

# Modular Statically Typed Multimethods

Todd Millstein and Craig Chambers

Department of Computer Science and Engineering, University of Washington, Seattle, Washington 98195  
E-mail: todd@cs.washington.edu, chambers@cs.washington.edu

Published online April 22, 2002

Multimethods offer several well-known advantages over the single dispatching of conventional object-oriented languages, including a simple solution to the binary method problem, a natural implementation of the strategy design pattern, and a form of open objects that enables easy addition of new operations to existing classes. 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. © 2002 Elsevier Science (USA)

## 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.*, 1995) and a simple implementation of the strategy design pattern (Gamma *et al.*, 1995). Finally, multimethods naturally allow clients to add new operations that dynamically dispatch on existing classes, supporting a form of what we call open objects (Chambers 1998) that enables easy programming of the visitor design pattern (Gamma *et al.*, 1995; Baumgartner *et al.*, 1996) and is a key element of aspect-oriented programming (Kiczales *et al.*, 1997). In this way, multimethod languages support the addition of both new subclasses and new operations to existing classes, relieving the tension that has been observed by others (Cook, 1990; Odersky and Wadler, 1997, Findler and Flatt, 1998) between these forms of extension.

A key challenge for multimethods is separate static typechecking: it is possible for two modules containing multimethods to typecheck successfully in isolation but generate type errors when linked together. Previous work on statically typed multimethods has dealt with this problem in two ways. Some work has simply forced programs to be typechecked as a whole (Mugridge *et al.*, 1991; Castagna *et al.*, 1992; Chambers, 1992; Bourdoncle and Merz, 1997; Castagna, 1997; Chambers and Leavens, 1997; Leavens and Millstein, 1998), thereby losing the ability to safely, independently develop code that is later combined into a single program. Other work sacrifices symmetric treatment of multimethod arguments to ensure the safety of modular typechecking (Agrawal *et al.*, 1991; Bruce *et al.*, 1995; Boyland and Castagna, 1997), thereby giving up the natural multimethod dispatching semantics, which reports all potential ambiguities rather than silently resolving them.

We have designed Dubious, a simple core language supporting both *symmetric multimethods*, multimethods whose dispatching semantics treats all argument positions uniformly, and separately typechecked modules. We have identified two sets of restrictions on modules that achieve modular type safety, representing different tradeoffs between expressiveness and modularity. We also identify an intermediate point between these two that provides programmers with fine-grained control over this tradeoff. We have proved all three of these modular type systems sound.

The rest of this paper is organized as follows. 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. The appendices provide formal dynamic and static semantics for Dubious, as well as a sketch of the soundness proofs for the various type systems presented.

## 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, or encapsulation mechanisms.

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

Dubious’s syntax appears in Fig. 1;  $\varepsilon$  denotes the empty string and brackets denote optional parts of the syntax. The following subsections present Dubious’s semantics and typechecking rules informally. The formal dynamic and static semantics of Dubious are detailed in Appendices A and B, respectively.

### 2.1. Informal Semantics

A Dubious program consists of a collection of modules, followed by an expression that is 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.)

Four Dubious modules are shown in Fig. 2; they will be used as a running example throughout this section. The `GraphicMod` module creates a template for objects that can be drawn. The `PointMod` module then implements this template, creating a point object with an  $x$ -coordinate. The `ColorPointMod` module creates a subclass of `point` that has a color. The `OriginMod` module creates a particular instance of a `point`, with a zero-valued  $x$ -coordinate.

One module **imports** another, in order to access its objects, by making their names visible. The import relation is transitive; for example, in Fig. 2 the `ColorPointMod` module may refer to objects in the `StdLibMod` module. For simplicity, we address name clashes by making an imported object shadow any previously imported object of the same name.

The body of a module is a sequence of declarations. Dubious has only two kinds of declarations. 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

```

P ::= M1 ... Mn import I in E end
M ::= module I imports I1, ..., Im { D1 ... Dn }
D ::= Q object I isa O1, ..., On
    | I has method(F1, ..., Fn) { E }
Q ::= abstract | interface |  $\varepsilon$ 
O ::= I | (O1, ..., On)  $\rightarrow$  O
F ::= I1 [ @ I2 ]
E ::= I | E0(E1, ..., En)
I ::= identifier

```

FIG. 1. Syntax of Dubious.

```

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 }
  draw has method(p@point, d) { ... }
  object equal isa (point, point)→bool
  equal has method(p1@point, p2@point) {
    eq_int(x(p1), x(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(eq_int(x(cp1), x(cp2)),
        eq_color(col(cp1), col(cp2))) }
}
module OriginMod imports PointMod {
  object origin isa point
  x has method(o@origin) { zero }
  equal has method(o1@origin, o2@origin) { true }
}

```

FIG. 2. A simple Dubious program fragment.

relation. As in other classless languages (LaLonde *et al.*, 1986; Lieberman, 1986; Ungar and Smith, 1987; Chambers, 1992; Abadi and Cardelli, 1996), objects play the roles of both classes and instances, and **isa** accordingly plays the roles of both inheritance and instantiation. For example, in the modules of Fig. 2, the `colorPoint` object represents a subclass of `point` that has a color, while the `origin` object represents a particular point instance.

Dubious has explicit *generic functions*, collections of methods of the same name. To make generic functions first class, they are modeled as objects that inherit from an *arrow object* defining the legal arguments and return values of the function. For example, the `equal` object declared in the `PointMod` module in Fig. 2 inherits from the  $(\text{point}, \text{point}) \rightarrow \text{bool}$  arrow object, specifying that `equal` is 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, 1988) 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 descends. We refer to this most-specific arrow object as the arrow object of the generic function.

A method with name  $I$  is implemented by adding the method to the generic function named  $I$  via the **has method** declaration. In the `PointMod` module example in Fig. 2, the method added to the `draw` generic function 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 formal 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. This method is (explicitly) specialized only on its first argument; in traditional class-based object-oriented languages, this method would be modeled as the “draw” method inside the `point` class. On the other hand, the method added to the `equal` object in the `PointMod` module 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 1995, 1996. For example, the `ColorPointMod` module adds a second method to the `draw` generic function 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 (nonterminal  $E$  in Fig. 1). For example, the abstract `graphic` object in Fig. 2 is a template for objects that can be drawn, which is then implemented by the `point` object via inheritance. 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 and Gosling, 1998; Gosling *et al.*, 1996). 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 *nonconcrete* object is either abstract or an interface, and a *noninterface* object is either concrete or abstract.)

To evaluate a generic function application (message send)  $E(E_1, \dots, E_n)$ , we employ the symmetric multimethod dispatching semantics. We evaluate  $E$  to some generic function  $f$ , evaluate each expression  $E_i$  to some object  $o_i$ , extract the methods added to the generic function  $f$ , and then select and invoke the *most specific applicable method* for  $(o_1, \dots, o_n)$ . A method in  $f$  is *applicable* to  $(o_1, \dots, o_n)$  if the method has  $n$  arguments and if  $(o_1, \dots, o_n)$  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 Fig. 2. First, the `equal` and `colorPoint` expressions are evaluated, yielding the objects with those names.<sup>1</sup> Then the methods that were added to the `equal` generic function 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.

## 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 and Leavens, 1995). *Client-side* 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  $E(E_1, \dots, E_n)$  in the program, we check that  $E$  descends from an arrow object  $(O_1, \dots, O_n) \rightarrow O$  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  $(I_1@I'_1, \dots, I_n@I'_n) \{E\}$  in the program, we check that  $I$  descends from an arrow object  $(O_1, \dots, O_n) \rightarrow O$ , 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  $f$  fully and unambiguously implements its arrow object  $(O_1, \dots, O_n) \rightarrow O$ . We say that  $(o_1, \dots, o_n)$  is a *legal argument tuple* to  $f$  if each  $o_i$  is concrete and descends from  $O_i$ . Implementation-side checks are that each legal argument tuple to  $f$  has a most specific applicable method added to  $f$ . For example, consider implementation-side checks on the `equal` generic function, which is declared to inherit from `(point, point) → bool`. There are nine legal argument tuples to `equal`: all possible pairs of the objects `point`, `colorPoint`, and `origin`. The `(@colorPoint, @colorPoint)` method is most specific for two `colorPoints`, the `(@origin, @origin)` method is most specific for two `origins`, and the `(@point, @point)` method is most specific for all other tuples.

<sup>1</sup>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.

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. On the other hand, for each generic function  $f$ , implementation-side checks require knowledge of every method added to  $f$  and every possible legal argument tuple to  $f$ . Therefore, implementation-side typechecking on all generic functions is deferred until link-time, when all modules are present. This global typechecking approach, which we refer to as *System G*, is used for typechecking all previous languages with symmetric multimethods, including Kea (Mugridge *et al.*, 1991), Cecil (Chambers, 1992; Chambers, 1995),  $ML_{\leq}$  (Bourdoncle and Merz, 1997), and Tuple (Leavens and Millstein, 1998).

### 3. EXAMPLE PROGRAMMING IDIOMS

There are several flexible programming idioms expressible in Dubious that we would like to be able to statically typecheck. These idioms will serve as benchmarks for evaluating the various typechecking algorithms presented in this paper. The global typechecking of System G supports all of 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 Fig. 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 multimethods* result, which enable a simple solution to the binary method problem (Bruce *et al.*, 1995). For example, the three `equal` methods in Fig. 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 subtypes of `point`, able to be safely substituted for `point` anywhere that it is expected, so subtype polymorphism over the `point` hierarchy is still available.

