Modular Statically Typed Multimethods

Todd Millstein and Craig Chambers

Department of Computer Science and Engineering
University of Washington
{todd,chambers}@cs.washington.edu

Abstract. Multimethods offer several well-known advantages over the single dispatching of conventional object-oriented languages, including a simple solution to the "binary method" problem, cleaner implementations of the "visitor," "strategy," and similar design patterns, and a form of "open objects." However, previous work on statically typed multimethods whose arguments are treated symmetrically has required the whole program to be available in order to perform typechecking. We describe Dubious, a simple core language including first-class generic functions with symmetric multimethods, a classless object model, and modules that can be separately typechecked. We identify two sets of restrictions that ensure modular type safety for Dubious as well as an interesting intermediate point between these two. We have proved each of these modular type systems sound.

1 Introduction

In object-oriented languages with multimethods (such as Common Lisp, Dylan, and Cecil), the appropriate method to invoke for a message send can depend on the run-time class of any subset of the message arguments, rather than a distinguished receiver argument. Multimethods unify the otherwise distinct concepts of functions, methods, and static overloading, leading to a potentially simpler language. They also support safe covariant overriding in the face of subtype polymorphism, providing a natural solution to the "binary method" problem [Bruce *et al.* 95] and a simple implementation of the "strategy" design pattern [Gamma *et al.* 95]. Finally, multimethods allow clients to add new operations that dynamically dispatch on existing classes, supporting a form of what we call "open objects" [Chambers 98] that enables easy programming of the "visitor" design pattern [Gamma *et al.* 95, Baumgartner *et al.* 96] and is a key element of aspect-oriented programming [Kiczales *et al.* 97]. Open objects also relieve the tension observed by others [Cook 90, Odersky & Wadler 97, Findler & Flatt 98] between ease of adding operations to existing classes and ease of adding subclasses.

A key challenge for multimethods is separate static typechecking: it is possible for two modules containing arbitrary multimethods to typecheck successfully in isolation but generate type errors when linked together. Previous work on statically typed multimethods has dealt with this problem either by forcing programs to be typechecked as a whole [Mugridge *et al.* 91, Castagna *et al.* 92, Chambers 92, Bourdoncle & Merz 97, Castagna 97, Chambers & Leavens 97, Leavens & Millstein 98] or by sacrificing symmetric treatment of multimethod arguments to ensure the safety of modular typechecking [Agrawal *et al.* 91, Bruce *et al.* 95, Boyland & Castagna 97].

We have designed Dubious, a simple core language supporting both symmetric multimethods and separately typechecked modules. We have identified two sets of restrictions on modules that achieve modular type safety, representing the endpoints in a range of modular type systems trading off flexibility and expressiveness for modularity. We also identify an intermediate point in this range that provides programmers with fine-

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 279-303, 1999.

grained control over this tradeoff. We have proved all three of these modular type systems sound.

Section 2 describes the language's features, along with a simple global typechecking algorithm. Section 3 presents a set of important programming idioms that we use as expressiveness benchmarks for our various modular typechecking algorithms. Section 4 focuses on the challenges and solutions for modular typechecking, describing the two main sets of restrictions for modular type safety as well as an interesting intermediate point. Section 5 sketches several extensions to Dubious that are necessary for a practical language. Section 6 discusses related work, and section 7 concludes. A separate technical report provides formal dynamic and static semantics for Dubious, as well as soundness proofs for the various type systems presented [Millstein & Chambers 99].

2 The Dubious Language

The design of Dubious is focused on the issue of modular typechecking of symmetric multimethods; we consciously omit many useful but less relevant features. Dubious includes:

- a classless object model with explicitly declared objects and inheritance (but not dynamically created objects and state),
- first-class generic functions (but not lexically nested closures),
- explicitly declared function types (but not explicitly declared object types nor subtyping independent of objects and inheritance[*]), and
- modules to support separate typechecking and namespace management (but not nested modules, parameterized modules, nor encapsulation mechanisms).

Section 5 sketches how Dubious could be extended to handle many of these omissions.

Dubious's syntax appears in figure 1; brackets delimit optional parts of the syntax. The following subsections present Dubious's semantics and typechecking rules informally.

2.1 Informal Semantics

The **`object`** declaration creates a fresh object with a unique, statically known identity and binds it to the given name. The declaration also names the objects from which the new object inherits (its parents). The *descends* relation among objects is the reflexive, transitive closure of the declared inheritance relation. As in other classless languages [LaLonde *et al.* 86, Lieberman 86, Ungar & Smith 87, Chambers 92, Abadi & Cardelli 96], objects play the roles of both classes and instances, and **`isa`** accordingly plays the

P ::= M_1 ... M_n **`import`** I **`in`** E **`end`**

M ::= **`module`** I **`imports`** $I_1, \ldots, I_m$ `{` D_1 ... D_n `}`

D ::= [**`abstract`** | **`interface`**] **`object`** I **`isa`** $O_1, \ldots, O_n$
| I **`has method`**$(F_1, \ldots, F_n)$ `{` E `}`

E ::= I | $E_0(E_1, \ldots, E_n)$

O ::= I | $(O_1, \ldots, O_n) \rightarrow O$

F ::= I_1 [`@` I_2]

I ::= *identifier*

Figure 1: Syntax of Dubious

[*] An earlier version of this work separated objects and types, but we removed this feature to simplify the presentation of the language and its semantics.

```
module GraphicMod imports StdLibMod {
  abstract object graphic
  object draw isa (graphic, display)→void
}
module PointMod imports GraphicMod {
  object point isa graphic
  object x isa (point)→int
  x has method(p@point) { one }
  object y isa (point)→int
  y has method(p@point) { two }
  draw has method(p@point, d) { … }
  object equal isa (point, point)→bool
  equal has method(p1@point, p2@point) {
    and(=(x(p1), x(p2)), =(y(p1), y(p2))) }
}
module ColorPointMod imports PointMod, ColorMod {
  object colorPoint isa point
  object col isa (colorPoint)→color
  col has method(cp@colorPoint) { red }
  draw has method(cp@colorPoint, d) { … -- draw in color }
  equal has method(cp1@colorPoint, cp2@colorPoint) {
    and(and(=(x(cp1), x(cp2)), =(y(cp1), y(cp2))),
        eq_color(col(cp1), col(cp2))) }
}
module OriginMod imports PointMod {
  object origin isa point
  x has method(o@origin) { zero }
  y has method(o@origin) { zero }
  equal has method(o1@origin, o2@origin) { true }
}
```

Figure 2: A simple Dubious program fragment

roles of both inheritance and instantiation. For example, in the modules of figure 2, the `colorPoint` object represents a subclass of `point` that has a color, while the `origin` object represents a particular `point` instance.

An object can also act like a (generic) function by inheriting from an *arrow object* that defines the legal arguments and return values of the function. For example, the `equal` object declared in the `PointMod` module in figure 2 inherits from the `(point, point)→bool` arrow object, specifying that `equal` can act like a generic function accepting two objects, each of which is `point` or a descendant, and returning `bool` or a descendant. The ordinary contravariant subtyping rule for function types [Cardelli 84] is used as the descends relation among arrow objects. For simplicity, we require that every generic function have a unique most-specific arrow object from which it inherits. We refer to this most-specific arrow object as the arrow object of the generic function.

A generic function is implemented by adding methods to it, via the **`has method`** declaration. In the `PointMod` module example in figure 2, the method added to the `draw` object has two formal parameters, named `p` and `d`. The first formal is *specialized* by providing the `@point` suffix, which specifies the object on which the argument is dynamically dispatched. We require that a specializer object on a method descend from the object in the corresponding position of the associated generic function's arrow object. For uniformity, unspecialized formals are treated as if they specialize on the object in the corresponding position of the generic function's arrow object. Therefore, the second formal in the `draw` method implicitly specializes on the `display` object. While this method is (explicitly) specialized only on its first argument, the method added

to the `equal` object is a true multimethod, dynamically dispatching on both of its arguments. The **`has method`** declaration is an imperative, side-effecting operation, like the method update construct in the calculi of Abadi and Cardelli [Abadi & Cardelli 95, Abadi & Cardelli 96]. For example, the `ColorPointMod` module adds a second method to the `draw` object created in the `GraphicMod` module.

