and the second unconstrained. When invoked, the set accessor mutates the contents of the field to refer to its second argument; set accessors do not return results. To illustrate, the declaration:

```
var(o@obj) { field }
```

defines two methods:

```
var(o@obj) { ↑ <field_contents> }
set_var(o@obj, v) { <field_contents> := v }
```

The argument constraint of the get and set accessors establishes the connection between the accessor methods and the object "containing" the memory location. Accessor methods are invoked just like other methods and can be overridden by other methods and vice versa, thereby streamlining the language's semantics and increasing flexibility.

In class-based object-oriented 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, such as SELF and Lieberman's delegation language, 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. To get the effect of object-specific state, in SELF 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) the 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].

In Cecil, these problems with other prototype-based languages are addressed by allowing a field to be declared as local (the default) or shared (by prefixing the `field` keyword with the `shared` annotation). A shared field is shared by all inheriting objects, and so acts like a class variable or like a data slot in SELF. A single memory location is allocated for a shared field, and its get and set accessors manipulate this same memory location for all inheriting objects. A local field, on the other hand, maintains a different memory location for each inheriting object, and so acts like an instance variable. When an accessor method of a local field is invoked, the identity of the accessor method's first argument determines which memory location to fetch or update. Each object effectively receives its own copies of the memory locations for all its inherited local fields. Thus Cecil programmers can define a data type as a single object and still support both shared and object-specific variables.

Cecil allows a field to be given an initial value in the form of an expression that will be evaluated the first time the field is accessed; this supports functionality similar to the `once` functions of Eiffel [Meyer 88, Meyer 92] and other languages. Cecil also allows a shared field to be annotated as `read_only` (implying that no set accessor method should be generated, thus supporting constant fields) and allows a local field to be annotated as `init_only` (implying that no set accessor method should be generated but still allowing an initial value to be specified whenever an object inheriting the field is created).

The following declarations illustrate fields and multi-methods by defining a hierarchy for immutable lists:

```
list = object inherits orderedCollection
isEmpty(l@list) { ↑ size(l) = 0 }
size(l@list) { abstract } -- size must be implemented in all concrete children
do(l@list, block) { abstract } -- iteration; implemented below
pairDo(l1@list, l2@list, block) { abstract } -- pair-wise iteration; implemented below
maxSize(l@list) { shared field := 0 } -- records maximum length of any list

nil = object inherits list -- empty list
size(n@nil) { ↑ 0 }
do(n@nil, block) { } -- iterating over all elements of the empty list: do nothing
pairDo(n@nil, l@list, block) { } -- do nothing
pairDo(l@list, n@nil, block) { } -- do nothing
pairDo(n1@nil, n2@nil, block) { } -- do nothing

cons = object inherits list -- non-empty lists
head(c@cons) { init_only field }
tail(c@cons) { init_only field }
size(c@cons) { ↑ 1 + size(c.tail) } -- c.tail is syntactic sugar for tail(c)
do(c@cons, block) {
   eval(block, c.head) -- call block on head of list
   do(c.tail, block) } -- recur down tail of list
pairDo(c1@cons, c2@cons, block) {
   eval(block, c1.head, c2.head)
   pairDo(c1.tail, c2.tail, block) }
```

All lists will share the one `maxSize` field, but each `cons` cell will have its own `head` and `tail` fields. These latter two fields can only be set when a `cons` cell is created; no `set_head` or `set_tail` methods are generated.

2.5 Object Creation

New objects are created at run-time in Cecil using existing language features: object and inheritance declarations. For example, the following method creates a new list object:

```
prepend(h, t@list) { -- dispatch on second argument (!)
   -- create new child of cons with the given initial field values
   c ::= object inherits cons [ head := h, tail := t ]
   c.maxSize := max(c.maxSize, c.size) -- 3 instances of syntactic sugar here
   ↑ c }
```

A new object is created by evaluating an `object` expression, inheriting from existing objects to get the desired behavior. Object-specific values for local fields that are not annotated as read-only can be specified as part of the `inherits` clause as a series of (field name, field value) pairs. This same functionality is available for objects defined at program-definition-time.