A 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 a *first-local multimethod*. It is similar to the style of multimethods allowed by encapsulated multimethods (Castagna, 1995; Bruce *et al.*, 1995) and parasitic multimethods (Boyland and Castagna, 1997). Using first-local multimethods, new objects can interact with imported objects in interesting ways, without modifying the code for the imported module. For example, suppose that the `StdLibMod` module, which is imported by the `GraphicMod` module, contains a whole hierarchy of objects representing different kinds of displays. The `ColorPointMod` module can use first-local multimethods to program how colored points are drawn on these various displays, without modifying the display hierarchy:

```
draw has method(cp@colorPoint, d@display) {...} --default draw method
draw has method(cp@colorPoint, d@colorDisplay) {...} --draw in color
...
```

If we remove all restrictions on method specializers, allowing any or all to be imported objects, we obtain what we call *arbitrary multimethods*. These multimethods allow interesting interactions among

imported objects to be added anywhere in the program. For example, suppose the implementor of the `ColorPointMod` module did not foresee the need for the above `draw` methods. A client can add the new methods to the program without modifying either the `ColorPointMod` or the `StdLibMod` modules by creating a new module for the methods:

```

module DrawMod imports ColorPointMod {
  draw has method (cp@colorPoint, d@display) {...} --default draw method
  draw has method (cp@colorPoint, d@colorDisplay) {...} --draw in color
  ...
}

```

A final desirable idiom is *open objects* (Chambers, 1998), where an object declared in an imported module is extended by adding new operations (generic functions) that dispatch on the object, without modifying the imported module. For example, the following module introduces a `lineseg` object representing a line segment. The newly created `distance` generic function extends the `point` object and descendants with a new operation, without modifying existing code.

```

module LineSegMod imports OriginMod {
  object 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 very useful for singly dispatched methods as well, as in the case of the `distance` generic function above. Open objects support simpler programming of the visitor design pattern (Gamma *et al.*, 1995). Client-specific visitors are programmed directly, without needing to build a special visitor infrastructure for each object hierarchy. Importantly, the open object idiom retains the ability to add new subclasses without modifying existing code, while the visitor pattern does not. Therefore, open objects allow both kinds of object extension, new descendants and new operations, without modifying existing code. In the `lineseg` example, the `distance` generic function is a kind of client-specific visitor of the `point` hierarchy. This new operation is added without modifying or breaking the `point` hierarchy, and new descendants of `point` can be added in other modules without modifying or breaking the `distance` generic function (as long as the implementations of `distance` inherited by new descendants are still appropriate).

#### 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 to typecheck each module in isolation, and we would like these modular typechecks to have the following properties:

- The generic functions created in the module are implementation-side typechecked given only the *interfaces* of imported modules: the names of objects in those modules and the associated inheritance hierarchy.
- 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 that descends from a nonconcrete object and that may be sent as an argument to the generic function. Further, this checking does not reexamine argument tuples already checked by the importee.

As an example, consider a modular implementation-side typechecking scheme for the `equal` generic function in Fig. 2. The `equal` generic function is created in the `PointMod` module, so the typechecks on the `PointMod` module must include implementation-side typechecking of `equal`. This checking ensures that the only visible legal argument tuple, `(point, point)`, has a most specific applicable visible method implementation. Since the `ColorPointMod` module adds a new equality method, the `equal` generic function is also implementation-side typechecked during the `ColorPointMod` module's checks. However, the `(point, point)` tuple is not rechecked; only visible legal argument tuples containing at least one `colorPoint` are checked. A similar situation occurs when the `OriginMod` module is typechecked.

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 and Leavens, 1995). Section 4.1 describes the situations that can lead to such errors. Sections 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. Section 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 section, 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 Fig. 3, where each of the `ColorPointMod` and `OriginMod` modules from Fig. 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. In particular, both methods in the example are applicable to the message `send`, but neither method is more specific than the other; the `ColorPointMod` module's method is more specific in the first argument position, while the `OriginMod` module's method is more specific in the second argument position.

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, 1990; Paepcke, 1993) and Polyglot (Agrawal *et al.*, 1991), 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

```

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) { ... }
}

```

FIG. 3. Ambiguity problem 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 }
}

```

FIG. 4. Incompleteness problem with multimethods on nonconcrete objects.

used in the languages Cecil (Chambers, 1992, 1995), Dylan (Shalit, 1997, Feinberg *et al.*, 1997), the  $\lambda$ -calculus (Castagna *et al.*, 1992; Castagna, 1997), Kea (Mugridge *et al.*, 1991),  $ML_{\leq}$  (Bourdoncle and Merz, 1997), and Tuple (Leavens and Millstein, 1998).

A second unsafe scenario involves the combination of nonconcrete objects with multimethods. For example, suppose we make the `point` object in Fig. 2 abstract, so that equality for `point` objects need not be 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 each legal type of concrete objects has 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 Fig. 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 these modules do not see each other statically, typechecks on each module will fail to detect 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, 1990; Paepcke, 1993) and Dylan (Shalit, 1997; Feinberg *et al.*, 1997). 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 nonconcrete objects. In Fig. 6, the `EraseMod` module extends the `graphic` interface from Fig. 2 with an operation for erasing a graphical object. The `erase` generic function is completely implemented for all the concrete implementations of `graphic` that are visible to that module. The `MyGraphicMod` module creates a concrete implementation of a `graphic` object. From its point of view, the `myGraphic` object completely implements the `graphic` interface. 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).

```

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
}

```

FIG. 5. Ambiguity problem 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) { ... }
}

```

FIG. 6. Incompleteness problem combining open objects with nonconcrete objects.

#### 4.2. System M: Maximizing Modularity

Because of the four unsafe programming scenarios described above, any completely modular type-checking scheme must restrict the usage of certain Dubious language constructs. In this section, 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 aim with System M is to provide the most flexible type system possible, subject to those strict modularity goals.

We say that a nonarrow object or a method is *local* if it is declared in the current module, and otherwise it is *nonlocal*. An arrow object is local if it has a local object in a positive<sup>2</sup> position, and otherwise it is nonlocal. Two objects are *related* if one object descends 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 descends from a nonlocal, nonconcrete object  $o$  without also descending from some concrete descendent of  $o$ . *Implementation inheritance* is inheritance from a noninterface 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 add a first-local method to the same generic function and multiple implementation inheritance across module boundaries is disallowed, then the two methods will be applicable to disjoint sets of legal argument tuples. Therefore, these restrictions safely remove method ambiguity problems. Potential incompleteness problems are removed by treating visible nonconcrete 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 nonconcrete objects in this way except for local nonconcrete objects that may be sent as the first argument to a local 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 nonlocal generic function must be a first-local method; i.e., the first argument must be specialized to a local object.<sup>3</sup>
- **(M2)** If a local noninterface object  $o$  descends from two unrelated, nonlocal, noninterface objects  $o_1$  and  $o_2$ , then  $o$  must also descend from a noninterface object  $o_3 \neq o$  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 nonconcrete visible descendants of  $o$  to be concrete on that argument position.
- **(M4)** If a local generic function's arrow object has the nonlocal object  $o$  in the first argument position, then implementation-side typechecks of the generic function must consider any nonlocal, nonconcrete visible descendants of  $o$  to be concrete on the first argument position.

<sup>2</sup> 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.*, 1989). 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 generic functions that descend from the arrow. Due to contravariance, the same is not true if local objects appear only in negative positions.

<sup>3</sup> 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.

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 type-checks local generic functions given only the interfaces of its importees. In addition, there are two scenarios in which a module must recheck nonlocal generic functions:

- If the module adds methods to a nonlocal generic function, then this generic function is implementation-side typechecked. However, only legal argument tuples to which a local method is applicable need be checked.
- If the module creates an orphan  $o$ , then all nonlocal 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 nonconcrete descendants of the first argument in a nonlocal generic function’s arrow object, which was left unchecked by rule **M4** above. In this way, we safely allow abstract object idioms.)

We can use the above restrictions to resolve the problems in the examples of the previous section. In Fig. 3, the `equal` method in the `OriginMod` module would cause a static type error. In particular, it is not a first-local method because it has a nonlocal 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 Fig. 3 to be first-local (in conjunction with the multiple inheritance limitations imposed by restriction **M2**) ensures that the methods are applicable to disjoint sets of argument tuples. In particular, the revised method above is no longer applicable to the message `send equal(colorPoint, origin)`.

In Fig. 4, both the `colorPoint` and `origin` objects are orphans. Because `equal` accepts the `colorPoint` and `origin` objects on its first position, it is rechecked when the `ColorPointMod` and `OriginMod` modules are typechecked. By restriction **M3**, these checks must consider the abstract `point` object to be concrete for the second argument position. Therefore, rechecks from the `ColorPointMod` module will find an incompleteness for the argument tuple `(colorPoint, point)`, and similarly for the `OriginMod` module’s rechecks. 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. In this way, we safely allow abstract object idioms, as the `PointMod` module need not implement the `equal` generic function for the abstract `point` object. Instead, appropriate implementations are provided by importers that create concrete descendants of `point`.

In Fig. 5, the `colorOrigin` object fails restriction **M2** because it descends from two unrelated, nonlocal, noninterface objects without also descending from some noninterface 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, so interface objects cannot generate ambiguities), as is arbitrary multiple implementation inheritance within a module.

In addition, multiple implementation inheritance can be programmed safely within System M’s restrictions if it is *anticipated* when at least one of the parents is implemented. For example, if the implementor of the `OriginMod` module anticipated that multiple inheritance from the `colorPoint` and `origin` objects 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 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. For example, if the `PrintMod` module in Fig. 5 imports the revised `OriginMod` module above, restriction **M4** would force the existence of a `print` method to disambiguate printing a `colorPointAndOrigin`, thereby removing the potential message-ambiguous error. 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 Fig. 6, the first argument of the `erase` generic function's arrow object is the nonlocal 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 nonlocal first specializer are disallowed, but all other kinds of multimethods may be programmed. This provides several important multimethod idioms, including binary methods and multimethods in the encapsulated style (Castagna, 1995; Bruce *et al.*, 1995). Abstract object idioms are allowed, as long as the appropriate default method implementations are included to prevent unseen incompletenesses. Finally, arbitrary open objects may be programmed modularly, at the cost of a loss of unanticipated cross-module multiple implementation inheritance. Therefore, System M provides the ability to safely extend existing objects modularly, both by adding new subclasses and by adding new operations.

#### 4.3. System E: Maximizing Expressiveness

Because 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 nonlocal specializer, and unanticipated multiple implementation inheritance. In this section, we present *System E*, whose fundamental requirement is to be able to express all of our benchmark programming 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  $m_1$  *extending* another module  $m_2$  instead of importing it (Chambers and Leavens, 1995) whenever  $m_1$  needed to make use of one of the more aggressive idioms with respect to objects in  $m_2$ . If each module in the program rechecks all the generic functions created in 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:

$$M ::= \text{module imports } I_1, \dots, I_m \text{ extends } I'_1, \dots, I'_r \{D_1 \dots D_n\}$$

In the context of System E, we say that a nonarrow 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 extended if it has only extended objects in positive positions, and otherwise it is imported. A *nonlocal* object is either imported or extended, and similarly for a *nonimported* and *nonextended* 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 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 with respect to imported objects. Second, because extenders can write arbitrary multimethods, we restrict modules to add only all-local methods (not just first-local 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 importers. For example, such methods could not even accept a descendant of `int` as an argument unless that descendant was created in the importing module. To alleviate this problem, we introduce syntax for specifying, as part of a generic function declaration, which of its arguments may be specialized:

$$\begin{aligned} O &::= I \mid (S_1, \dots, S_n) \rightarrow O \\ S &::= HO \\ H &::= \# \mid \varepsilon \end{aligned}$$

A `#` marker denotes a *marked argument position* of an arrow object. The marked argument positions of a generic function are simply the marked argument positions of the generic function's arrow object. Methods may not specialize at unmarked argument positions. This allows methods added to imported generic functions to safely accept nonlocal objects on unmarked positions. Methods added to 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 nonconcrete objects to be concrete for the purposes of implementation-side typechecking a generic function. As with System M, there are situations in which nonconcrete objects may safely remain incompletely implemented. In particular, if the generic function is local or extended and has exactly one marked argument position, then any local nonconcrete objects that may be sent to the marked position need not be considered concrete during implementation-side typechecking. This safely allows abstract object idioms on singly dispatched generic functions in importers. However, to ensure the safety of open object idioms, we must require that nonlocal, noninterface objects that may be sent to the marked argument position be considered concrete during the implementation-side typechecking of an extended generic function. In order that this does not prohibit abstract object idioms in extenders, we require argument tuples including such objects to be checked only for ambiguity, but not for incompleteness, thereby allowing the generic function to remain unimplemented for such argument tuples.

More formally, System E imposes the following five restrictions on modules. The first four are directly analogous to the four restrictions of System M, and the fifth restriction rules out open object idioms in importers:

- (**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 noninterface object  $o$  descends from an imported object and descends from two unrelated, noninterface objects  $o_1$  and  $o_2$ , where at least one of  $o_1$  and  $o_2$  is nonlocal, then  $o$  must also descend from a noninterface object  $o_3 \neq o$  that descends from  $o_1$  and  $o_2$  as well.

- (**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 nonconcrete 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 nonconcrete visible descendants of  $o$  to be concrete.

- (**E4a**) If a local singly dispatched generic function's arrow object has the nonlocal object  $o$  in its marked argument position, then implementation-side typechecks of the generic function must consider any nonlocal, nonconcrete visible descendants of  $o$  to be concrete. (**E4b**) If an extended singly dispatched generic function's arrow object has the nonlocal object  $o$  in its marked argument position, then implementation-side typechecks of the generic function must consider any nonlocal, nonconcrete visible descendants of  $o$  to be concrete. However, legal argument tuples including such descendants are checked only for ambiguity, not for incompleteness.

- (**E5**) If a local generic function's arrow object has the object  $o$  in a marked argument position, then no visible descendant of  $o$  (including  $o$  itself) may be an imported object.

To ensure the safety of module extension, each module reimplementationside typechecks the generic functions created in extended modules. In addition, there are two kinds of rechecks 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 function 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 nonconcrete descendants of the first argument in an imported generic function's arrow object, which was left unchecked by rule *E4* above. In this way, we safely allow abstract object idioms in importers for singly dispatched generic functions.)

Finally, 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, abstract object idioms for singly dispatched operations, and binary methods. For example, each of the modules in Fig. 2 satisfies the System E 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. To ensure safety of the `equal` generic function, the rechecks in the `ColorPointMod` module simply need 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 rechecks in the `OriginMod` module are analogous.

The more expressive programming idioms must use extenders, which can now resolve the problems from Section 4.1 that System M could not solve. In Fig. 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 a nonlocal specializer on their `equal` method (the `ColorPointMod` module still need only import the `ColorMod` module). Further, if the two