If an object is declared **`abstract`**, it is used solely as a template for other objects and may not be referred to in expressions. For example, the abstract `graphic` object in figure 2 is a template for objects that can be drawn, which is then implemented by the `point` object and descendants. Methods can be specialized on abstract objects, allowing such objects to be partially implemented. These methods are then inherited for use by descendants. An **`interface`** object is an abstract object that additionally may not act as a specializer (although it may be an implicit specializer on an unspecialized position, as described above). This is similar to the interface construct in Java [Arnold & Gosling 98, Gosling et al. 96]. Arrow objects are implicitly treated as interface objects. (Distinguishing between abstract and interface objects will become useful when considering modular typechecking algorithms in section 4.) An object that is neither abstract nor an interface is called *concrete*. (A *non-concrete* object is either abstract or an interface, and similarly for *non-abstract* and *non-interface*.)

To evaluate a generic function application (message send) $E_0(E_1, \ldots, E_n)$, we evaluate each expression E_i to some object o_i, extract the methods that were added to the generic function object o_0, and then select and invoke the *most-specific applicable method* for $(o_1, \ldots, o_n)$. A method in o_0 is *applicable* to $(o_1, \ldots, o_n)$ if the method has n arguments and if $(o_1, \ldots, o_n)$ pointwise descends from the tuple of the method's specializers. The *most-specific* applicable method is the unique applicable method whose specializers pointwise descend from the specializers of every applicable method. If there are no applicable methods, a *message-not-understood* error occurs, while if there are applicable methods but no most-specific one, a *message-ambiguous* error occurs. For example, consider the application `equal(colorPoint, colorPoint)`, evaluated in some context that imports all the modules in figure 2. First, the `equal` and `colorPoint` expressions are evaluated, yielding the objects with those names.* Then the methods that were added to the `equal` object are extracted. Of the three methods, the `(@point, @point)` and `(@colorPoint, @colorPoint)` methods are applicable, and the `(@colorPoint, @colorPoint)` method is the most-specific one. Therefore, this most-specific applicable method is selected and invoked.

One module **`imports`** another in order to access its objects. The import relation is transitive; for example, the `ColorPointMod` module may refer to objects in the `StdLibMod` module.

A program is simply an expression evaluated in the context of a single module. (Restricting this top-level expression to import a single module is no loss of expressiveness, as that module can import as many modules as are needed.)

2.2 Static Type Checking

Dubious's static type system ensures that legal programs do not have message-not-understood or message-ambiguous errors. Ruling out these errors involves two kinds of checks: *client-side* and *implementation-side* [Chambers & Leavens 95]. Client-side

* A different object could be named `equal` in a different scope, so this application expression would invoke a different generic function in that other scope. Message names are not special, but are simply identifiers that evaluate to some object via regular scoping rules. The explicit distinction in Dubious between introducing a new generic function and adding a method to an existing generic function clarifies a number of issues, such as overriding versus overloading, that are often confusing in traditional object-oriented languages lacking explicit generic function declarations.

checks are local checks on declarations and expressions. The most important of these checks relate to invoking and implementing generic functions. For each message send expression $E(E_1, \ldots, E_n)$ in the program, we check that E descends from an arrow object $(O_1, \ldots, O_n) \rightarrow O$ and each E_i descends from O_i. The message send expression is then known to yield a value that descends from O. For each method declaration I **has method**$(I_1 \texttt{ @ } I_1', \ldots, I_n \texttt{ @ } I_n')$ { E } in the program, we check that I descends from an arrow object $(O_1, \ldots, O_n) \rightarrow O$, each I_i' descends from O_i, and E descends from O when typechecked in an environment where each I_i is known to descend from I_i'.

Implementation-side checks ensure that each concrete generic function o correctly implements its arrow object $(O_1, \ldots, O_n) \rightarrow O$. For every tuple of concrete objects $(o_1, \ldots, o_n)$ such that each o_i descends from O_i, there must exist a most-specific applicable method in o. For example, consider implementation-side checks on the `equal` generic function. `equal` is declared to inherit from `(point, point)`→`bool`. Nine concrete argument tuples pointwise descend from (`point`, `point`): all possible pairs of the objects `point`, `colorPoint`, and `origin`. The (`@colorPoint`, `@colorPoint`) method is most-specific for two `colorPoint`s, the (`@origin`, `@origin`) method is most-specific for two `origin`s, and the (`@point`, `@point`) method is most-specific for all other tuples.

This description suggests a straightforward typechecking algorithm. Client-side checks on the declarations in a module require only inheritance information from imported modules, and they can therefore be performed on a module-by-module basis. The implementation-side checks as described above assume global knowledge of the program's objects and methods. Therefore, implementation-side typechecking on all generic functions is deferred until link-time, when all modules are present. This link-time typechecking approach is used for typechecking previous languages with symmetric multimethods, including Kea [Mugridge *et al*. 91], Cecil [Chambers 92, Chambers 95], $ML_{\leq}$ [Bourdoncle & Merz 97], and Tuple [Leavens & Millstein 98]. We refer to this global typechecking algorithm as *System G*.

3 Example Programming Idioms

There are several flexible programming idioms that we would like to be able to express and statically typecheck in Dubious. These idioms will serve as benchmarks for evaluating the various typechecking algorithms presented in this paper. The global typechecking of System G supports all these idioms.

First, we wish to support traditional receiver-oriented programming. In this idiom, an object is declared with its associated methods in its own module, which imports the modules defining the parent objects. More generally, we can allow multiple objects and their associated methods to be declared in a single module. We refer to this idiom, in which each method is specialized solely on its first argument to an object declared in the same module, as *single dispatching*.

We also wish to allow a module to define abstract objects, whose operations are not required to be implemented, and to provide concrete implementations of these objects in separate modules. For example, the `GraphicMod` module in figure 2 defines an abstract template for graphical objects. Because the `graphic` object is abstract, its `draw` generic function need not be completely implemented. Concrete descendants of the `graphic` object can be declared in other modules, which must provide appropriate implementations for the generic functions declared in the `GraphicMod` module. For example, the `point` object is a legal implementation of `graphic`. Clients of the abstract object need not be aware of the various concrete implementations.

Several expressive idioms exploit multimethods. If single dispatching is generalized to allow method arguments in addition to the first to be specialized on objects declared in the enclosing module, *all-local (multi)methods* result, which enable a simple solution to the "binary method" problem [Bruce *et al.* 95]. For example, the three `equal` methods in figure 2 are all-local multimethods, each specialized on two objects declared in the enclosing module. All-local multimethods allow easy programming of one reasonable semantics for equality on various combinations of points. At the same time, `colorPoint` and `origin` remain safe subtypes of `point`, so subtype polymorphism over the `point` hierarchy is still available.

A further generalization of single dispatching and all-local multimethods allows arguments other than the first to be specialized on any object, including objects declared in imported modules; the first argument is still restricted to be specialized on a locally declared object. We refer to this kind of multimethod as an *encapsulated-style multimethod*; it is similar to the style of multimethods allowed by encapsulated multimethods [Castagna 95, Bruce *et al.* 95] and parasitic multimethods [Boyland & Castagna 97]. Using encapsulated-style multimethods, new objects can interact with imported objects in interesting ways, without modifying the code for the imported module. For example, the `ColorPointMod` module could include the following declaration to program how colored points should be compared to uncolored points:

```
equal has method(cp@colorPoint, p@point) { … }
```

If we remove all restrictions on method specializers, allowing any or all to be imported objects, we obtain what we call *arbitrary (multi)methods*. Using arbitrary multimethods, we can program special behavior for equality on one `point` and one `colorPoint`, without modifying either the `PointMod` or `ColorPointMod` modules, with the following code:

```
module PCPMod imports ColorPointMod {
  equal has method(p@point, cp@colorPoint) { … }
}
```

A final desirable idiom is *open objects* [Chambers 98], where an object declared in an imported module is extended by adding new operations that dispatch on the object, without modifying any existing modules. For example, the following code introduces a module that defines a `lineseg` object representing a line segment. The `distance` generic function extends the `point` object and descendants with a new operation, without modifying existing code.

```
module LineSegMod imports OriginMod {
  object lineseg
  object p1 isa (lineseg)→point
  p1 has method(ls@lineseg) { … }
  object p2 isa (lineseg)→point
  p1 has method(ls@lineseg) { … }
  object distance isa (point, lineseg)→real
  distance has method (p@point, ls) {
    … -- distance from a point to a line segment }
  distance has method (p@origin, ls) {
    … -- faster algorithm for the origin }
}
```

Open objects arise naturally in languages based on multimethods, but they are useful even for singly dispatched methods, as in the case of the `distance` generic function above. Open objects support simpler programming of the "visitor" design pattern [Gamma *et al.* 95]. Client-specific visitors are programmed directly, without needing to build a special visitor infrastructure for each object hierarchy. In addition, the open

object idiom retains the ability to add new subclasses without modifying existing code, while the visitor pattern does not. In the `lineseg` example, the `distance` generic function is a kind of client-specific visitor of the `point` hierarchy. New descendants of `point` can be added in other modules without modifying or breaking the `distance` method (as long as the implementations of `distance` inherited by new descendants are still appropriate), and new operations can be added to existing object hierarchies without modifying or breaking those hierarchies. Aspect-oriented programming [Kiczales *et al.* 97] relies heavily on open objects to implement cross-cutting concerns.