2.6 Method Lookup

All computation in Cecil is accomplished by sending messages to objects. The lion's share of the semantics of message passing specifies method lookup. In single inheritance languages, method lookup is straightforward. Multiple inheritance is more expressive, but it 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 constraints 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 or multiple dispatching is how exactly ambiguities are resolved.

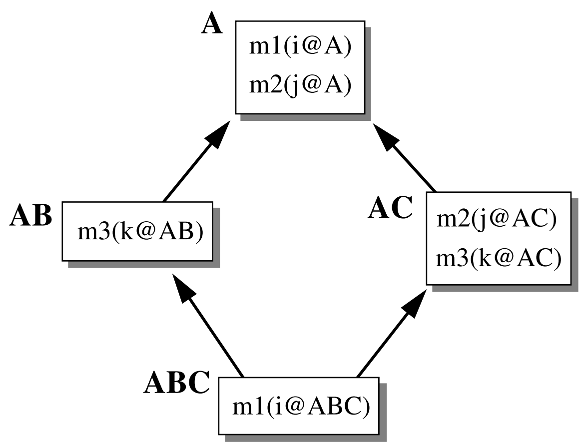
Some languages resolve all ambiguities automatically. For example, Flavors linearizes the class hierarchy, producing a total ordering on classes based on each class' local left-to-right ordering of superclasses that can be searched without ambiguity just as in the single inheritance case. However, linearization can produce unexpected method lookup results, especially if the program contains errors [Snyder 86]. CLOS and CommonLoops 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. First, we expect programmers will sometimes make mistakes during program development. The language should provide the programmer with feedback about errors rather than silently resolving them. Our experience with SELF leads us to believe that bugs that are hidden by such automatic language mechanisms are some of the most difficult and time-consuming to find. Our SELF experience also encourages us to strive for the simplest possible inheritance rules that are adequate. Even the most straightforward extensions can have subtle interactions that make the extensions difficult to understand and to use [Chambers *et al.* 91]. Finally, complex inheritance patterns can hinder future program evolution. If method lookup can depend on program details such as parent ordering and argument ordering, the programmer must constantly be concerned with such details. Even worse, usually it is unclear from the program text which details are important for method lookup. Accordingly, for Cecil we sought for as simple a system as was reasonable that supported multiple inheritance and multiple dispatching. We hope that this emphasis on simplicity will reduce the complexity traditionally associated with multiple dispatching.

Cecil's method lookup rules interpret a program's inheritance graph as a partial order on objects: an object C is greater in the partial order than another object P if and only if C is a descendant of P. This ordering on objects in turn induces analogous orderings on the connected methods that determine when one method overrides another: in the partial order on methods with a particular name and number of arguments, one method M is greater than another method N (i.e., M overrides N) if and only if for each argument position i, the i^{th} argument constraint object of M is greater than or equal to the i^{th} argument constraint object of N; constrained arguments are greater than unconstrained arguments. Since the system can include at most one method with a particular name and sequence of argument constraint objects, this ordering implies that M has at least one argument constraint object that is strictly greater than the corresponding argument constraint object of N. In other words, one method overrides another if it uniformly is at least as specific as the other, and is strictly more specific for at least one argument. Methods left unordered by this rule are considered mutually ambiguous.

Once the methods are ordered, method lookup is simple. First the set of applicable methods is computed as those methods with the same name and number of arguments as the message and whose formal argument constraint objects are less than or equal to the corresponding message argument objects. Then the single greatest method is extracted from the set of applicable methods, and this method is returned as the result of method lookup. If the set of applicable methods is empty, the message results in a "message not understood" error. If more than one method applies but none is strictly greater than all others, the message results in a "message ambiguous" error.