```

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

```

FIG. 7. Extension modules.

modules are combined in the same program, then the link-time check for most extending modules will fail for the `PointMod` module, forcing an additional module to be written that extends both the `ColorPointMod` and the `OriginMod` modules. 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 Fig. 7.

An analogous solution is used to resolve the problem illustrated in Fig. 5. Assuming the `print` generic function's first argument position is marked, `print` violates restriction *E5* 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, noninterface objects without also descending from some noninterface descendant of both of these 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 Fig. 7. Further, if the two modules in Fig. 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 Fig. 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 Fig. 6, the `erase` generic function, assuming that its first argument position is marked, violates restriction *E5* by having the imported `graphic` object on a marked position in its arrow object. Therefore, the `EraseMod` module must extend `GraphicMod`. Then, by restriction *E4a*, 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.

Finally, Fig. 8 illustrates the use of restriction *E4b*, which resolves ambiguity problems that can arise from the combination of abstract objects and the power of extenders. As before, the `GraphicMod`

```

module AbstractPointMod imports GraphicMod {
  abstract object point
  draw has method(p@point, d) { ... }
}
module ColorPointMod imports AbstractPointMod {
  object colorPoint isa point
}
module BadMod imports AbstractPointMod extends GraphicMod {
  draw has method(p@point, d) { ... }
}

```

FIG. 8. Using restriction *E4b* to resolve ambiguities.

module creates an abstract `graphic` object with an associated `draw` generic function. The `AbstractPointMod` module creates an abstract point child of `graphic`, along with an implementation for drawing points. This implementation is inherited for use by `colorPoint`, a concrete child of `point` declared in the `ColorPointMod` module. The `BadMod` module creates a second implementation for drawing points.

Without restriction **E4b**, all modules in Fig. 8 pass the System E restrictions. In particular, since the `BadMod` module extends the `GraphicMod` module, the `BadMod` module may add arbitrary multimethods to the `draw` generic function. Although both `draw` methods for `point` are visible in the `BadMod` module, `point` is not considered during implementation-side typechecking of `draw` in the `BadMod` module, because `point` is abstract. Therefore, the ambiguity for `draw(point)` is not detected statically, and at run-time the message `send draw(colorPoint)` will cause a message-ambiguous error. Restriction **E4b** forces the implementation-side typechecking of `draw` from the `BadMod` module to consider the `point` object to be concrete, thereby statically detecting the ambiguity for `draw(point)`.<sup>4</sup> Because restriction **E4b** checks `draw(point)` for ambiguity, but not for incompleteness, the check would succeed if both `draw` methods in Fig. 8 were removed. In that case, since `colorPoint` is an orphan, rechecks of the `draw` generic function from the `ColorPointMod` module would find the incompleteness for `draw(colorPoint)`, forcing an appropriate implementation to be written.

Conceptually, 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 rechecks its extended modules. The modular checks in the most extending module of a region ensure the safety of that region. 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

System M maximizes Dubious’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 Dubious’s expressiveness, at the cost of some regional typechecking, a simple link-time check for 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 section, 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 have arbitrary multimethods 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–E5**, 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**, **E4**, and **E5** are only those generic functions that have at least one marked argument position. The combination of restrictions **M2** and **E2** disallows unanticipated multiple implementation inheritance even in extenders.

This system gives implementors control over the tradeoff between flexibility and modularity 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 Fig. 3, retaining the use of arbitrary multimethods. At the same time, by using System M’s rules for the `erase` generic function in Fig. 6, we can keep the open object idiom in importers, with completely modular typechecking.

<sup>4</sup> Rather than introducing a special restriction to handle this problem, we could alternatively extend the definition of implementation-side typechecking such that argument tuples containing at least one nonconcrete object are checked for ambiguity, in addition to the current implementation-side requirement that argument tuples containing only concrete objects have a most specific applicable method.

TABLE 1

Overview of the Expressiveness and Modularity of the Various Type Systems Discussed

	Traditional object-oriented languages	M	E		ME		G
			Importers	Extenders	Importers	Extenders	
Single dispatching	X	X	X	X	X	X	X
Abstract objects	X	X	X	X	X	X	X
All-local multimethods		X	X	X	X	X	X
First-local 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	local	local	local	regional	local	regional	global

#### 4.5. Summary

Table 1 summarizes the expressiveness and modularity of the various type systems we have described. First we rate traditional object-oriented languages, which provide single dispatching, abstract object idioms, and arbitrary multiple inheritance, all with completely modular typechecking. Our System M augments the power of traditional object-oriented languages with all-local and first-local multimethods. In addition, System M provides arbitrary open objects, allowing both new descendants and new operations of existing objects to be added without modifying existing code. At the same time, System M retains completely modular typechecking. The cost of the open object idioms is a loss of unanticipated multiple implementation inheritance. The expressive power of the locally checked importers of System E is more limited than in System M, but importers can still express single dispatching and abstract object idioms, as well as the important multimethod idiom of binary methods. System E’s extenders allow all idioms to be expressed, including arbitrary multimethods and multiple implementation inheritance, at the cost of regional typechecking and a small link-time check. 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 creator of a generic function must decide *a priori* which kinds of extensibility are needed and thus which kinds of restrictions to impose. Finally, System G is the type system used for all previous languages with symmetric multimethods, allowing all idioms to be expressed at the cost of a global typechecking algorithm.

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 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 allow 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 incorporate Dubious’s 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 support 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 (modules may not 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 these restrictions, allowing arbitrary nesting of modules and other declarations. For example, Dubious’s program, module, and declaration forms could be reorganized as follows:

$$\begin{aligned}
 P &::= BE \\
 B &::= D_1 \dots D_n \\
 D &::= Q \text{ \textbf{object} } I \text{ \textbf{isa} } O_1, \dots, O_n \\
 &\quad | I \text{ \textbf{has method} } (F_1, \dots, F_n) \{BE\} \\
 &\quad | \text{ \textbf{module} } I \text{ \textbf{extends} } I_1, \dots, I_m \{B\} \\
 &\quad | \text{ \textbf{import} } I
 \end{aligned}$$

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.

Figure 9 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-coordinate appropriately. The `curriedequal` 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 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. Figure 10 shows an example of a mutable variable representing the number of elements in a stack.

```

module PointMod {
  import GraphicMod
  object point
  ...
  object newPoint isa (int)→point
  newPoint has method(newx) {
    object newp isa point
    x has method(p@newp) { newx }
    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) { eq_int(x(p1), x(p2)) }
    eq_p1 }
}
import PointMod
curried_equal(newPoint(one))(newPoint(two))

```

FIG. 9. Nested declarations in an extension of Dubious.

```

object stackSize isa ()→int -- declare variable
screenSize has method() { zero } -- set initial value
object update isa (int)→void
update has method(i) {
  stackSize has method() { i } } -- update value
...
stackSize() -- read current value

```

FIG. 10. Programming mutable variables via nested declarations.

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.)

## 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:

$$D ::= \dots \mid I \text{ **has method** } (I_1 @ = E_1, \dots, I_n @ = E_n) \{ BE \}$$

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 new  $@ =$  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 set isa (point, int)→point
set has method (myP, newX) {
  x has method (p@=myP) {newX}
  myP}

```

By requiring that all argument positions use  $@ =$ , we ensure that the method is applicable to only a single argument tuple. Consequently, there can be no possibility of ambiguity between this method and any other method declaration, so implementation-side typechecking can ignore  $@ =$  methods.<sup>5</sup>

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 object. 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 the combination of subsumption and method update.

<sup>5</sup> There is no conflict in having one  $@$  and one  $@ =$  method, each specialized to the same objects; both methods are applicable to that tuple of objects, but the  $@ =$  method is treated as the more specific method. The  $@$  method will still be applicable to any descending tuples.