4 Modular Typechecking

We would like a modular approach to implementation-side typechecking of generic functions, unlike the global algorithm of System G. In particular, we would like a typechecking scheme with the following properties for each module:

- The generic functions created in the module can be safely implementation-side typechecked given only the interfaces of imported modules: their objects, associated inheritance hierarchy, and the headers of each generic function's methods.
- Implementation-side typechecking of an imported generic function is only necessary when the importer adds a new method to the generic function, or when the importer creates a concrete object directly inheriting from a non-concrete object, and the concrete object is applicable to one of the generic function's argument positions. Further, this checking does not re-examine legal argument tuples already checked by the importee.

This typechecking scheme is a generalization of the modular typechecking scheme of conventional statically typed, singly dispatched object-oriented languages. Unfortunately, applying such a modular typechecking scheme to the unrestricted Dubious language is unsound. It is possible for two importers of a module to pass these checks in isolation but still cause message-not-understood or message-ambiguous errors when combined in a single program [Chambers & Leavens 95]. Subsection 4.1 describes the situations that can lead to such errors. Subsections 4.2, 4.3, and 4.4 describe three different sets of restrictions that support safe modular typechecking, representing different tradeoffs between expressiveness, modularity, and complexity. Subsection 4.5 summarizes the key features of the various type systems discussed in this paper.

4.1 Challenges for Modular Typechecking

The unrestricted Dubious language poses several problems for modular typechecking. In particular, there are four scenarios where a modular typechecking approach applied to the unrestricted language will fail to statically detect errors that can occur at run-time. The first two problems are specific to multimethods, while the other two involve open objects. In this subsection, we give examples of each of these kinds of problems.

First, the ability to add arbitrary multimethods anywhere in the program can cause undetected ambiguities. A simple example of the problems that can occur appears in figure 3, where each of the `ColorPointMod` and `OriginMod` modules from figure 2 is augmented with a second `equal` method. Each module typechecks in isolation, given only information about its imported modules. From the `ColorPointMod` module's point of view, every visible legal argument tuple to the `equal` generic function has a single, most-specific method implementation, and similarly for the `OriginMod` module. However, when the two modules are combined in a single program, a run-time message-ambiguous error will occur if the message `equal(colorPoint,origin)` is ever sent, since neither of the methods in the example is more specific than the other.

```
module ColorPointMod imports PointMod, ColorMod {
  ... -- as in figure 2
  equal has method(cp@colorPoint, p@point) { … }
}
module OriginMod imports PointMod {
  ... -- as in figure 2
  equal has method(p@point, o@origin) { … }
}
```

Figure 3: Ambiguity problems with arbitrary multimethods

```
module AbstractPointMod {
  abstract object point
  object equal isa (point, point)→bool
}
module ColorPointMod imports AbstractPointMod, ColorMod {
  object colorPoint isa point
  equal has method(cp1@colorPoint, cp2@colorPoint) { … }
}
module OriginMod imports AbstractPointMod {
  object origin isa point
  equal has method(o1@origin, o2@origin) { true }
}
```

Figure 4: Incompleteness problems with multimethods on abstract objects

```
module PrintMod imports ColorPointMod, OriginMod {
  object print isa (point)→void
  print has method(p@point) { ... -- print points }
  print has method(cp@colorPoint) { ... -- print in color }
  print has method(o@origin) { ... -- print the origin specially }
}
module ColorOriginMod imports ColorPointMod, OriginMod {
  object colorOrigin isa colorPoint, origin
}
```

Figure 5: Ambiguity problems combining open objects with multiple inheritance

```
module EraseMod imports PointMod {
  object erase isa (graphic, display)→int
  erase has method(p@point, d) { ... -- erase points }
}
module MyGraphicMod imports GraphicMod {
  object myGraphic isa graphic
  draw has method(g@myGraphic, d) { … }
}
```

Figure 6: Incompleteness problems combining open objects with abstract objects

One way to solve this problem is to break the symmetry of the dispatching semantics. For example, if we linearized the specificity of argument positions, comparing specializers lexicographically left-to-right (rather than pointwise) as is done in Common Lisp [Steele 90, Paepcke 93] and Polyglot [Agrawal *et al.* 91], then the `(@colorPoint, @point)` method would be more specific than the `(@point, @origin)` method. However, one of our major design goals for Dubious is to retain the symmetric multimethod dispatching semantics, which we believe is more natural and less error-prone, since it reports potential ambiguities rather than silently resolving them. The symmetric semantics is used in the languages Cecil [Chambers 92, Chambers 95], Dylan [Shalit 97, Feinberg *et al.* 97], the λ&-calculus [Castagna *et al.* 92, Castagna 97],

Kea [Mugridge *et al*. 91], $ML_{\leq}$ [Bourdoncle & Merz 97], and Tuple [Leavens & Millstein 98].

A second unsafe scenario involves the combination of non-concrete objects with multimethods. For example, suppose we want to make the `point` object in figure 2 be abstract, so that it need not be fully implemented. Figure 4 shows the relevant parts of the revised modules. Each module passes implementation-side typechecks on the `equal` generic function in isolation, as all legal tuples of concrete objects have a single, most-specific implementation from each module's point of view. However, at run-time a message-not-understood error will occur if the message `equal(colorPoint, origin)` or `equal(origin, colorPoint)` is sent.

The last two unsafe scenarios involve the ability to program open object idioms. Figure 5 shows an example of the problems that can occur when the open object idiom is combined with multiple inheritance. The `PrintMod` module extends the interface of points from figure 2 with a function for printing points. From this module's point of view, the `print` generic function is completely and unambiguously implemented. Independently, the `ColorOriginMod` module creates a new point object that multiply inherits from `colorPoint` and `origin`. Since this module does not know about the `print` generic function, it has no way of statically detecting the ambiguity for `print(colorOrigin)`, which can therefore cause a run-time error.

One way to fix this ambiguity is to linearize the inheritance hierarchy, as is done in Common Lisp [Steele 90, Paepcke 93] and Dylan [Shalit 97, Feinberg *et al.* 97], or to provide some other total ordering over methods, as in parasitic multimethods [Boyland & Castagna 97]. However, we reject these solutions for reasons similar to our rejection of argument-position linearization for multimethods, preferring the simpler and less error-prone semantics.

The final unsafe scenario involves the combination of open object idioms with non-concrete objects. In figure 6, the `EraseMod` module extends the `graphic` interface from figure 2 with an operation for erasing the graphical object. The `erase` generic function is completely implemented for all the concrete implementations of `graphic` that are visible to that module. The `MyGraphicMod` creates a concrete implementation of a `graphic` object. From its point of view, the `graphic` interface is completely implemented for `myGraphic`. However, at run-time there will be a message-not-understood error if the message send `erase(myGraphic, d)` occurs (where `d` is some `display` descendant).

4.2 System M: Maximizing Modularity

Because of the four unsafe programming scenarios described above, any completely modular typechecking scheme must restrict the usage of certain Dubious language constructs. In this subsection, we detail *System M*, a set of restrictions that allows a modular typechecking scheme with the desirable properties described at the beginning of this section. Our goal with System M is to provide the most flexible type system possible, subject to those strict modularity goals.