The following inheritance graph illustrates the method lookup rules in the presence of multiple inheritance but only single dispatching:



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

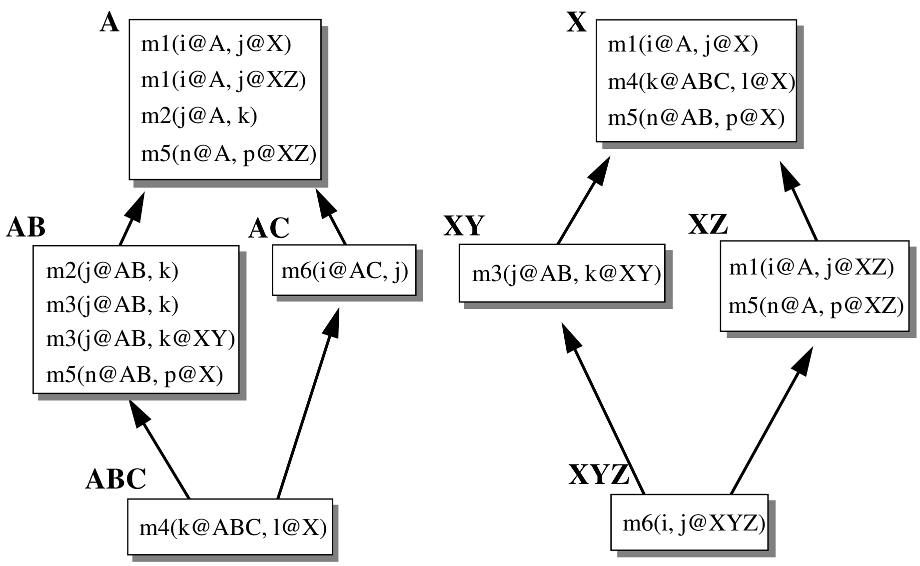
If the message `m1` is sent to the `ABC` object, both the implementation of `m1` whose formal argument is constrained to the `ABC` object and the implementation of `m1` constrained by `A` will apply, but the method constrained by `ABC` will be greater than the one constrained by `A` (since `ABC` is greater 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` constrained for the `A` object would be chosen; the version of `m1` constrained by `ABC` would be too specific and so would not apply.

If the `m2` message is sent to `ABC`, then both the version of `m2` whose formal argument is constrained by `A` and the one whose formal is constrained by `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 greatest 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, the system therefore reports an ambiguous message error. The programmer then is responsible for resolving the ambiguity explicitly.* Sends of `m3` to either `AB` or `AC` would be unambiguous, since the other method would not apply.

* Typically these ambiguities are resolved by writing a method in the child object which redirects the message to a particular parent. To support delegating messages to any parent or to a specific parent, Cecil provides a variant of SELF's *resend* mechanism extended to the multiple dispatching case. Other languages provide related facilities such as CLOS's `call-next-method`, Smalltalk-80's `super`, and C++'s `::` operator.

The following inheritance graph illustrates Cecil's method lookup rules in the presence of multiple dispatching (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 (constrained by) the same object, as long as some other arguments are constrained 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 unconstrained
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 greatest applicable method
m6(ABC, XYZ)	"message ambiguous"	AC overrides unconstrained but XYZ overrides unconstrained ⇒ no single greatest applicable method

This partial ordering view of multiple inheritance has several desirable properties. First, it is simple. It implements the intuitive rule that children override their parents (they are "greater" in the partial ordering), but does not otherwise order parents or count inheritance links or invoke other sorts of complicated rules. Second, ambiguities are not masked; programmers are warned about potential ambiguities before the program runs. Third, this form of multiple inheritance is robust in the face of programming changes. Programmers can change programs fairly easily, and the system will report immediately any ambiguities which may arise. 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. Finally, 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.