### 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 information about 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 the `ImpMod` module 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. Our example above can be fixed using this approach to encapsulation in System M. In particular, 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 `BadMod` module. Therefore, the `badFun` 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 message-not-understood error that eludes typechecks from the `ImpMod` module.

### 5.4. Predicate Dispatching

A recent paper described predicate dispatching (Ernst *et al.*, 1998), 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 a natural translation of our current restrictions to the more general model. For example, System M requires that a method added to an imported generic function be a first-local 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 argument tuples to which the method is applicable.

## 6. RELATED WORK

Chambers and Leavens made the first effort toward modular typechecking of symmetric multimethods (1995) in the Cecil language (Chambers, 1992, 1995). 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 and Leavens, 1997) 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  $\lambda\&$ -calculus and variants (Castagna *et al.*, 1992; Castagna, 1997) 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  $\lambda$ .object augments the calculi with the ability to define new types and subtyping relations. Neither the  $\lambda\&$ -calculi nor  $\lambda$ .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 independent code development.

$ML_{\leq}$  (Bourdoncle and Merz, 1997) 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.*, 1991) 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 and Millstein, 1998) 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, 1995; Bruce *et al.*, 1995) 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's 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 or abstract method idioms. Parasitic multimethods (Boylard and Castagna, 1997) 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, 1990; Paepcke, 1993) and Dylan (Shalit, 1997; Feinberg *et al.*, 1997) 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.*, 1991) 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.*, 1992; Reynolds, 1980) 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 and Cook, 1990; Flatt *et al.*, 1998) and role-based programming (Reenskaug *et al.*, 1992; VanHilst and Notkin, 1996; Smaragdakis and Batory, 1998). Finally, we are using the ideas in Dubious as a foundation for the design of Diesel, a practical programming language succeeding Cecil.

## APPENDIX A. DYNAMIC SEMANTICS

This section presents the formal dynamic semantics of Dubious. Section A.1 presents the dynamic semantics corresponding to the syntax used in Systems G and M, and Section A.2 presents the modifications to the dynamic semantics for the augmented syntax necessary for Systems E and ME.

### A.1. Dynamic Semantics for Systems G and M

#### A.1.1. Preliminaries

Figure A.1 defines the necessary domains for the dynamic semantics. The notation  $\wp(Dom)$  denotes a set of elements from the domain  $Dom$ , while  $Dom^*$  represents a (ordered) list of elements from the domain  $Dom$ . The notation  $Dom_1 \times Dom_2$  represents a pair whose first component is an element of  $Dom_1$  and second component is an element of  $Dom_2$ . The notation  $Dom_1 + Dom_2$  represents an element that is either from the domain  $Dom_1$  or from the domain  $Dom_2$ . The notation  $Dom_1 \rightarrow Dom_2$  denotes a function from elements of  $Dom_1$  to elements of  $Dom_2$ ; such functions are sometimes manipulated as sets of the form  $\wp(Dom_1 \times Dom_2)$ .

The domains  $I$  and  $E$  in Fig. A.1 refer to the syntactic domains defined in Fig. 1. The  $Env$  domain represents the dynamic environment, mapping identifiers to values.  $Store$  maintains information on the inheritance relation among objects and the methods contained in each generic function.  $ModEnv$  is a mapping from each module name to an environment containing the names of the values available

$e$	$\in$	$Env$	$=$	$I \rightarrow Val$
$s$	$\in$	$Store$	$=$	$Isa \times GFMethods$
$me$	$\in$	$ModEnv$	$=$	$I \rightarrow Env$
$isa$	$\in$	$Isa$	$=$	$\wp(Val \times Obj)$
$gfms$	$\in$	$GFMethods$	$=$	$((Val \times MethodHeader) \times MethodBody)$
$mh$	$\in$	$MethodHeader$	$=$	$Obj^*$
$mb$	$\in$	$MethodBody$	$=$	$I^* \times E \times Env$
$o$	$\in$	$Obj$	$=$	$Val + Arrow$
$arr$	$\in$	$Arrow$	$=$	$\mathbf{arrow}(Obj^*, Obj)$
$v$	$\in$	$Val$	$=$	$\mathbf{obj}(Nat)$

FIG. A.1. Domains for the dynamic semantics.

in that module.  $Isa$  is the domain representing the declared inheritance relationship. The  $GFMethods$  domain contains the relevant information about each method in a generic function object; the  $Val$  component in the definition of  $GFMethods$  is the generic function object, the  $MethodHeader$  is the list of specializer objects of the method, and the  $MethodBody$  is the list of formal parameter names, method body, and lexical environment of the method.  $Obj$  is the domain of all objects in the program, including both arrow and nonarrow objects.  $Arrow$  is the domain of arrow objects; an object of the form  $\mathbf{arrow}([o_1, \dots, o_n], o)$  is an arrow object with argument objects  $[o_1, \dots, o_n]$  and result object  $o$ .  $Val$  is the domain of all nonarrow objects in the program. This domain contains all possible program values, the entities which may be passed to and returned from functions. An object of the form  $\mathbf{obj}(i)$  is a nonarrow object with unique identity  $i$ , a natural number. Unique identities are given to each nonarrow object in order to distinguish among the nonarrow objects in a program.

Figure A.2 gives several useful definitions and functions. Given a  $Store$   $s$ ,  $isa(s)$  extracts the first component of  $s$ , while  $gfms(s)$  extracts the second component of  $s$ . The  $\&$  function is a *shadowing union* operator on two relations (sets of ordered pairs), favoring the second relation. The  $+$  function is a pointwise union on two tuples of sets. The  $name$  function extracts the name from a possibly specialized formal argument. The  $disjoint$  function returns true if and only if all of its arguments are distinct.

### A.1.2. Inference Rules

Now we present the inference rules for the dynamic semantics. In general, a judgment may have the form  $d_1, \dots, d_m \vdash X \Rightarrow o \succ d_1', \dots, d_n'$ . Such a judgment is interpreted as follows: “Given the information in domain elements  $d_1, \dots, d_m$  as the context for evaluation, the program fragment  $X$  evaluates to the object  $o$  and produces  $d_1', \dots, d_n'$  as additions to the current context.” The  $\Rightarrow o$  or  $\succ d_1', \dots, d_n'$  parts may be omitted.

As mentioned above, each nonarrow object is given a unique identity, a natural number. To do this, we assume that each textual occurrence of object in a Dubious program is subscripted with a unique natural number. The inference rules that follow then use an object’s subscript as its unique identity. This is a simple alternative to keeping a global counter in the dynamic semantics in order to provide unique object identities.

Figure A.3 contains the inference rules for programs and modules. The program rule evaluates each module in the program, returning a global store and an environment for each module. The rule then evaluates the given expression in the context of the global store and the environment of the imported module. A list of modules is evaluated sequentially. The declarations in each module are evaluated in the context of the environments of all modules being imported.

We define accessor functions on the components of  $Store$ , with the names  $isa$  and  $gfms$  respectively.

$$\begin{aligned}
 r_1 \& r_2 &= \{(x, y_1) \mid (x, y_1) \in r_2 \vee (x, y_1) \in r_1 \wedge \neg \exists y_2. (x, y_2) \in r_2\} \\
 (s_{11}, \dots, s_{1n}) + (s_{21}, \dots, s_{2n}) &= (s_{11} \cup s_{21}, \dots, s_{1n} \cup s_{2n}) \\
 name(I_1 @ I_2) &= I_1 & name(I_1) &= I_1 \\
 disjoint(x_1, \dots, x_n) &= \forall 1 \leq i \leq n. \forall 1 \leq j \leq n. (x_i = x_j) \Rightarrow (i = j)
 \end{aligned}$$

FIG. A.2. Definitions and functions.

[prg-d]	$\frac{\vdash M_1 \dots M_n \succ s, me \quad e, s \vdash E \Rightarrow v}{\vdash M_1 \dots M_n \text{ import } I \text{ in } E \text{ end} \Rightarrow v \succ s, me}$	where $n \geq 0$ , $I \in \text{domain}(me)$ , $e = me(I)$
[md*-d]	$\frac{\begin{array}{c} (\{\cdot\}) \vdash M_1 \succ s_1, me_1 \\ s_1, me_1 \vdash M_2 \succ s_2, me_2 \quad \dots \quad s_1 + \dots + s_{n-1}, me_1 \& me_2 \& \dots \& me_{n-1} \vdash M_n \succ s_n, me_n \end{array}}{\vdash M_1 \dots M_n \succ s_1 + s_2 + \dots + s_n, me_1 \& me_2 \& \dots \& me_n}$	where $n \geq 0$
[mod-d]	$\frac{e, s \vdash D_1 \dots D_n \succ e_0, s_0}{s, me \vdash \text{module } I \text{ imports } I_1, \dots, I_n \{D_1 \dots D_n\} \succ s_0, \{(I, e \& e_0)\}}$	where $n \geq 0$ , $t \geq 0$ , $I_1 \in \text{domain}(me)$ , $I_t \in \text{domain}(me)$ , $e = me(I_1) \& \dots \& me(I_t)$

FIG. A.3. Inference rules for programs and modules.

The inference rules for declarations are shown in Fig. A.4. Each declaration in a declaration list is evaluated in an environment which includes any name bindings from previous declarations in the list. Although declaration lists are not recursive, it is still possible to write (mutually) recursive methods because generic functions are declared separate from their methods. For example, the body of a method  $m$  may refer to the same generic function to which  $m$  was added, since the generic function was declared previously. The **object** declaration is evaluated by evaluating each inheritance parent of the new object and adding the new inheritance pairs to the resulting store. Methods are evaluated by evaluating each specialization and recording the appropriate information in the *GFMethods* component of the resulting store.

Figure A.5 contains the inference rules for expressions and objects. A generic function application is evaluated by evaluating the generic function expression and the actual argument expressions to objects. The most specific applicable method for the argument objects is extracted from the generic function object, and the associated method body is then evaluated in the context of the method's lexical environment, augmented with bindings from the formal to the actual parameters. An identifier is evaluated simply by looking up the identifier's binding in the current environment. An arrow object is evaluated by evaluating each of its subobjects.

The inference rules for evaluating formal argument specializers are given in Fig. A.6. If the formal is specialized, the specializer object is evaluated and returned. If the formal is unspecialized, we consider the formal to be implicitly specialized on the associated object in an arrow object of the generic function (the static semantics will ensure that there is at most one such arrow object for each generic function).

The method lookup rule appears in Fig. A.7. The rule first creates a list consisting of the tuple of specializers of each method added to the generic function  $v$  that is applicable to  $[v_1, \dots, v_m]$ . A method is applicable to  $[v_1, \dots, v_m]$  if  $[v_1, \dots, v_m]$  pointwise descends from the method's tuple of specializers. After ensuring that the list contains no duplicates, the rule finds the unique tuple of specializers in the list,  $mh_i$ , that is more specific than all other tuples of specializers in the list. Finally, the rule finds and returns the *MethodBody* of the method corresponding to the specializer tuple  $mh_i$ .