We say that a non-arrow object or a method is *local* if it is declared in the current module, and otherwise it is *non-local*. An arrow object is local if it has a local object in a positive[*] position, and otherwise it is non-local. Two objects are *related* if one object descends

[*] The result object of an arrow object is in a positive position, while the arguments are in negative positions. If an arrow object appears in a negative position, then the polarity of its positions is reversed [Canning *et al.* 89]. An arrow object with a local object in a positive position is local because no module other than importers of the current module can define functions that descend from the arrow. Due to contravariance, the same is not true if local objects appear only in negative positions.

from the other, and otherwise they are *unrelated*. Similarly, two modules are related if one imports the other, and otherwise they are unrelated. An *orphan* is a concrete object that directly inherits from a non-local, non-concrete object. *Implementation inheritance* is inheritance from a non-interface object, and *interface inheritance* is inheritance from an interface object.

The key insight of System M is that if two unrelated modules M_1 and M_2 each create an encapsulated-style method on the same generic function and multiple implementation inheritance across module boundaries is disallowed, then the two methods will apply to disjoint sets of legal argument tuples. Therefore, these restrictions safely remove method ambiguity problems. Potential incompleteness problems are removed by treating visible non-concrete objects as if they were concrete during the implementation-side typechecking of a generic function, thereby forcing the existence of appropriate method implementations to handle unseen concrete descendants of these objects. We treat all visible non-concrete objects in this way except for local non-concrete objects that apply to the first argument of the generic function. Such objects can remain safely unimplemented, with appropriate implementations for concrete descendants to be added by importers, thereby safely allowing abstract object idioms.

More precisely, System M imposes the following four restrictions on each module:

- (***M1***) Each method added to a non-local generic function must be an encapsulated-style method, i.e., the first argument must be specialized to a local object. (Strictly speaking, all that is required is that each generic function designate some argument position for which all methods of that generic function agree to have a local specializer; for simplicity in this paper, we assume this designated position is the first.)
- (***M2***) If a local non-interface object descends from two unrelated, non-local, non-interface objects o_1 and o_2, then it must also descend from a non-interface object that descends from o_1 and o_2 as well.
- (***M3***) If a generic function's arrow object has the object o in some argument position other than the first, then implementation-side typechecks of the generic function must consider any non-concrete visible descendants* of o to be concrete.
- (***M4***) If a generic function's arrow object has the object o in the first argument position, then implementation-side typechecks of the generic function must consider any non-local, non-concrete visible descendants of o to be concrete.

By imposing the above restrictions, we can ensure safety in Dubious while meeting the modularity goals described at the beginning of this section. In particular, each module implementation-side typechecks its local generic functions given only the interfaces of its importees. In addition, there are two scenarios in which a module must re-check imported generic functions:

- If the module adds methods to a non-local generic function, then this generic function is implementation-side typechecked. However, we only need to check argument tuples to which a local method applies.
- If the module creates an orphan o, then all non-local generic functions accepting o as an argument in the first position are implementation-side typechecked. However, only legal argument tuples containing o at that position need be checked. (This check ensures the safety of local, non-concrete descendants of the first argument in a generic function's arrow object, which was left unchecked by rule ***M4*** above. In this way, we safely allow abstract object idioms.)

* Recall that the *descends* relation is the reflexive, transitive closure of the declared inheritance relation.

We can use the restrictions to resolve the problems in the examples of the previous subsection. In figure 3, the `equal` method in the `OriginMod` would cause a static type error. In particular, it is not an encapsulated-style method because it has a non-local first specializer, so it violates restriction ***M1***. The restriction would be satisfied if the method instead had the form

```
equal has method(o@origin, p@point) { … }
```

and indeed this removes the multimethod ambiguity. Forcing both methods in figure 3 to be encapsulated-style (in conjunction with the multiple inheritance limitations imposed by restriction ***M2***) ensures that no argument tuple will be applicable to both of them.

In figure 4, both the `colorPoint` and `origin` objects are orphans. Because `equal` accepts the `colorPoint` and `origin` objects on its first position, it is re-checked by the `ColorPointMod` and `OriginMod` modules. By restriction ***M3***, these checks must consider the abstract `point` object to be concrete for the second argument position. Therefore, re-checks from the `ColorPointMod` module will find an incompleteness for the argument tuple (`colorPoint,point`), and similarly for the `OriginMod` module's checks. Therefore, methods must be created to cover these cases safely. For example, the `ColorPointMod` module could include the method declaration

```
equal has method(cp@colorPoint, p@point) { … }
```

and similarly for the `OriginMod` module, thereby safely allowing abstract object idioms.

In figure 5, the `colorOrigin` object fails restriction ***M2*** because it descends from two unrelated, non-local, non-interface objects without also descending from some non-interface descendant of both of these objects, so the `colorOrigin` object cannot be programmed. Because of the conflict between cross-module multiple implementation inheritance and open object idioms, we were forced to eliminate one of these idioms to ensure the safety of modular checking. We chose to disallow cross-module multiple implementation inheritance because of the importance of open objects. Multiple interface inheritance across import boundaries is still safe (because methods cannot specialize on interface objects and so cannot generate ambiguities), as is arbitrary multiple implementation inheritance within a module.

In addition, many cases of multiple implementation inheritance can be programmed safely within System M's restrictions if they are *anticipated* when at least one of the parents is implemented. For example, if the implementor of the `OriginMod` module anticipated that multiple inheritance between `colorPoint` and `origin` might be needed, then a placeholder abstract object could be added to `OriginMod` that multiply inherits from the two objects:

```
module OriginMod imports ColorPointMod {
  object origin isa point
  ...
  abstract object colorPointAndOrigin isa colorPoint, origin
}
```

Clients that wish to use multiple inheritance can then singly inherit from the placeholder object:

```
module ColorOriginMod imports OriginMod {
  object colorOrigin isa colorPointAndOrigin
}
```

Any modules that add new generic functions to existing objects, such as the `PrintMod` module in figure 5, will see the placeholder object whenever they see both its parents, so

they will be forced to write methods that resolve any potential multiple-inheritance ambiguities. In this example, restriction ***M4*** would force the existence of a `print` method to disambiguate printing a `colorPointAndOrigin`. This addition fixes the potential message-ambiguous error in the example. In our experience, gained largely from writing a 125,000-line optimizing compiler in Cecil, anticipated multiple implementation inheritance is common, while successful unanticipated multiple implementation inheritance is very rare.

In figure 6, the first argument of the `erase` generic function's arrow object is the non-local abstract `graphic` object. By restriction ***M4***, implementation-side typechecks on this generic function must assume that `graphic` is concrete. Therefore, an incompleteness is found, which is fixed by adding an appropriate default method implementation. Adding such a method allows the module to typecheck and ensures that the message send `erase(myGraphic, d)` now has a single, most-specific method implementation to invoke.

In summary, System M provides a completely modular and safe typechecking algorithm, while maintaining a high level of flexibility. Multimethods with a non-local first specializer are disallowed, but all other kinds of multimethods are safe. This provides several important multimethod idioms, including binary methods and encapsulated-style multimethods. The implementation of a non-concrete object across module boundaries and the use of arbitrary open objects are allowed, as long as the appropriate default method implementations are included to prevent unseen incompletenesses. The cost of allowing the open object idioms is a loss of unanticipated cross-module multiple implementation inheritance.

4.3 System E: Maximizing Expressiveness

Although the set of restrictions in System M provides completely modular typechecking for Dubious, there are certain idioms that cannot be expressed across module boundaries. These missing idioms include arbitrary multimethods, where the first argument position has a non-local specializer, and unanticipated multiple implementation inheritance. In this subsection, we present *System E*, whose fundamental requirement is to be able to express all idioms. Since a completely modular static type system cannot support all these idioms safely, System E includes a simple link-time check to ensure soundness of the more aggressive idioms, while striving to retain modular typechecking for as many idioms as possible.

A previous work informally introduced the idea of a module *extending* another instead of importing it [Chambers & Leavens 95] whenever the module needed to make use of one of the more aggressive idioms across that module boundary. If each module rechecks all the generic functions from modules it extends, and if at link-time each module in the program has a *unique most-extending module* in the reflexive, transitive closure of the direct module extends relation, then the program is guaranteed to be safe. In particular, the most-extending module's checks are enough to ensure run-time safety of all the modules it (directly or indirectly) extends.