3 Encapsulation

Our alternative view of multi-methods is intended to foster data-abstraction-oriented programming. Towards this end, multi-methods are closely connected with the objects that they dispatch on, and only weakly connected with each other, just as in a single-dispatching language. Programmers can concentrate on designing and implementing abstractions; multi-methods enable programmers to build cooperating implementations of abstractions. However, true data abstraction requires some sort of *encapsulation* of the internal implementation details of an abstract data type. The clients of an abstraction should be aware only of the externally-visible interface to the abstraction (the set of operations supported by the abstraction), and clients should be unaffected by any changes to how these externally-visible operations are implemented.

Most languages achieve encapsulation by dividing up the operations of the data type into public, externally-visible operations and private, internal operations. Anyone may invoke a public operation, but only methods that are part of the object's implementation can invoke one of the object's private operations.[*] The difference between public and private operations hinges on the notion of "inside" and "outside" an abstract data type: only methods "inside" the data type can access private operations. But in a language with multiple dispatching, multi-methods cannot be seen as inside any single abstract data type implementation. How can a multi-method get access to an internal operation without forcing that operation to be made public? How can a multi-method itself be made private? If these questions cannot be answered satisfactorily, then languages with multiple dispatching will not fully support data-abstraction-oriented programming.

3.1 Privileged Multi-Methods

Fortunately, these questions can be answered in a way that still provides the benefits of encapsulation and abstract data types. The key insight is that a multi-method is "inside" of all the objects which are its argument constraints, and so a multi-method is granted access to all the private operations of all of its constrained arguments. Multi-methods are *not* granted privileged access to unconstrained arguments; an unconstrained argument may be manipulated only using the externally-visible public interface of the argument. This is a direct extension to the situation in single-dispatching languages, where a method is granted access to all the private operations of its receiver argument.

This model of encapsulation seems reasonable from a practical programming point of view. When writing a multi-method that is intended for arguments implemented in certain ways (indicated by the formal arguments' constraints), it seems natural to invoke operations

[*] Some object-oriented languages such as C++ and Trellis further subdivide private operations into operations private to a single class and operations private to a class and its subclasses. Whether or not this distinction exists is orthogonal to support for object-oriented multi-methods, and so we do not delve into the issue further, other than to note that Cecil currently does not include the distinction, i.e., Cecil does not provide support for encapsulating a parent from its children.

internal to those arguments' implementations. For example, the earlier non-empty list abstraction might be rewritten with a protected representation:

```
cons = object inherits list
private head(c@cons) { field }
private tail(c@cons) { field }
size(c@cons) { ↑ 1 + size(c.tail) }
do(c@cons, block) {
   eval(block, c.head)
   do(c.tail, block) }
pairDo(c1@cons, c2@cons, block) {
   eval(block, c1.head, c2.head)
   pairDo(c1.tail, c2.tail, block) }
```

The `head`, `set_head`, `tail`, and `set_tail` operations would be hidden from public view; only operations that dispatch on the `cons` object (or one of its children) could access these private operations. The operations that are part of the `cons` implementation must be granted access to the protected representation, however, or they could not do their job. In particular, the `pairDo` operation needs privileged access to both dispatched arguments. Allowing multi-methods privileged access to their constrained arguments doesn't represent a serious breach of encapsulation, since these multi-methods already have some amount of special knowledge about their constrained arguments beyond what a normal client would know: they know part of each constrained argument's ancestry, conveying a certain amount of internal implementation information.

Granting privileged access only to operations that dispatch on the object also seems reasonable from the standpoint of an implementor of an abstract data type. Encapsulation is intended 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 constraints are just as easy to locate as are normal singly-dispatched methods, especially in Cecil where multi-methods are closely linked to their argument constraints. In the above 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.