[dc*-d]	$\frac{\begin{array}{c} e, s \vdash D_1 \succ e_1, s_1 \\ e \& e_1, s + s_1 \vdash D_2 \succ e_2, s_2 \\ e \& e_1 \& e_2 \& \dots \& e_{n-1}, s + s_1 + s_2 + \dots + s_{n-1} \vdash D_n \succ e_n, s_n \end{array}}{e, s \vdash D_1 \dots D_n \succ e_1 \& e_2 \& \dots \& e_n, s_1 + s_2 + \dots + s_n}$	where $n \geq 0$
[obj-d]	$\frac{e \vdash O_1 \Rightarrow o_1 \quad \dots \quad e \vdash O_n \Rightarrow o_n}{e, s \vdash Q \text{ object}; I \text{ isa } O_1, \dots, O_n \succ \{(I, v)\}, \{(v, o_1)\}, \dots, \{(v, o_n)\}, \{\cdot\}}$	where $n \geq 0$ , $v = \text{obj}(i)$
[has-d]	$\frac{e \vdash I \Rightarrow v \quad e, s \vdash (v, [F_1, \dots, F_n]) \Rightarrow (o_1, \dots, o_n)}{e, s \vdash I \text{ has method } (F_1, \dots, F_n) \{E\} \succ \{\cdot\}, \{\cdot\}, \{(v, [o_1, \dots, o_n]), (I^*, E, e)\}}$	where $n \geq 0$ , $I^* = \text{name}(F_1), \dots, \text{name}(F_n)$

FIG. A.4. Inference rules for declarations.

$$\begin{array}{c}
 \text{[app-d]} \quad \frac{e, s \vdash E_0 \Rightarrow v_0 \quad \dots \quad e, s \vdash E_n \Rightarrow v_n \quad s \vdash v_0 \text{ lookup-method}[v_1, \dots, v_n] \triangleright ([I_1, \dots, I_n], E, e_0) \quad e_0 \& \{(I_1, v_1), \dots, (I_n, v_n)\}, s \vdash E \Rightarrow v}{e, s \vdash E_0(E_1, \dots, E_n) \Rightarrow v} \quad \text{where } n \geq 0 \\
 \\
 \text{[id-d]} \quad \frac{}{e, s \vdash I \Rightarrow v} \quad \text{where } I \in \text{domair}(e), e(I) = v \\
 \\
 \text{[arr-d]} \quad \frac{e \vdash O_1 \Rightarrow o_1 \quad \dots \quad e \vdash O_n \Rightarrow o_n \quad e \vdash O \Rightarrow o}{e \vdash (O_1, \dots, O_n) \rightarrow O \Rightarrow \mathbf{arrow}([o_1, \dots, o_n], o)} \quad \text{where } n \geq 0
 \end{array}$$

FIG. A.5. Inference rules for expressions and objects.

$$\begin{array}{c}
 \text{[form-d]} \quad \frac{e, s \vdash (o, F_1, o_1) \triangleright o_1' \quad \dots \quad e, s \vdash (o, F_n, o_n) \triangleright o_n'}{e, s \vdash (o, [F_1, \dots, F_n]) \triangleright [o_1', \dots, o_n']} \quad \text{where } n \geq 0, (o, \mathbf{arrow}([o_1, \dots, o_n], o_0)) \in \text{isa}(s) \\
 \\
 \text{[spe-d]} \quad \frac{e, s \vdash I_2 \Rightarrow o_2}{e, s \vdash (o, I_1 @ I_2, o') \Rightarrow o_2} \\
 \\
 \text{[uns-d]} \quad \frac{}{e, s \vdash (o, I, o') \triangleright o'}
 \end{array}$$

FIG. A.6. Inference rules for formal arguments.

$$\text{[look-d]} \quad \frac{s \vdash mh_i \leq_{\text{isa}^*} mh_1 \quad \dots \quad s \vdash mh_i \leq_{\text{isa}^*} mh_n}{s \vdash v \text{ lookup-method}[v_1, \dots, v_m] \triangleright mb_i} \quad \begin{array}{l} \text{where } n \geq 1, m \geq 0, \\ [mh_1, \dots, mh_n] = [mh]((v, mh), mb) \in \text{gfm}(s) \wedge \\ s \vdash [v_1, \dots, v_m] \leq_{\text{isa}^*} mh \\ \text{disjoint}([mh_1, \dots, mh_n]) \\ \exists i \ 1 \leq i \leq n \wedge ((v, mh), mb) \in \text{gfm}(s) \end{array}$$

FIG. A.7. Inference rule for method lookup.

$$\begin{array}{c}
 \text{[isa*-d]} \quad \frac{s \vdash o_1 \leq_{\text{isa}} o_1' \quad \dots \quad s \vdash o_n \leq_{\text{isa}} o_n'}{s \vdash [o_1, \dots, o_n] \leq_{\text{isa}^*} [o_1', \dots, o_n']} \quad \text{where } n \geq 0 \\
 \\
 \text{[isb-d]} \quad \frac{}{s \vdash o_1 \leq_{\text{isa}} o_2} \quad \text{where } (o_1, o_2) \in \text{isa}(s) \\
 \\
 \text{[isr-d]} \quad \frac{}{s \vdash o \leq_{\text{isa}} o} \\
 \\
 \text{[ist-d]} \quad \frac{s \vdash o_1 \leq_{\text{isa}} o_2 \quad s \vdash o_2 \leq_{\text{isa}} o_3}{s \vdash o_1 \leq_{\text{isa}} o_3} \\
 \\
 \text{[isar-s]} \quad \frac{s \vdash o_1' \leq_{\text{isa}} o_1 \quad \dots \quad s \vdash o_n' \leq_{\text{isa}} o_n \quad s \vdash o \leq_{\text{isa}} o'}{s \vdash \mathbf{arrow}([o_1, \dots, o_n], o) \leq_{\text{isa}} \mathbf{arrow}([o_1', \dots, o_n'], o')} \quad \text{where } n \geq 0
 \end{array}$$

FIG. A.8. Inference rules for the descendant relation.

$$\begin{array}{c}
 \text{[mod-d]} \quad \frac{e, s \vdash D_1 \dots D_n \triangleright e_0, s_0}{s, me \vdash \mathbf{module} \ I \ \mathbf{imports} \ I_1, \dots, I_t \ \mathbf{extends} \ I_1', \dots, I_r' \ \{D_1 \dots D_n\} \triangleright s_0, \{(I, e \& e_0)\}} \quad \begin{array}{l} \text{where } n \geq 0, t \geq 0, r \geq 0, \\ I_1 \in \text{domair}(me), \dots, \\ I_t \in \text{domair}(me), \\ I_1' \in \text{domair}(me), \dots, \\ I_r' \in \text{domair}(me), \\ e = me(I_1) \& \dots \& me(I_t) \& \\ me(I_1') \& \dots \& me(I_r') \end{array} \\
 \\
 \text{[arr-d]} \quad \frac{e \vdash O_1 : o_1 \quad \dots \quad e \vdash O_n : o_n \quad e \vdash O : o}{e \vdash (H_1 O_1, \dots, H_n O_n) \rightarrow O : \mathbf{arrow}([o_1, \dots, o_n], o)} \quad \text{where } n \geq 0
 \end{array}$$

FIG. A.9. Inference rule modifications for Systems E and ME.

The rules for extending the direct inheritance relation to form the descendant relation are given in Fig. A.8. The  $\leq_{isa^*}$  relation is a pointwise extension of the  $\leq_{isa}$  relation. The  $\leq_{isa}$  relation is the reflexive, transitive closure of the declared inheritance relation, along with the standard contravariant rule for relating arrow objects.

## A.2. Modifications for Systems E and ME

Two minor modifications of the dynamic semantics are needed to accommodate the syntax extensions for Systems E and ME. These modified rules are shown in Fig. A.9. The modified rule for modules treats extended modules identically to imported modules, using their environments as context for the evaluation of the current module’s declarations. The rule for evaluating arrow objects simply ignores the optional # markers on argument objects.

## APPENDIX B. STATIC SEMANTICS

This section presents the formal static semantics of Dubious. Section B.1 presents the base static semantics, which is independent of the particular type system—G, M, E, or ME—used. Sections B.2 through B.5 present the modifications and additions to the base static semantics for Systems G, M, E, and ME, respectively.

### B.1. Base Static Semantics

#### B.1.1. Preliminaries

Figure B.1 defines the necessary domains for the static semantics. *TypeEnv* is the static analog of the dynamic environment, mapping object identifiers to their types. (The name  $p$  for an element of *TypeEnv* is vaguely similar to  $\Gamma$ , which is sometimes used in the literature to represent the type environment.) *TypeStore* is the static analog of the program store. (The name  $k$  for an element of *TypeStore* is meant to stand for “context,” since the type store is the static context in which a program is typechecked.) A *TypeStore* contains all the information about objects and methods necessary to enforce the static type restrictions. In particular, the *TypeStore* domain maintains information on the inheritance relation among objects, which objects are concrete, which objects are abstract, the methods contained in each generic function, and local information about the current module being typechecked. *Modules* maintains a mapping from each module name to its associated type environment and a mapping from each module name to its associated type store. The *GFMETHODTYPES* domain maintains, for each method in the program, a pair of the method’s generic function object and the method’s tuple of specializers. *Locals*

$p$	$\in$ <i>TypeEnv</i>	$=$ $I \rightarrow \text{ObjType}$
$k$	$\in$ <i>TypeStore</i>	$=$ $\text{Isa} \times \text{Concrete} \times \text{Abstract} \times \text{GFMethodTypes} \times \text{LocalTypeStore}$
$m$	$\in$ <i>Modules</i>	$=$ $\text{ModTypeEnv} \times \text{ModTypeStore}$
$cnc$	$\in$ <i>Concrete</i>	$=$ $\wp(\text{Const})$
$abs$	$\in$ <i>Abstract</i>	$=$ $\wp(\text{Const})$
$isa$	$\in$ <i>Isa</i>	$=$ $\wp(\text{Const} \times \text{ConstType})$
$gfms$	$\in$ <i>GFMethodTypes</i>	$=$ $\wp(\text{Const} \times \text{MethodHeaderType})$
$mh$	$\in$ <i>MethodHeaderType</i>	$=$ $\text{ConstType}^*$
$lk$	$\in$ <i>LocalTypeStore</i>	$=$ $\text{Locals} \times \text{LocalGFMethods}$
$ls$	$\in$ <i>Locals</i>	$=$ $\wp(\text{Const})$
$lgf$	$\in$ <i>LocalGFMethods</i>	$=$ <i>GFMethodTypes</i>
$mp$	$\in$ <i>ModTypeEnv</i>	$=$ $I \rightarrow \text{TypeEnv}$
$mk$	$\in$ <i>ModTypeStore</i>	$=$ $I \rightarrow \text{TypeStore}$
$t$	$\in$ <i>Type</i>	$=$ $\text{ObjType} + \text{ArrowType}$
$ot$	$\in$ <i>ObjType</i>	$=$ $\text{Const} + \mathbf{unk}(\text{ConstType})$
$art$	$\in$ <i>ArrowType</i>	$=$ $\mathbf{arrow}(\text{ConstType}^*, \text{ConstType})$
$ct$	$\in$ <i>ConstType</i>	$=$ $\text{Const} + \text{ArrowType}$
$c$	$\in$ <i>Const</i>	$=$ $\mathbf{object}(\text{Nat})$

FIG. B.1. Domains for the static semantics.