We have formalized this notion of module extension in Dubious. We modify the module declaration to allow a set of extenders to be declared:

M ::= **module imports** $I_1, \ldots, I_m$ **extends** $I_1', \ldots, I_r'$ { D_1 ... D_n }

In the context of System E, we say that a non-arrow object or a method is *local* if it was declared in the current module, *extended* if it was declared in an extended module, and *imported* otherwise. An arrow object is local if it has a local object in a positive position, otherwise it is imported if it has an imported object in a positive position, and otherwise it is extended. A *non-local* object is either imported or extended, and similarly for a *non-*

imported and *non-extended* object. A module m_1 is the most-extending module of m_2 if m_1 extends every module that extends m_2, using the reflexive, transitive closure of the module declarations' **`extends`** clauses.

A module is unrestricted in its interactions with extended objects and generic functions, and thus all of our benchmark idioms are available to extenders. However, because of the greater potential effects of unseen extenders, importers must obey stricter restrictions than those of System M. First, because extenders can use multiple implementation inheritance freely, open object idioms must be disallowed in importers. Second, because extenders can write arbitrary multimethods, we restrict modules to add only all-local methods (not just encapsulated methods) to imported generic functions.

If all methods added to imported generic functions were forced to be all-local, then very few idioms could be legally programmed in importing modules. To eliminate this problem, we introduce syntax for specifying, as part of a generic function declaration, which subset of its arguments may be specialized:

O ::= I | $(S_1, \ldots, S_n) \rightarrow O$
S ::= [#]O

A **#** marker denotes a *marked argument position* of a generic function. All arrow objects a generic function inherits must agree on which argument positions are marked. Methods may not specialize at unmarked argument positions. This allows methods on imported generic functions to safely accept non-local objects on unmarked positions. Methods on imported generic functions must still have only local specializers on marked positions, as in an all-local method.

To avoid incompleteness, we use the same technique as in System M of considering all visible non-concrete objects to be concrete for the purposes of implementation-side typechecking a generic function. As with System M, there is one situation in which non-concrete objects may safely remain incompletely implemented. In particular, if the generic function has exactly one marked argument position, then any local non-concrete objects that apply to the marked position need not be considered concrete during implementation-side typechecking. This safely allows abstract object idioms for singly dispatched functions.

More formally, System E imposes the following four restrictions on modules, directly analogous to the four restrictions of System M:

- (***E1a***) A method may not specialize on an unmarked argument position of its generic function, and (***E1b***) a method added to an imported generic function must have a local specializer object at each marked argument position.
- (***E2***) If a local non-interface object *o* descends from an imported object, then every two non-interface parents of *o*, where at least one of the parents is non-local, must be related.
- (***E3a***) If a generic function's arrow object has the object *o* in some unmarked argument position, then implementation-side typechecks of the generic function must consider any non-concrete visible descendants of *o* to be concrete. (***E3b***) If a generic function's arrow object has the object *o* in some marked argument position and the generic function is not singly dispatched, then implementation-side typechecks of the generic function must consider any non-concrete visible descendants of *o* to be concrete.
- (***E4a***) If a local generic function's arrow object has the object *o* in a marked argument position, then no visible descendant of *o* may be an imported object. (***E4b***) If a singly dispatched generic function's arrow object has the object *o* in its marked

```
module ResolveColorPointAndOriginMod extends ColorPointMod,
                                              OriginMod {
    equal has method(cp@colorPoint, o@origin) { … }
}
module ResolvePrintModAndColorOriginMod extends PrintMod,
                                                 ColorOriginMod
    print has method(co@colorOrigin) { … }
}
```

Figure 7: Extension Modules

argument position, then implementation-side typechecks of the generic function must consider any non-local, non-concrete visible descendants of *o* to be concrete.

To typecheck a module, we need only the interfaces of extended and imported modules. A module re-implementation-side typechecks extended generic functions. In addition, there are two kinds of re-checks on imported generic functions, similar to the two cases in System M:

- If the module adds methods to an imported generic function, then this generic function is implementation-side typechecked. However, we only need to check argument tuples to which a local method applies.
- If the module creates an orphan *o*, then all imported, singly dispatched generic functions accepting *o* as an argument on the marked position are implementation-side typechecked. However, only legal argument tuples containing *o* at that position need be checked. (This check ensures the safety of local, non-concrete descendants of the argument in the marked position in a singly dispatched generic function's arrow object, which was left unchecked by rule ***E4b*** above. In this way, we safely allow abstract object idioms for singly dispatched generic functions.)

A link-time check ensures that every module has a unique, most-extending module. This is the only global check needed by System E, and it does not include any client-side or implementation-side typechecking; it merely ensures that modules exist that have already performed the necessary checks in their modular fashion. This check takes time $O(m + e)$, where m is the number of modules and e is the number of extends declarations in the program.

The restrictions on importers, while stronger than those of System M, still allow safe modular typechecking of singly dispatched programming idioms, the implementation of an imported non-concrete object for singly dispatched functions, and binary methods. For example, each of the modules in figure 2 satisfies these restrictions (assuming every generic function's first argument position is marked, and `equal` also has a marked second argument position). In particular, `point` is a safe implementation of the abstract `graphic` object, even though the `draw` method is not implemented for `graphic` objects and there are potentially other implementations of `graphic` that are unseen by the `PointMod` module. In addition, binary methods like `equal` are allowed by these rules, even though the `ColorPointMod` and `OriginMod` modules cannot see each other statically. For example, to ensure safety of the `equal` generic function, the `ColorPointMod` module simply needs to check that the (`colorPoint`, `colorPoint`) tuple has a most-specific method; the (`point`, `point`), (`colorPoint`, `point`), and (`point`, `colorPoint`) tuples need not be checked. The `OriginMod` module will perform analogous checks.

However, the more expressive programming idioms must use extenders, which can now resolve the problems from subsection 4.1 that System M could not solve. In figure 3, the revised `ColorPointMod` and `OriginMod` modules must use the clause **`extends`** `PointMod` instead of **`imports`** `PointMod` because they each violate restriction ***E1b***

by having an imported specializer on the `equal` method (the `ColorPointMod` module still need only import the `ColorMod` module). Further, if the two modules are combined in the same program, an additional module must be written that extends them both, in order for the `PointMod` module to have a unique most-extending module. In order for checks on the `equal` generic function to succeed in this new module, it must include the necessary declarations to fix the ambiguity, as shown in the `ResolveColorPointAndOriginMod` module in figure 7.

An analogous solution is used to resolve the problem illustrated in figure 5. Assuming the `print` generic function's first argument position is marked, `print` violates restriction ***E4a*** by having an imported object on a marked argument position in its arrow object, and `colorOrigin` violates restriction ***E2*** by multiply inheriting from unrelated, imported, non-interface objects. Therefore, both modules must extend the `PointMod` module rather than import it, requiring the existence of a most-extending module to fix the ambiguity. This module is shown in figure 7. Further, if the modules in figure 7 were combined in a single program, a module extending both of these would be required, in order to provide the `PointMod` module with a single most-extending module. (Because there are no conflicts, this module could be empty.)

In figure 4, the `equal` generic function in the `AbstractPointMod` module has multiple marked positions (assuming both argument positions are marked) and has the abstract `point` object on a marked position in its arrow object. Therefore, by restriction ***E3b***, implementation-side typechecks must consider `point` to be concrete. These checks will find an incompleteness for the argument tuple (`point`, `point`), forcing the creation of a default method implementation such as

```
equal has method(p1@point, p2@point) { … }
```

to cover this case safely. This has the effect of disallowing abstract multimethods; singly dispatched abstract methods are still safe and can be implemented modularly.

In figure 6, the `erase` generic function, assuming that its first argument position is marked, violates restriction ***E4a*** by having the imported `graphic` object on a marked position in its arrow object. Therefore, the `EraseMod` module must extend `GraphicMod`. Then, by restriction ***E4b***, implementation-side checks on `erase` must assume `graphic` is concrete, requiring a default method declaration such as

```
erase has method(g@graphic, d) { … }
```

This method safely handles the unseen `myGraphic` object.

The reflexive, transitive closure of the declared extension relation partitions the program into a set of module regions, the modules in each region connected to one another by extension and having a unique most-extending module. As opposed to the global typechecking of the naive algorithm and the local typechecking of importers, extenders represent a kind of *regional* typechecking. Implementation-side typechecking is still performed in a modular fashion, but each module re-checks its extended modules. The modular checks in the most-extending module of a region ensure the safety of that region, and each region's checking is separate from the others'. A simple link-time check ensures that each module has a most-extending module. By sacrificing complete modularity, System E provides all of our expressive programming idioms, including arbitrary multimethods and multiple implementation inheritance. Binary method idioms, abstract singly dispatched methods, and multiple interface inheritance are still completely modular.