3.2 Private Multi-Methods

Multi-methods must also be able to be declared private, particularly in Cecil where all methods potentially are multi-methods. With singly-dispatched methods, the meaning of private is fairly clear, but with multi-methods, the language designer is faced with two choices: are private multi-methods private to each of the argument constraints individually, or private to them all as a group? The distinction between these two choices is exposed by some caller that is "inside" the implementation of one of a private multi-method's argument constraints but not another. For example, the implementation of `displayOn` for filled polygons on a fancy graphics device might rely on an internal private method:

```
displayOn(shape@filledPolygon, device@fancyGraphicsHardware) {
   setUpDisplay(shape, device)
   ... rest of code ... }
private setUpDisplay(shape@filledPolygon, device@fancyGraphicsHardware) {
   ... initialize graphics hardware for filled polygon displays ... }
```

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

```
draw(shape@filledPolygon) {
   setUpDisplay(shape, standard_display())
   ... more code ... }
```

If `standard_display()` returns `fancyGraphicsHardware` at run-time, should this method be granted access to `setUpDisplay`, even though `draw`'s argument constraints imply that it is part of only the filled polygon implementation? Under the first design where private multi-methods are accessible from any of their connected implementations, the `draw` method would be granted access. Under the second design where private multi-methods can only be accessed from multi-methods that also are part of the same abstract data type implementations, the `draw` method would be denied access, since it is not part of the implementation of the fancy graphics hardware data type. Cecil adopts this second design, since it provides stricter encapsulation and makes it easier to identify code that may depend on internal implementation details.

We can now define precisely when one multi-method C sending a message m has privileged access to a private multi-method P invoked by m. C is granted access to P at call site m if and only if for each formal argument constraint a_p of P, the corresponding actual argument of m is in turn one of C's formal arguments, and this formal argument is constrained by an object a_c that is equal to, a descendant of, or an ancestor of a_p. Privileged access is allowed only where it is clear statically from the program text; that is why constrained formal arguments must be passed through directly as arguments to the message seeking privileged access. This rule grants children privileged access to the private methods of their ancestors and also grants parents privileged access to the private methods of their descendents; both directions are important for practical programming.

3.3 Possible Breaches of Encapsulation

With the encapsulation rules described so far, 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 constrained by the target object. This new method would be considered part of the constrained object's implementation and so receive privileged access. We consider the easy extensibility of existing objects to be one of Cecil's strengths, but this extensibility does not always require privileged access; new methods can be added to an existing object that only manipulate the object through public operations. To protect the internals of an object from future multi-methods added to the object, an object declaration could list explicitly those multi-methods that are granted access to the object's private operations; the previous encapsulation rule would be amended to check this "access control list" in addition to the other checks. Other multi-methods would be allowed to dispatch on the object, but wouldn't be able to access its private operations. To make this list easy to specify, the programming environment could include a facility to construct an initial approximation to the access list from the set of methods that currently dispatch for an object, and the programmer could then edit this list as appropriate. Since this approach would complicate the encapsulation model and might become a maintenance problem, the current Cecil design does not include this support.

The encapsulation rules allow a multi-method dispatching on a parent to access a private method of a child. This might be viewed as unwise, since it grants methods defined at the roots of the inheritance graph nearly unrestricted access to private methods. However, a practical need motivates this decision:

- An object should be granted access to its own private methods.
- A child should be allowed to override a private method inherited from a parent with its own version, perhaps also private.
- If an object has access to one of its own methods, it should continue to have access to a child's overriding version.*

For the parent's method to be able to invoke the child's overriding version of one of the parent's private methods, the parent must be granted access to the child's method.

Fortunately, we can limit the scope of an ancestor's privilege without sacrificing the ability to override private methods. The earlier encapsulation rule could be amended so that a parent would be granted access to a descendant's private method only if the child's method is overriding some other method to which the parent would have access, i.e., to a method defined on the parent or one of the parent's ancestors. Since in many cases the parent may not be able to provide a reasonable implementation, the parent can define a private abstract method to reserve privileged access to future concrete implementations provided by children. (Abstract methods were described in section 2.2.) Cecil currently does not include this extension, since Cecil is intended to support exploratory programming in addition to production programming and extra declarations just to allow private access may be too cumbersome, but Cecil may be changed in the future to include this extension.