We define names for the components of a *TypeStore* and *Module*, and associated accessor functions with those names:  $(isa, cnc, abs, gfms, (ls, lgf))$  and  $(mp, mk)$

Let  $()$  denote the empty *TypeStore* and  $(x_1 = v_1, \dots, x_n = v_n)$  denote the empty *TypeStore* augmented with component  $x_i$  equal to  $v_i$ .

$import-store(isa, cnc, abs, gfms, lk) = (isa, cnc, abs, gfms, (\{\}, \{\}))$

$name(\mathbf{module} \ I \ \mathbf{imports} \ I_1, \dots, I_t \ \{D^*\}) = I$

$name(\mathbf{module} \ I \ \mathbf{imports} \ I_1, \dots, I_t \ \mathbf{extends} \ I'_1, \dots, I'_r \ \{D^*\}) = I$

$length([x_1, \dots, x_n]) = n$

FIG. B.2. Definitions and functions.

records all nonarrow objects created in the current module being typechecked, and *LocalGFMethods* records all methods created in the current module being typechecked.

Several domains represent the types of objects. The *Type* domain encompasses the types of both nonarrow and arrow objects. The *ObjType* domain represents the types of nonarrow objects. A nonarrow object can have one of two kinds of types, depending on whether or not the object is statically known. The domains *Const* represents the types of statically known nonarrow objects. A *Const* type is precise. In particular, if an object has a type of the form **object**( $i$ ), then the object is known to be exactly the object **object**( $i$ ), where  $i$  is the object's unique identity. On the other hand, if an object has the type **unk**( $t$ ), then the object is known only to be a descendant of  $t$ . These “unknown” object types are used for values that cannot in general be statically known, such as the result of a function application. A similar distinction between precise and imprecise types is presented in Bruce *et al.*, (1997). The *ArrowType* domain represents the types of arrow objects. Arrow objects in Dubious are always statically known, so types of the form **unk**( $t$ ) do not appear within arrow types. Finally, the *ConstType* domain represents all statically known objects, both arrow and nonarrow.

Figure B.2 gives several useful definitions and functions, augmenting those of Fig. A.2. As in the dynamic semantics, we define accessor functions for easy manipulation of the *TypeStore* and *Modules* domains. In addition, we define a record-like syntax for representing sparse *TypeStores*, where most of the components are empty. This is useful because most judgments modify only one or two components of the current *TypeStore*, leaving the rest unchanged. The *import-store* function is used to obtain the necessary information from the type store of an imported module, and this information is then added to the type store of the importer. The function simply ignores the *LocalTypeStore* component of the imported module's *TypeStore*, while copying the rest. The *name* function extracts the name of a module, and the *length* function returns the number of elements in the given list.

### B.1.2. Inference Rules

Now we present the inference rules for the base static semantics. In general, a judgment may have the form  $d_1, \dots, d_m \vdash X : t \succ d'_1, \dots, d'_n$ . Such a judgment is interpreted as follows: “Given the information in domain elements  $d_1, \dots, d_m$  as the context for typechecking, the program fragment  $X$  has type  $t$  and produces  $d'_1, \dots, d'_n$  as additions to the current context.” The:  $t$  or  $\succ d'_1, \dots, d'_n$  parts may be omitted.

Figure B.3 contains the inference rules for programs and modules. The program rule typechecks each module in the program, returning a global type store as well as a type environment and type store for each module. The rule then typechecks the given expression in the context of the global type store and the type environment of the imported module. A list of modules is typechecked sequentially, and the global type store of the program is the pointwise union of the type stores of each module. The *program-has-safe-modules* judgment performs any necessary global typechecking. The rule for this judgment is supplied by each particular type system—G, M, E, or ME—being added to these base semantics, as described in Sections B.2–B.5. The declarations in a module are typechecked in the context of the type environments and type stores of each imported module. The four *module-has-safe-x* rules will be supplied by each particular type system, performing any necessary local and regional typechecking.

The inference rules for declarations are shown in Fig. B.4. Each declaration in a declaration list is typechecked in the context of the name bindings and type store produced from typechecking all preceding declarations in the list. The [obj-s] rule typechecks objects that are not acting as generic functions. In particular, the rule checks that none of the object's ancestors (either declared or inherited) are arrow

[prg-s]	$\frac{\vdash M_1 \dots M_n \succ k, m \quad p, k \vdash E : \mathbf{unk}(t)}{\vdash M_1 \dots M_n \mathbf{import} \ I \ \mathbf{in} \ E \ \mathbf{end} : \mathbf{unk}(t) \succ k, m}$	where $n \geq 0$ , $I \in \mathit{domair}(mp(m))$ , $p = (mp(m))(I)$
[md*-s]	$\frac{\begin{array}{l} (\{\}, \{\}) \vdash M_1 \succ m_1 \\ m_1 \vdash M_2 \succ m_2 \dots m_1 + m_2^+ \dots + m_{n-1} \vdash M_n \succ m_n \\ k, m \vdash \mathit{program}\text{-has}\text{-safe}\text{-modules} \end{array}}{\vdash M_1 \dots M_n \succ k, m}$	where $n \geq 0$ , $m = m_1 + m_2^+ \dots + m_n$ , $\mathit{name}(M_1) = I_1, \dots, \mathit{name}(M_n) = I_n$ , $\mathit{disjoint}(I_1, \dots, I_n)$ , $I_1 \in \mathit{domair}(mk(m)), \dots, I_n \in \mathit{domair}(mk(m))$ , $k = mk(m)(I_1) + \dots + mk(m)(I_n)$
[mod-s]	$\frac{\begin{array}{l} p, k \vdash D_1 \dots D_n \succ p_0, k_0 \\ k+k_0 \vdash \mathit{module}\text{-has}\text{-safe}\text{-objects} \\ k+k_0 \vdash \mathit{module}\text{-has}\text{-safe}\text{-methods} \\ k+k_0 \vdash \mathit{module}\text{-has}\text{-safe}\text{-gfs} \\ k+k_0 \vdash \mathit{module}\text{-has}\text{-safe}\text{-imported}\text{-gfs} \end{array}}{m \vdash \mathbf{module} \ I \ \mathbf{imports} \ I_1, \dots, I_n \{D_1 \dots D_n\} \\ \succ \{((I, p)), \{(k+k_0)\}\}}$	where $n \geq 0$ , $t \geq 0$ , $I_1 \in \mathit{domair}(mp(m)), \dots, I_t \in \mathit{domair}(mp(m))$ , $I_{t+1} \in \mathit{domair}(mk(m)), \dots, I_n \in \mathit{domair}(mk(m))$ , $p = mp(m)(I_1) \& \dots \& mp(m)(I_t)$ , $k = \mathit{import}\text{-store}(mk(m)(I_1)) + \dots +$ $\mathit{import}\text{-store}(mk(m)(I_t))$

FIG. B.3. Inference rules for programs and modules.

objects. The rule simply evaluates each declared inheritance parent and stores the new inheritance pairs in the *Isa* component of the type store. It also adds the new object to *Locals*, indicating that the object was created in the current module. The [gf-s] rule typechecks generic functions. It is similar to the [obj-s] rule, except that it ensures that the new generic function has exactly one most specific ancestor arrow object. All of the declared nonarrow inheritance parents of the new generic function, as well as its most specific arrow object, are recorded as its parents in the *Isa* component of the type store. The

[dc*-s]	$\frac{\begin{array}{l} p, k \vdash D_1 \succ p_1, k_1 \\ p \& p_1, k+k_1 \vdash D_2 \succ p_2, k_2 \\ \dots \\ p \& p_1 \& p_2 \& \dots \& p_{n-1}, k+k_1+k_2+\dots+k_{n-1} \vdash D_n \succ p_n, k_n \end{array}}{p, k \vdash D_1 \dots D_n \succ p_1 \& p_2 \& \dots \& p_n, k_1+k_2+\dots+k_n}$	where $n \geq 0$
[obj-s]	$\frac{\begin{array}{l} p, k \vdash O_1 : t_1 \dots p, k \vdash O_n : t_n \\ p, k \vdash \mathbf{interface} \ \mathbf{object}_i \ I \ \mathit{isa} \ O_1, \dots, O_n \\ \succ \{(l, c)\}, (\mathit{isa} = \{(c, t_1), \dots, (c, t_n)\}, \mathit{ls} = \{c\}) \end{array}}{\{ \} = \{t_j \mid 1 \leq j \leq n \wedge t_j \in \mathit{ArrowType}\}, \\ \{ \} = \{t \mid 1 \leq j \leq n \wedge (t_j, t) \in \mathit{isa}(k) \wedge t \in \mathit{ArrowType}\}, \\ c = \mathbf{object}(i)}$	where $n \geq 0$ , $\{ \} = \{t_j \mid 1 \leq j \leq n \wedge t_j \in \mathit{ArrowType}\}$ , $\{ \} = \{t \mid 1 \leq j \leq n \wedge (t_j, t) \in \mathit{isa}(k) \wedge t \in \mathit{ArrowType}\}$ , $c = \mathbf{object}(i)$
[gf-s]	$\frac{\begin{array}{l} p, k \vdash O_1 : t_1 \dots p, k \vdash O_n : t_n \\ k \vdash \mathit{art}_r \leq_{\mathit{isa}} \mathit{art}_1 \dots k \vdash \mathit{art}_r \leq_{\mathit{isa}} \mathit{art}_q \end{array}}{p, k \vdash \mathbf{interface} \ \mathbf{object}_i \ I \ \mathit{isa} \ O_1, \dots, O_n \\ \succ \{(l, c)\}, (\mathit{isa} = \{(c, c_1), \dots, (c, c_m), (c, \mathit{art}_r)\}, \mathit{ls} = \{c\})}$	where $n \geq 0$ , $m \geq 0$ , $q \geq 0$ , $\{c_1, \dots, c_m\} = \{t_j \mid 1 \leq j \leq n \wedge t_j \in \mathit{Const}\}$ , $\{\mathit{art}_1, \dots, \mathit{art}_q\} = \{t_j \mid 1 \leq j \leq n \wedge t_j \in \mathit{ArrowType}\} \cup$ $\{t \mid 1 \leq j \leq m \wedge (c_j, t) \in \mathit{isa}(k) \wedge t \in \mathit{ArrowType}\}$ , $\exists r. 1 \leq r \leq q, \quad c = \mathbf{object}(i)$
[absos]	$\frac{p, k \vdash \mathbf{interface} \ \mathbf{object}_i \ I \ \mathit{isa} \ O_1, \dots, O_n \succ p_0, k_0}{p, k \vdash \mathbf{abstract} \ \mathbf{object}_i \ I \ \mathit{isa} \ O_1, \dots, O_n \\ \succ p_0, k_0 + (\mathit{abs} = \{\mathbf{object}(i)\})}$	where $n \geq 0$
[cnco-s]	$\frac{p, k \vdash \mathbf{interface} \ \mathbf{object}_i \ I \ \mathit{isa} \ O_1, \dots, O_n \succ p_0, k_0}{p, k \vdash \mathbf{object}_i \ I \ \mathit{isa} \ O_1, \dots, O_n \\ \succ p_0, k_0 + (\mathit{cnc} = \{\mathbf{object}(i)\})}$	where $n \geq 0$
[has-s]	$\frac{\begin{array}{l} p, k \vdash I : c \\ p, k \vdash (c, [F_1, \dots, F_n]) \succ [t_1, \dots, t_n] \\ p \& \{(I, \mathbf{unk}(t_1)), \dots, (I_n, \mathbf{unk}(t_n)), k \vdash E : \mathbf{unk}(t)\} \end{array}}{p, k \vdash I \ \mathbf{has} \ \mathbf{method} \ (F_1, \dots, F_n) \ \{E\} \\ \succ \{ \}, (\mathit{gfs} = \{(c, [t_1, \dots, t_n])\}, \mathit{lgf} = \{(c, [t_1, \dots, t_n])\})}$	where $n \geq 0$ , $(c, \mathbf{arrow}(\{[t_1', \dots, t_n']\}, t)) \in \mathit{isa}(k)$ , $I_1 = \mathit{name}(F_1), \dots, I_n = \mathit{name}(F_n)$ , $\mathit{disjoint}(I_1, \dots, I_n)$