4.4 System ME: Combining Modularity and Expressiveness

Systems M and E are the endpoints in a range of possible modular type systems. System M maximizes the language's modularity of typechecking. The cost for this modularity is a loss of certain expressive programming idioms across module boundaries, including arbitrary multimethods and unanticipated multiple implementation inheritance. System E maximizes the language's expressiveness, at the cost of some regional typechecking, a simple link-time check for unique most-extending modules, and more restrictions on what can be expressed in importers.

By selecting different subsets of restrictions from each of Systems M and E, it is possible to design safe modular type systems with intermediate abilities between these two extremes. In this subsection, we describe one interesting point in this range: *System ME*. In this system, each generic function uses System M's restrictions by default, but if a generic function is expected to need arbitrary methods added to it, it may be given an arrow object with **#** markers on argument positions, in which case System E's rules apply to that generic function. More precisely, System ME includes restrictions ***M1***-***M4*** and ***E1***-***E4***, with a few modifications. The generic functions referred to in restrictions ***M1***, ***M3***, and ***M4*** are only those generic functions that have no marked argument positions, while the generic functions referred to in restrictions ***E1***, ***E3***, and ***E4*** are only those generic functions that have at least one marked argument position. The combination of restrictions ***M2*** and ***E2*** allows multiple inheritance only if it is anticipated and if the two unrelated parent objects are non-imported.

This system gives implementors control over the tradeoff between modularity and flexibility at the granularity of an individual generic function. For example, with System E's rules for the `equal` generic function, we can use extension modules to resolve the ambiguity in figure 3, retaining the use of arbitrary multimethods. At the same time, by using System M's rules for the `erase` generic function in figure 6, we can keep the open object idiom in importers, with completely modular typechecking.

4.5 Summary

Table 1 summarizes the expressiveness of the various type systems we have described and compares them with the expressiveness of singly dispatched languages. System G expresses all of our benchmark programming idioms, at the cost of a global typechecking algorithm. System M has a substantial amount of expressive power while safely using solely local typechecking. The expressive power of the locally checked importers of System E is more limited, but its extenders allow all idioms to be expressed, including arbitrary multimethods and multiple implementation inheritance. System ME provides a nice balance between Systems M and E, maintaining local checking of all modular programming idioms in System M while allowing arbitrary multimethods to be programmed in extenders when needed. The main disadvantage of this system is its complexity, as the declarer of a generic function must decide *a priori* which kinds of extensibility are needed and thus which kinds of restrictions to impose. Finally, all of the type systems compare favorably to conventional singly dispatched languages, which provide fewer than half of these expressive programming idioms. However, such languages typecheck arbitrary multiple implementation inheritance modularly, which none of Dubious's type systems can do because of their support for open object idioms.

In summary, we have presented several alternatives for modular typechecking of multimethods, ranging from a completely modular approach that sacrifices certain programming idioms, to a completely expressive approach that requires some regional typechecking and a simple link-time check. Several issues must be better understood before a clear winner can emerge from among these type systems. Such issues include

	G	M	E		ME		singly dispatched languages
			importers	extenders	importers	extenders	
single dispatching	X	X	X	X	X	X	X
abstract objects	X	X	X	X	X	X	X
binary multimethods	X	X	X	X	X	X	
encapsulated multimethods	X	X		X	X	X	
arbitrary multimethods	X			X		X	
open objects	X	X		X	X	X	
multiple implementation inheritance	X			X			X
multiple interface inheritance	X	X	X	X	X	X	X
typechecking scope	global	local	local	regional	local	regional	local

Table 1: Overview of the expressiveness of the various type systems discussed

understanding which programming idioms are critical to be able to express in the language and which can be sacrificed, which idioms are critical to express purely modularly and which can require regional checking, and which kinds of restrictions are simpler or more intuitive than others. A practical evaluation of these systems is necessary to better understand these issues. We plan to undertake such an evaluation in the context of Diesel, a full programming language succeeding Cecil that will be based on Dubious's foundation for modular type systems for symmetric multimethods.

5 Extensions

This section sketches several extensions to Dubious, moving it closer to a full programming language. First we show how the modular typechecking rules can be exploited to safely allow arbitrarily nested declarations, which supports dynamic object creation and first-class nested functions. Then we describe how to add mutable state, how to add encapsulation, and how to generalize Dubious to the predicate dispatching model.

5.1 Nested Declarations

Dubious enforces a flat structure: modules are declared only at top-level (no modules may be nested in other modules or in methods), and object and method declarations exist only immediately within a module (not at top-level or within a method body). We can relax this restriction, allowing arbitrary nesting of modules and other declarations. For

```
module PointMod {
  import GraphicMod
  object point
  …
  object newPoint isa (int, int)→point
  newPoint has method(newx, newy) {
    object newp isa point
    x has method(p@newp) { newx }
    y has method(p@newp) { newy }
    newp }
  object curried_equal isa (point)→((point)→bool)
  curried_equal has method(p1@point) {
    object eq_p1 isa (point)→bool
    eq_p1 has method(p2) {
      and(=(x(p1),x(p2)), =(y(p1),y(p2))) }
    eq_p1 }
}
import PointMod
curried_equal(newPoint(one,two))(newPoint(two,one))
```

Figure 8: Nested declarations in an extension of Dubious

example, Dubious's program, module, and declaration forms could be reorganized as follows:

P ::= $B\ E$

B ::= $D_1\ \ldots\ D_n$

D ::= [**abstract** | **interface**] **object** I **isa** $O_1,\ldots,O_n$
| I **has method**$(F_1,\ldots,F_n)$ { $B\ E$ }
| **module** I **extends** $I_1,\ldots,I_m$ { B }
| **import** I

In this reorganization, modules are just regular kinds of declarations, method bodies may begin with an arbitrary block of declarations, and programs simplify to a block of declarations followed by an expression. The **import** clause from the **module** declaration is now a separate declaration, allowing any scope to import a module. Nestable modules are largely a namespace-management convenience, but declarations nested in method bodies provide significant additional expressive power, as they are executed each time the enclosing method is invoked at run-time. In particular, this provides dynamic object creation and lexically nested functions, among other programming idioms.

Figure 8 shows a simple example of these two uses of nested declarations. The `newPoint` generic function is a constructor for point "instances" that creates and returns a fresh child of `point` each time it is invoked, initializing the new child's `x` and `y` coordinates appropriately. The `curried_equal` generic function is a curried version of the `equal` function on points, illustrating the creation of lexically nested functions.

Nested **has method** declarations within a method body provide a limited form of mutable state. The `newPoint` generic function illustrates the use of such nested methods for field initialization. In addition, mutable variables can be derived from zero-argument functions. A variable is simply a generic function with no arguments, with a single method whose body returns the variable's value. A variable assignment is simply a method update of this generic function to have a new method body. To make this work, we modify the semantics so that a new method declaration replaces any existing method

with the same tuple of specializers on the same generic function. The following example shows a mutable variable representing the screen size of a display:

```
object screenSize isa ()→point -- declare variable
screenSize has method() { newPoint(1024, 768) } -- set initial value
object update isa (point)→void
update has method(p) { screenSize has method() { p } -- update value }
...
screenSize() -- read current value
```

To typecheck nested declaration blocks (instances of the *B* nonterminal in the grammar above), we make use of the modular typechecking restrictions described in section 4. The fundamental idea is to consider a nested declaration block to be an *importer* of its enclosing scope. If the nested declaration block obeys the typechecking restrictions on importers, then we know that the block will not conflict with unseen importers or other nested declaration blocks of the enclosing scope. To ensure that we can statically check that the nested declaration block is a legal importer, we require that the inheritance parents in the **object** declaration as well as the generic function and specializer objects in the **has method** declaration be statically known objects, rather than potentially computed expressions such as formal arguments. (We also restrict modules nested in a method to extend only modules defined in that same method. Otherwise we could not verify the single most-extending module restriction at link-time.)