4 Project Status and Future Work

The Cecil language project has several goals. One is to explore the practicality of multiple dispatching, in particular its integration with a traditional data-abstraction-oriented programming style. Another is to produce a purely object-oriented language that supports both exploratory programming and production programming, and supports gradual evolution of programs from one style to the other. Towards this end Cecil includes a flexible static type system and allows existing dynamically-typed Cecil programs to be annotated incrementally with declarations to gradually migrate more and more of the type checking to program-definition-time.

The design of Cecil's dynamically-typed subset is reasonably stable with a denotational specification of its semantics. Claudia Chiang is implementing an interpreter (in SELF) for this part of Cecil. As of this writing, the interpreter is roughly half complete, and we hope to have it finished by this summer. An initial design exists for Cecil's static type system, and type inference and checking rules have been written for the non-polymorphic core of the type checker. Stuart Williams is extending the Cecil interpreter to perform static type checking of Cecil programs.

After gaining programming experience using the interpreter and experimenting with alternative language designs, we plan to construct an efficient Cecil implementation. We first will extend the current SELF implementation to support Cecil-style multiple dispatching and encapsulation, augmenting the existing SELF compilation techniques [Chambers *et al.* 89, Hölzle *et al.* 91b, Chambers & Ungar 91, Chambers 92] with support

---

* Experience with SELF's encapsulation rules leads us to decide that a child should not be allowed to override a public method with a private one. Otherwise, subtle changes in calling code and consequently in privilege classification could lead to different methods being invoked.

for multiple dispatching. We expect that the bulk of the SELF implementation can remain unchanged, since it does not depend on the details of method lookup or even on whether the source language is based on single dispatching or multiple dispatching. This part of the effort should produce an efficient implementation of the dynamically-typed subset of Cecil. We then will extend this implementation to support static type checking to reach the full Cecil language. We also will explore programming environments for languages like Cecil, since we believe the presence of a supporting environment to be crucial for fostering a data-abstraction-oriented programming style.

5 Related Work

Much of the related research has already been discussed. CLOS [Bobrow *et al.* 88] and its predecessor CommonLoops [Bobrow *et al.* 86] pioneered the use of multiple dispatching. CLOS's inheritance rules are perhaps the most complex of any object-oriented language, based on linearization of the inheritance graph using parent and argument order to totally order multi-methods, thus avoiding any possibility of ambiguity in the method lookup at the cost of masking ambiguities introduced by programming errors. CLOS also provides `:before`, `:after`, and `:around` methods which automatically combine with primary methods to achieve interesting effects. In contrast, Cecil supports a comparatively simple multiple inheritance system, with no order defined on parents or arguments, that admits the possibility of ambiguity in method lookup; such ambiguities are viewed as programming errors that need to be reported to the programmer rather than silently resolved. CLOS also supports reflective capabilities [Kiczales *et al.* 91], an interesting area which Cecil does not address. Recently, some researchers have developed a static type system for CLOS-like languages [Agrawal *et al.* 91]. However, their type system does not handle ambiguous messages, abstract methods, or parameterized classes, nor does it provide a clean separation between types and classes. The Cecil type system provides these additional features.

Only a few languages outside the CLOS family support multiple dispatching. Kea is a polymorphic functional programming language with strong static typing [Mugridge *et al.* 91]. In addition to several interesting type system features, Kea includes multivariant functions (Kea's version of multi-methods) whose static and dynamic semantics is defined in terms of a translation into the lambda calculus. Leavens describes NOAL, a statically-typed functional language that supports overloaded functions that are resolved at run-time [Leavens 89, Leavens & Weihl 90]. This language was designed primarily as a vehicle for exploring formal verification of programs with subtyping. A theoretical treatment of typing issues by Rouaix included a similar toy language that supported run-time overloading of functions [Rouaix 90]. 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. Finally, a number of languages, including C++ [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.