FIG. B.4. Inference rules for declarations.

$$\begin{array}{c}
 \text{[app-s]} \quad \frac{p, k \vdash E_0 : \mathbf{unk}(\mathbf{arrow}([t_1, \dots, t_n], t)) \quad p, k \vdash E_1 : \mathbf{unk}(t_1) \quad \dots \quad p, k \vdash E_n : \mathbf{unk}(t_n)}{p, k \vdash E_0(E_1, \dots, E_n) : \mathbf{unk}(t)} \quad \text{where } n \geq 0 \\
 \\
 \text{[id-s]} \quad \frac{}{p, k \vdash I : ot} \quad \text{where } I \in \mathit{domair}(p), \quad p(I) = ot \\
 \\
 \text{[smpc-s]} \quad \frac{p, k \vdash E : c \quad k \vdash c \leq_{isa} t}{p, k \vdash E : \mathbf{unk}(t)} \quad \text{where } c \in \mathit{cnc}(k) \\
 \\
 \text{[smpu-s]} \quad \frac{p, k \vdash E : \mathbf{unk}(t_1) \quad k \vdash t_1 \leq_{isa} t_2}{p, k \vdash E : \mathbf{unk}(t_2)} \\
 \\
 \text{[art-s]} \quad \frac{p, k \vdash O_1 : t_1 \quad \dots \quad p, k \vdash O_n : t_n \quad p, k \vdash O : t}{p, k \vdash (O_1, \dots, O_n) \rightarrow O : \mathbf{arrow}([t_1, \dots, t_n], t)} \quad \text{where } n \geq 0
 \end{array}$$

FIG. B.5. Inference rules for expressions and types.

[abso-s] and [cnco-s] rules are used as wrappers around the previous two rules, in order to correctly note in the type store if the new object is abstract or concrete. Methods are typechecked by typechecking each formal argument specializer, returning an associated specializer type. Then the method body is typechecked in the context of the current lexical type environment, augmented with type bindings for the formal arguments. The method body must have a type that is a descendant of the result type of the associated generic function’s arrow object.

Figure B.5 contains the inference rules for expressions and types. The [app-s] rule is the client-side typechecking rule for generic function applications. The result of a generic function application cannot be statically known in general, so the result type is always of the form  $\mathbf{unk}(t)$ . An identifier is typechecked simply by looking up the identifier’s binding in the current type environment. There are two subsumption rules, which allow the types of expressions to be “raised.” The [smpc-s] rule has a side condition ensuring that only concrete objects can have their types raised. Because generic function applications must return an object with an unknown type, thereby requiring the use of subsumption at some point in the typing derivation, this side condition has the effect of disallowing reference to nonconcrete objects in expressions. An arrow object is typechecked by typechecking each of its subobjects.

The inference rules for typechecking formal argument specializers are given in Fig. B.6. If the formal parameter is specialized, the specializer object is typechecked, and its type is returned. The specializer must not be an interface and it must be a descendant of the associated argument object of the generic function’s arrow object. If the formal parameter is unspecialized, we consider the formal to be implicitly specialized on the associated argument object in the generic function’s arrow object.

Figure B.7 shows the static analog of the dynamic method lookup rule in Fig. A.7. The boolean function returns true if and only if the generic function  $c$  has a most specific method for the argument tuple  $[t_1, \dots, t_m]$ , according to type store  $k$ . This function is the main subroutine in the implementation-side typechecking of a generic function (each type system will fill in the other details of implementation-side typechecking).

$$\begin{array}{c}
 \text{[form-s]} \quad \frac{p, k \vdash (c, F_1, t_1) \succ t_1' \quad \dots \quad p, k \vdash (c, F_n, t_n) \succ t_n'}{p, k \vdash (c, [F_1, \dots, F_n]) \succ [t_1', \dots, t_n']} \quad \text{where } n \geq 0, \quad (c, \mathbf{arrow}([t_1, \dots, t_n], t_0)) \in isa(k) \\
 \\
 \text{[spe-s]} \quad \frac{p, k \vdash I_2 \Rightarrow c_2 \quad k \vdash c_2 \leq_{isa} t}{p, k \vdash (c, I_1 @ I_2, t) \Rightarrow c_2} \quad \text{where } c_2 \in \mathit{cnc}(k) \cup \mathit{abs}(k) \\
 \\
 \text{[uns-s]} \quad \frac{}{p, k \vdash (c, I, t) \succ t}
 \end{array}$$

FIG. B.6. Inference rules for formal arguments.

$$\begin{aligned}
& \text{has-most-specific-method-for}(k, c, [t_1, \dots, t_m]) = \\
& m \geq 0 \wedge n \geq 1 \wedge [mh_1, \dots, mh_n] = [mh \mid (c, mh) \in \text{gfn}(k) \wedge k \vdash [t_1, \dots, t_m] \leq_{isa} * mh] \wedge \\
& \text{disjoint}(mh_1, \dots, mh_n) \wedge \exists i. (1 \leq i \leq n \wedge k \vdash mh_i \leq_{isa} * mh_1 \wedge \dots \wedge k \vdash mh_i \leq_{isa} * mh_n)
\end{aligned}$$

FIG. B.7. Checking that a tuple has a most specific applicable method.

The rules for extending the direct inheritance relation to form the descendant relation are given in Fig. B.8. These are simply the analogs of the associated rules in the dynamic semantics in Fig. A.8.

## B.2. System G

This section provides additions to the base static semantics that enforce the typing restrictions of System G. In particular, we provide typing rules for the *has-safe* “hooks” in the base static semantics of Section B.1.

The typing rule and associated function for *program-has-safe-modules* are shown in Fig. B.9. Implementation-side typechecking is globally performed on each concrete generic function in the program. In particular, for each concrete generic function accepting  $n$  arguments, all possible argument tuples  $[c_1, \dots, c_n]$  are formed such that each  $c_i$  is concrete and descends from the corresponding object in the generic function’s arrow object. It is checked that each of these argument tuples has a most specific applicable method in the generic function.

The local and regional typechecking rules for System G are shown in Fig. B.10. The judgments are all trivially satisfied. Because of the global implementation-side typechecking, there is no need for any local or regional typechecking in System G.

## B.3. System M

This section provides additions to the base static semantics that enforce the typing restrictions of System M. The rule for *program-has-safe-modules* is shown in Fig. B.11. Since System M requires no global typechecking, this rule is trivially satisfied.

The rest of the rules implement restrictions **M1–M4** as well as the two kinds of reimplementationside typechecking required for importers. The restrictions on methods and objects appear in Fig. B.12. The *module-has-safe-methods* rule implements restriction **M1**, which ensures that, for each method created in the current module, either the method was added to a local generic function or the method is first-local (the method’s first specializer is a local object). The *module-has-safe-objects* rule implements restriction **M2**, which disallows unanticipated multiple implementation inheritance across module boundaries. In particular, it is checked that if a local noninterface object  $c$  descends from two nonlocal, noninterface objects  $c_1$  and  $c_2$ , then either  $c_1$  and  $c_2$  are related or there exists a noninterface object  $c_3$  that is an ancestor of  $c$  and a descendant of both  $c_1$  and  $c_2$ .