The nested declaration block needs to be typechecked in the context of the entire enclosing scope, however, not just the part of the enclosing scope that is visible when the nested declaration block is encountered. Otherwise, the nested declaration block may make unsafe assumptions about its enclosing scope. To handle this issue, we use a two-pass approach to typechecking a declaration block. On the first pass, we typecheck all declarations in textual order, but ignore the bodies of module and method declarations in the block. On the second pass, we typecheck the bodies of these skipped modules and methods, but now in the context of the full environment computed for the enclosing scope. No additional work is performed over a one-pass scheme; merely the order in which the individual typechecking steps are performed changes.

5.2 Mutable State

Nested declarations allow a limited form of mutable state for statically known objects. We incorporate arbitrary mutable state by introducing an alternative form of method update. In particular, we augment the syntax with an additional method declaration:

D ::= ... | *I* **has method**(*I*$_1$ **@=** *E*$_1$,...,*I*$_n$ **@=** *E*$_n$) { *B E* }

Unlike the existing **has method** declaration, in this variant the specializer objects can be computed expressions rather than statically known objects (the generic function identifier *I* must still refer to a statically known object). In conjunction, we require all formals of such methods to use the **@=** specializer symbol. Dynamically, a formal of the form *I***@=***E*, where *E* evaluates to *o* when the method declaration is evaluated, applies only to *o* itself rather than to all descendants of *o*. The following code creates a "set" method illustrating the use of mutable state.

```
object setx isa (point, int)→point
setx has method(myP, newx) {
  x has method(p@=myP) { newx }
  myP }
```

By requiring that all argument positions use **@=**, we ensure that only a single argument tuple applies to the method. Consequently, there can be no possibility of ambiguity between this method and any other method declaration, so implementation-side typechecking can ignore **@=** methods.*

Regular client-side typechecking of the **has method** declaration will verify that the specializer expressions descend from the corresponding argument objects of the generic function's arrow object, and that the body of the method returns an object that descends from the result object of the generic function's arrow type. By requiring that the generic function to which the method is added be a statically known object rather than a computed expression, and by consistently using the generic function's most-specific arrow object when checking its associated **has method** declarations, we prevent soundness problems caused by subsumption in the face of method update.

5.3 Encapsulation

A simple approach to adding privacy to Dubious is to allow an optional **private** keyword to precede an **object** declaration in a module. Making an object private has the effect of disallowing importers of the module from seeing the object. However, it is unsafe in general for a module to be typechecked given only the public interfaces of its importees. The following is a simple example of the problems that can occur:

```
module BadMod {
  abstract object abs
  private object badFun isa (abs)→point
  object fun isa (abs)→point
  fun has method(a@abs) { badFun(a) }
}

module ImpMod imports BadMod {
  object conc isa abs
}
```

Because the `ImpMod` creates a new concrete implementation of the abstract `abs` object, it must check that the new implementation is correctly implemented (under both Systems M and E). If the typechecking of `ImpMod` does not see the private `badFun` generic function, then the `conc` object appears correctly implemented, because the `fun` generic function has an appropriate implementation. However, if this function is ever invoked on `conc`, the resulting invocation of `badFun` will cause a message-not-understood error to occur.

Our modular typechecking restrictions can be applied to overcome this problem. In addition to our normal checks on a module, we require that the private part pass the necessary checks as if it were a separate module that imported the public part of the module. (This means, for example, that a public object cannot inherit from a private object.) For the purposes of dividing up the module, a method is treated as private if its generic function or any of its specializer objects is private. If the private part obeys the rules on importers, then we know that importers of the module can be safely typechecked without seeing this private part. In our example above, the `badFun` object would need to pass implementation-side typechecking as if it were in its own module that imported a module containing only the public part of the module. Therefore, the object is considered to import the `abs` object, so `badFun` is subject to restrictions on open object idioms. In order to satisfy these restrictions, `badFun` must provide a default method implementation, thereby removing the potential type error that eludes typechecks from the `ImpMod` module.

---

* There is no conflict in having one **@** and one **@=** method, each specialized to the same objects; both methods apply to that tuple of objects, but the **@=** method is treated as the more specific method. The **@** method will still apply to any descending tuples.

5.4 Predicate Dispatching

A recent paper described predicate dispatching [Ernst *et al.* 98], a generalized dispatching model that subsumes single dispatching, multiple dispatching, pattern matching, and predicate classes. The basic idea is to specify a method's applicability by an arbitrary predicate over the method's formals, formed from conjunctions, disjunctions, and negations of descends-from tests (`@` specializers), equal-to tests (`@=` specializers), and arbitrary boolean-valued expressions. Method overriding is deduced from predicate implication. Under this model, a traditional multimethod is simply a method whose predicate is a conjunction of descends-from tests of the method's formals. Global static typechecking rules were presented for this model.

Modular typechecking of the predicate dispatching model follows from the observation that our modular typechecking rules for multimethods are already sufficient conditions for the safety of predicate methods. For example, System M requires that a method added to an imported generic function be an encapsulated-style method. In predicate dispatching, this corresponds to a method whose predicate is a conjunction where one conjunct is a test that the first formal descends from a local object. All other conjuncts of the predicate are unrestricted; such conjuncts can only further constrain the applicable argument tuples to the method.

6 Related Work

Chambers and Leavens made the first effort toward modular typechecking of symmetric multimethods [Chambers & Leavens 95], in the Cecil language [Chambers 92, Chambers 95]. They defined client-side and implementation-side typechecking and sketched informal ideas for modular typechecking, including the notions of extension modules and unique most-extending modules. In this sketch, objects are not allowed to conform to an imported type (objects and types are orthogonal in their model). When the object and type hierarchies parallel each other, as is the common case, this restriction disallows modules even from creating a singly inheriting child of an imported object. This makes importers unable to express most standard object-oriented programming idioms, forcing much of the program into extension modules and thereby largely giving up modular typechecking. They did not formalize the static or dynamic semantics of their modular language nor prove any soundness results for modular typechecking.

BeCecil [Chambers & Leavens 97] is a core language for multimethods, intended as a formalization of the work described above. It includes the same core features as Dubious, as well as types and subtyping separate from objects and inheritance, and a block structure that allows arbitrarily nested declarations. However, BeCecil does not have a module system. BeCecil's dispatching semantics is more complicated than Dubious's, with **`inherits`** (BeCecil's version of **`isa`**) and **`has method`** declarations associated with particular scopes and only visible to certain call sites. As a result, separate typechecking was not achieved. Dubious is in some ways a reaction to BeCecil's problems: Dubious treats **`isa`** and **`has method`** declarations as having global extent, simplifying the semantics enough for us to develop modular typechecking rules and prove their soundness.

The λ&-calculus and variants [Castagna *et al.* 92, Castagna 97] are a family of calculi based on overloading of symmetric multimethods. Dispatching is performed on types which, along with the subtyping relation, are predefined rather than programmer constructed. Some of the calculi have second-order type systems, which Dubious lacks. The language λ_object augments the calculi with the ability to define new types and subtyping relations. Neither the λ&-calculi nor λ_object address the issue of separate typechecking. Moreover, the functional nature of the & operator to add a method to a

generic function prevents spreading the definition of a generic function across unrelated modules, as would be needed to model independently developed subclasses.

$ML_{\leq}$ [Bourdoncle & Merz 97] is an ML-like language augmented with a form of classes and symmetric multimethods. The type system is more sophisticated than ours, including types separate from implementations and polymorphic multimethods. The authors sketch an extension to the language that adds modules for encapsulation. However, there is no separate typechecking, as these modules simply desugar into a global "letrec."

Kea [Mugridge *et al.* 91] is a statically typed, class-based language with symmetric multimethods. Kea has a notion of separate compilation, but this requires run-time checking of generic functions for type safety.

Tuple [Leavens & Millstein 98] is a language that provides dispatching on tuples in order to add symmetric multiple dispatch to existing singly dispatched languages. However, the modularity issue for multimethods is not addressed, as "tuple classes" must be typechecked globally. The syntax for tuple classes, which clearly separates the specialized and unspecialized argument positions of a generic function, has the same effect as the **#** markers in our System E.