Hebel and Johnson developed a special browser to manage the highly-stylized double dispatching code for arithmetic over numbers and matrices in Smalltalk-80 [Hebel & Johnson 90]. Their browser presents a two-dimensional spreadsheet-like view of all combinations of numeric and matrix argument types for a particular arithmetic message, with each entry reporting whether the corresponding argument type combination defines a method or merely inherits one from either the receiver inheritance chain or the argument inheritance chain. Programmers can manipulate implementations of arithmetic messages through the interface provided by the browser, and it will in turn generate many of the double dispatching functions automatically. While this browser makes double dispatching more manageable, it does not completely solve the problems with double dispatching. Users who examine numeric classes through the normal Smalltalk-80 browser still are confronted with a large number of (automatically-generated) dispatching routines. Additional browsers would be required for other kinds of messages (such as the `displayOn` message) to receive the same sorts of benefits. In effect, their browser partially simulates the functionality of multiple dispatching in the programming environment; we argue instead for uniform language support of multiple dispatching. Nevertheless, their interface might be useful even for a multiple-dispatching language such as Cecil to help display and organize multi-methods.

Finally, Cecil owes much to SELF [Ungar & Smith 87, Hölzle *et al.* 91a]. Cecil's classless object model and its uniform treatment of state and behavior are direct results of our experience with SELF. Cecil departs from SELF in several respects, some of which have been discussed in this paper: instance variables can be either local or shared, and new objects are created by refining rather than cloning existing objects. Of course, Cecil also extends SELF with multiple dispatching. Freeman-Benson independently developed a proposal for adding multi-methods to SELF [Freeman-Benson 89].

6 Conclusions

Cecil is intended to support a data-abstraction-oriented programming style typical of single-dispatching object-oriented languages despite its reliance on multi-methods as the basic mechanism for procedural abstraction. Other multiple-dispatching languages such as CLOS organize programs around generic functions and consequently foster a function-oriented programming style. Cecil instead organizes programs around objects and their connected operations which together implement abstract data types. Multi-methods are integrated into this object-centered programming model by treating a multi-method as part of each of the object implementations for which the multi-method dispatches. This approach also enables the internals of abstract data type implementations to be encapsulated. Multi-methods are granted privileged access to the private operations of all the objects of which the multi-methods are a part; multi-methods themselves may be marked as private. An interactive graphical programming environment can play an important role in reinforcing this object-oriented view of multi-methods by displaying the non-hierarchical relationships among objects and methods much better than can standard text-based environments. Other multiple-dispatching languages, including CLOS, could adopt this alternative view to better support data-abstraction-oriented programming.

The Cecil project is just beginning, with an interpreter under construction. More research and experience is needed to determine fully the practicality of multi-methods and the effectiveness of Cecil's linguistic mechanisms at fostering a data-abstraction-oriented programming methodology. One particularly crucial unanswered question is whether the additional expressive power of multiple-dispatching object-oriented languages outweighs the unavoidable increase in complexity of the language and the programming model. Another question is whether the encapsulation model included in the current Cecil language is adequate, or whether some extension or modification is needed to completely encapsulate data structures in the presence of multi-methods. Nevertheless, we are excited by the prospects for object-oriented multi-methods. We hope that this work at least will spark discussion of these prospects.

Acknowledgments

The ideas in this paper and their presentation 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, Gus Lopez, Michael Sannella, and Denise Draper, with the fledgling Cecil group including Claudia Chiang, Stuart Williams, and Christine Sweeney, and others including Peter Deutsch and Barbara Lerner. A conversation with Danny Bobrow and David Ungar at OOPSLA '89 provided the original motivation for this work.

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.

[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.

[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 92] 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.

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

[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.

[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.

[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.

[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.

[Hebel & Johnson 90] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and Double Dispatching in Smalltalk-80. In *Journal of Object-Oriented Programming 2(6)*, pp. 40-44, 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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.