Encapsulated multimethods [Castagna 95, Bruce *et al.* 95] are an attempt to solve the modularity problem for multimethods by embedding multimethods in the traditional object-oriented class model. An encapsulated multimethod first dispatches on the receiver argument, then dispatches on the remaining arguments. All the multimethods with a given receiver are encapsulated within the receiver class and are not extensible or inheritable outside this class. In the face of multiple inheritance, all the encapsulated multimethods within a class must be totally ordered. Given these restrictions, each class can be typechecked separately. The resulting kinds of allowable multimethods are similar to those allowed in our System M, although we are able to keep the symmetric multimethod dispatching semantics and to maintain ordinary inheritance of methods. In addition, encapsulated multimethods do not address open objects and abstract methods. Parasitic multimethods [Boyland & Castagna 97] are a variant of encapsulated multimethods adapted to Java. Parasitic multimethods are additionally complicated by the use of the textual order of methods to resolve ambiguities, the inheritance of parasitic methods in the presence of this textual ordering, and the need to retain backward compatibility with Java's blend of dynamic single dispatching and static overloading on arguments.

Common Lisp [Steele 90, Paepcke 93] and Dylan [Shalit 97, Feinberg *et al.* 97] are both multimethod-based languages with generic functions and module systems. To avoid run-time ambiguities, Common Lisp totally orders the arguments of a generic function; Dylan uses the symmetric model. Both Common Lisp and Dylan totally order the inheritance hierarchy, eliminating the potential for multiple-inheritance ambiguities. The module systems provide name-space management and encapsulation, allowing the creation of generic functions private to a module. The languages are dynamically typed, so they do not consider the issue of (separate) typechecking.

Polyglot [Agrawal *et al.* 91] is a database programming language akin to Common Lisp with a first-order static type system. There are no abstract methods and the type of a generic function is determined by the types of its methods, so there is no possibility of message-not-understood errors. Further, the dispatching uses Common Lisp-style total ordering of multimethod arguments and inheritance, avoiding ambiguities. Therefore, only the monotonicity of the result types [Castagna *et al.* 92, Reynolds 80] of multimethods needs to be checked to ensure type safety.

7 Conclusions and Future Work

Dubious is a statically typed core language that supports symmetric multimethods and separate typechecking. The core language includes explicit declaration of (possibly abstract) objects, (possibly multiple) inheritance, and first-class generic functions. We have defined several modular type systems for Dubious, with properties ranging from complete modularity at the cost of giving up certain programming idioms, to complete expressiveness at the cost of regional typechecking and a simple link-time check on regions, and we have proved the systems sound. Dubious represents the first formalization and the first proof of soundness of separate typechecking for symmetric multimethods.

In the future, we plan to formalize and prove sound the extensions to the language sketched in section 5. Other interesting extensions include supporting polymorphic types in the presence of separate typechecking, and supporting module types and first-class modules to program mixin classes [Bracha & Cook 90, Flatt *et al.* 98] and role-based programming [Reenskaug *et al.* 92, VanHilst & Notkin 96, Smaragdakis & Batory 98]. Finally, we are using the ideas in Dubious as a foundation for the design of Diesel, a practical programming language succeeding Cecil.

Acknowledgments

Thanks to Gary Leavens for his collaboration on BeCecil, the predecessor to Dubious, and for many discussions about typechecking for multimethods. Thanks to Vassily Litvinov for discussion on the formal semantics of Dubious. Thanks to Greg Badros, Michael Ernst, David Grove, Craig Kaplan, Gary Leavens, and Vassily Litvinov for helpful comments on an earlier draft of this paper.

This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software.

References

[Abadi & Cardelli 95] Martín Abadi and Luca Cardelli. An Imperative Object Calculus. *Theory and Practice of Object Systems*, **1**(3):151-166, 1995.

[Abadi & Cardelli 96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Agrawal *et al.* 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. *OOPSLA'91 Conference Proceedings*, Phoenix, AZ, October, 1991, volume 26, number 11 of *ACM SIGPLAN Notices*, pp. 113-128. ACM, New York, November, 1991.

[Arnold & Gosling 98] Ken Arnold and James Gosling. *The Java Programming Language*. Second Edition, Addison-Wesley, Reading, Mass., 1998.

[Baumgartner *et al.* 96] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the Interaction of Object-Oriented Design Patterns and Programming Languages. Technical Report CSD-TR-96-020, Department of Computer Science, Purdue University, February 1996.

[Bourdoncle & Merz 97] François Bourdoncle and Stephan Merz. Type Checking Higher-Order Polymorphic Multi-Methods. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 302-315. ACM, New York, January 1997.

[Boyland & Castagna 97] John Boyland and Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 32, number 10 of *ACM SIGPLAN Notices*, pp. 66-76. ACM, New York, November 1997.

[Bracha & Cook 90] Gilad Bracha and William Cook. Mixin-Based Inheritance. *Proceedings of the Joint ACM Conference on Object-Oriented Programming Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, Ottawa, Canada, October 1990.

[Bruce *et al.* 95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, **1**(3):221-242, 1995.

[Canning *et al.* 89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, John C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273-280, September 1989.

[Cardelli 84] Luca Cardelli. A Semantics of Multiple Inheritance. *Semantics of Data Types Symposium*, LNCS 173, pp. 51-66, Springer-Verlag, 1984.

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

[Castagna 95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431-447, 1995.

[Castagna 97] Giuseppe Castagna. *Object-Oriented Programming A Unified Foundation*, Birkhäuser, Boston, 1997.

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

[Chambers 95] Craig Chambers. The Cecil Language: Specification and Rationale: Version 2.0. Department of Computer Science and Engineering, University of Washington, December, 1995. http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html

[Chambers & Leavens 95] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. *ACM Transactions on Programming Languages and Systems*, **17**(6):805-843. November, 1995.

[Chambers & Leavens 97] Craig Chambers and Gary T. Leavens. BeCecil, A Core Object-Oriented Language with Block Structure and Multimethods: Semantics and Typing. *The Fourth International Workshop on the Foundations of Object-oriented Languages*, Paris, France, January 1997.

[Chambers 98] Craig Chambers. Towards Diesel, a Next-Generation OO Language after Cecil. Invited talk, *The Fifth Workshop on Foundations of Object-oriented Languages*, San Diego, California, January 1998.

[Cook 90] William Cook. Object-Oriented Programming versus Abstract Data Types. *Foundations of Object-Oriented Languages*, REX School/Workshop Proceedings, Noordwijkerhout, the Netherlands, May/June, 1990, volume 489 of *Lecture Notes in Computer Science*, pp. 151-178. Springer-Verlag, New York, 1991.

[Ernst *et al.* 98] Michael D. Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. *Twelfth European Conference on Object-Oriented Programming*, Brussels, Belgium, pp. 186-211, July, 1998.

[Feinberg *et al.* 97] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.

[Findler & Flatt 98] Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. *International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[Flatt *et al.* 98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, pp. 171-183. ACM, New York, January 1998.

[Gamma *et al.* 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[Gosling *et al.* 96] James Gosling, Bill Joy, Guy Steele, Guy L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Mass., 1996.

[Kiczales *et al.* 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. In proceedings of the *Eleventh European Conference on Object-Oriented Programming*, Finland. Springer-Verlag LNCS 1241. June 1997.

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

[Leavens & Millstein 98] Gary T. Leavens and Todd D. Millstein. Multiple Dispatch as Dispatch on Tuples. *Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, October 1998.

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

[Millstein & Chambers 99] Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. Technical Report UW-CSE-99-03-02, Department of Computer Science and Engineering, University of Washington, March 1999.

[Mugridge *et al.* 91] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-Methods in a Statically-Typed Programming Language. *ECOOP '91 Conference Proceedings*, Geneva, Switzerland, July, 1991, volume 512 of Lecture Notes in Computer Science; Springer-Verlag, New York, 1991.

[Odersky & Wadler 97] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 146-159. ACM, New York, January 1997.

[Paepcke 93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Reenskaug *et al.* 92] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6): October 1992, pp. 27-41.

[Reynolds 80] John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. *Semantics-Directed Compiler Generation*, Aarhus, Denmark, pp. 211-258. Volume 94 of *Lecture Notes in Computer Science*, Springer-Verlag, NY, 1980.

[Shalit 97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

[Smaragdakis & Batory 98] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. *Twelfth European Conference on Object-Oriented Programming*, Brussels, Belgium, pp. 550-570, July 1998.

[Steele 90] Guy L. Steele Jr. *Common Lisp: The Language* (*second edition*). Digital Press, Bedford, MA, 1990.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *OOPSLA '87 Conference Proceedings*, Orlando, Florida, volume 22, number 12, of *ACM SIGPLAN Notices*, pp. 227-241. ACM, New York, December, 1987.

[VanHilst & Notkin 96] M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, pp. 22-37.