The rule and associated functions for implementation-side typechecking of a module appear in Fig. B.13. The *M-impl-side-typecheck-gfs* function ensures that, for each concrete generic function,

$$\begin{array}{l}
\text{[isa*-s]} \quad \frac{k \vdash t_1 \leq_{isa} t_1' \dots k \vdash t_n \leq_{isa} t_n'}{k \vdash [t_1, \dots, t_n] \leq_{isa} * [t_1', \dots, t_n']} \quad \text{where } n \geq 0 \\
\text{[isb-s]} \quad k \vdash t_1 \leq_{isa} t_2 \quad \text{where } (t_1, t_2) \in \text{isa}(k) \\
\text{[isr-s]} \quad k \vdash t \leq_{isa} t \\
\text{[ist-s]} \quad \frac{k \vdash t_1 \leq_{isa} t_2 \quad k \vdash t_2 \leq_{isa} t_3}{k \vdash t_1 \leq_{isa} t_3} \\
\text{[isar-s]} \quad \frac{k \vdash t_1' \leq_{isa} t_1 \dots k \vdash t_n' \leq_{isa} t_n}{k \vdash t \leq_{isa} t'} \quad \text{where } n \geq 0 \\
k \vdash \text{arrow}([t_1, \dots, t_n], t) \leq_{isa} \text{arrow}([t_1', \dots, t_n'], t')
\end{array}$$

FIG. B.8. Inference rules for the descendant relation.

$$\begin{aligned}
 &[\text{mdsf-s}] \quad km \vdash \text{program-has-safe-modules} \quad \text{where } G\text{-impl-side-typecheck-gfs}(k, ls(k)) \\
 &G\text{-impl-side-typecheck-gfs}(k, c^*) = \\
 &\quad \forall c \in c^*. \forall [c_1, \dots, c_n]. (n \geq 0 \wedge c \in \text{cnc}(k) \wedge (c, \text{arrow}([t_1, \dots, t_n], t)) \in \text{isa}(k) \wedge \forall 1 \leq i \leq n. (c_i \in \text{cnc}(k) \wedge k \vdash c_i \leq_{\text{isa}} t_i)) \\
 &\quad \Rightarrow \text{has-most-specific-method-for}(k, c, [c_1, \dots, c_n])
 \end{aligned}$$

FIG. B.9. Global typechecking for System G.

$$\begin{aligned}
 &[\text{gfsf-s}] \quad k \vdash \text{module-has-safe-gfs} \\
 &[\text{igfsf-s}] \quad k \vdash \text{module-has-safe-imported-gfs} \\
 &[\text{obsf-s}] \quad k \vdash \text{module-has-safe-objects} \\
 &[\text{mtsf-s}] \quad k \vdash \text{module-has-safe-methods}
 \end{aligned}$$

FIG. B.10. Local and regional typechecking for System G.

$$[\text{mdsf-s}] \quad km \vdash \text{program-has-safe-modules}$$

FIG. B.11. Global typechecking for System M.

$$[\text{mtsf-s}] \quad k \vdash \text{module-has-safe-methods} \quad \text{where } c^* = \{c \mid (c, mh) \in \text{lgf}(k), \text{restriction-M1}(k, c^*)\}$$

$$\begin{aligned}
 \text{restriction-M1}(k, c^*) = \\
 \quad \forall (c, [t_1, \dots, t_n]) \in \text{lgf}(k). (c \in c^* \wedge n \geq 0) \Rightarrow (c \in \text{ls}(k) \vee (n > 0 \wedge t_1 \in \text{ls}(k)))
 \end{aligned}$$

$$[\text{obsf-s}] \quad k \vdash \text{module-has-safe-objects} \quad \text{where } \text{restriction-M2}(k)$$

$$\begin{aligned}
 \text{restriction-M2}(k) = \\
 \quad \forall c \in \text{ls}(k). \forall c_1 \in \text{cnc}(k) \cup \text{abs}(k). \forall c_2 \in \text{cnc}(k) \cup \text{abs}(k). \\
 \quad (c \in \text{cnc}(k) \cup \text{abs}(k) \wedge k \vdash c \leq_{\text{isa}} c_1 \wedge k \vdash c \leq_{\text{isa}} c_2 \wedge c_1 \notin \text{ls}(k) \wedge c_2 \notin \text{ls}(k)) \\
 \quad \Rightarrow ((k \vdash c_1 \leq_{\text{isa}} c_2 \vee k \vdash c_2 \leq_{\text{isa}} c_1) \vee \\
 \quad (c_3 \in \text{cnc}(k) \cup \text{abs}(k) \wedge c_3 \neq c \wedge k \vdash c \leq_{\text{isa}} c_3 \wedge k \vdash c_3 \leq_{\text{isa}} c_1 \wedge k \vdash c_3 \leq_{\text{isa}} c_2))
 \end{aligned}$$

FIG. B.12. Restrictions M1 and M2.

$$[\text{gfsf-s}] \quad k \vdash \text{module-has-safe-gfs} \quad \text{where } M\text{-impl-side-typecheck-gfs}(k, ls(k))$$

$$\begin{aligned}
 M\text{-impl-side-typecheck-gfs}(k, c^*) = \\
 \quad \forall c \in c^*. \forall [t_1, \dots, t_n]. (c \in \text{cnc}(k) \wedge n \geq 0 \wedge (c, \text{arrow}([t_1, \dots, t_n], t)) \in \text{isa}(k) \wedge \\
 \quad M\text{-is-legal-arg-tuple}(k, [t_1, \dots, t_n], c, \text{arrow}([t_1, \dots, t_n], t))) \\
 \quad \Rightarrow \text{has-most-specific-method-for}(k, c, [t_1, \dots, t_n])
 \end{aligned}$$

$$\begin{aligned}
 M\text{-is-legal-arg-tuple}(k, [t_1, \dots, t_n], c, \text{arrow}([t_1, \dots, t_n], t)) = \\
 \quad n \geq 0 \wedge (\forall 1 \leq i \leq n. k \vdash t_i \leq_{\text{isa}} t_i) \wedge (t_1 \in \text{cnc}(k) \vee (\text{is-non-local}(k, t_1) \wedge c \in \text{ls}(k)))
 \end{aligned}$$

FIG. B.13. Implementation-side typechecking for System M.

$$[\text{igfsf-s}] \quad k \vdash \text{module-has-safe-imported-gfs} \quad \text{where } c^* = \{c \mid c \notin \text{ls}(k) \wedge c \in \text{cnc}(k) \wedge (c, t) \in \text{isa}(k) \wedge t \in \text{ArrowType}\}, \\
 M\text{-impl-side-typecheck-imported-gfs}(k, c^*)$$

$$\begin{aligned}
 M\text{-impl-side-typecheck-imported-gfs}(k, c^*) = \\
 \quad \forall c \in c^*. \forall [t_1, \dots, t_n]. (n \geq 0 \wedge (c, t) \in \text{isa}(k) \wedge t \in \text{ArrowType} \wedge \\
 \quad (M\text{-is-local-recheck1}(k, [t_1, \dots, t_n], c, t) \vee M\text{-is-local-recheck2}(k, [t_1, \dots, t_n], c, t))) \\
 \quad \Rightarrow \text{has-most-specific-method-for}(k, c, [t_1, \dots, t_n])
 \end{aligned}$$

$$\begin{aligned}
 M\text{-is-local-recheck1}(k, [t_1, \dots, t_n], c, \text{art}) = \\
 \quad n \geq 0 \wedge M\text{-is-legal-arg-tuple}(k, [t_1, \dots, t_n], c, \text{art}) \wedge (c, mh) \in \text{lgf}(k) \wedge k \vdash [t_1, \dots, t_n] \leq_{\text{isa}^*} mh
 \end{aligned}$$

$$\begin{aligned}
 M\text{-is-local-recheck2}(k, [t_1, \dots, t_n], c, \text{art}) = \\
 \quad n \geq 0 \wedge M\text{-is-legal-arg-tuple}(k, [t_1, \dots, t_n], c, \text{art}) \wedge \text{is-orphan}(k, t_1)
 \end{aligned}$$

$$\begin{aligned}
 \text{is-orphan}(k, c) = \\
 \quad c \in \text{ls}(k) \wedge \{c\} = \{c_0 \mid c_0 \in \text{cnc}(k) \wedge k \vdash c_0 \leq_{\text{isa}} t \wedge k \vdash c \leq_{\text{isa}} c_0\} \wedge \text{is-non-local}(k, t)
 \end{aligned}$$

FIG. B.14. Reimplementation-side typechecking for System M.

$$\begin{aligned}
\text{is-non-local}(k, t) &= \text{has-no-local-in-positive-position}(k, t) \\
\text{has-no-local-in-positive-position}(k, \text{object}(i)) &= i \geq 0 \wedge \text{object}(i) \notin k(k) \\
\text{has-no-local-in-positive-position}(k, \text{arrow}([t_1, \dots, t_n], t)) &= \\
& n \geq 0 \wedge (\forall 1 \leq i \leq n. \text{has-no-local-in-negative-position}(k, t_i)) \wedge \text{has-no-local-in-positive-position}(k, t) \\
\text{has-no-local-in-negative-position}(k, \text{object}(i)) &= \text{true} \\
\text{has-no-local-in-negative-position}(k, \text{arrow}([t_1, \dots, t_n], t)) &= \\
& n \geq 0 \wedge (\forall 1 \leq i \leq n. \text{has-no-local-in-positive-position}(k, t_i)) \wedge \text{has-no-local-in-negative-position}(k, t)
\end{aligned}$$

FIG. B.15. Functions for determining if an object is nonlocal.

all of the argument tuples to be checked have a most specific applicable method. The *M-is-legal-argument-tuple* function defines which argument tuples must be checked for each generic function. The function implements restrictions **M3** and **M4**. In particular, let **arrow** ( $[t'_1 \dots, t'_n], t$ ) be the arrow object of the generic function being checked. For any argument position  $i$  other than the first, all descendants of  $t'_i$  are considered, regardless of whether these descendants are concrete. Concrete descendants of  $t'_1$  are considered, and any nonlocal descendants of  $t'_1$  are considered when the generic function being checked is local.

The rule and functions for the two kinds of reimplementation-side typechecking of imported generic functions are shown in Fig. B.14. The [igfsf-s] rule ensures that all nonlocal concrete generic functions are checked. The *M-impl-side-typecheck-imported-gfs* function ensures that, for each such generic function, all argument tuples satisfying one of the two recheck conditions has a most specific method to invoke. A tuple satisfies the first recheck condition if it is an argument tuple to be checked according to System M restrictions and the generic function has an applicable method that was created in the current module. A tuple satisfies the second recheck condition if it is an argument tuple to be checked according to System M restrictions and its first component is an orphan. An orphan is a local, concrete object that descends from a nonlocal, nonconcrete object  $t$  without also descending from a concrete descendant of  $t$ .

Finally, the functions defining when an object is nonlocal appear in Fig. B.15. A nonarrow object is nonlocal if it was not created in the current module. An arrow object is nonlocal if it does not have a local object in a positive position.

#### B.4. System E

This section provides modifications and additions to the base static semantics for System E. Section B.4.1 presents the modifications necessary to accommodate the augmented syntax of the language, described in Section 4.3. Section B.4.2 presents the formalization of System E's modular typing restrictions.

##### B.4.1. Modifications to the Base Static Semantics

Figure B.16 defines the necessary modifications and additions to the domains for the static semantics. The *TypeStore* now includes an element of the *Specializers* domain. This domain records, for each generic function, which argument positions are marked and which are unmarked. An argument position gets the *Spec s* (for *specializable*) if it is marked, and otherwise it gets the *Spec u* (for *unspecialized*).

$$\begin{aligned}
k &\in \text{TypeStore} &= \text{Isa} \times \text{Concrete} \times \text{Abstract} \times \text{GFMethodTypes} \times \text{Specializers} \times \\
&& \text{LocalTypeStore} \\
m &\in \text{Modules} &= \text{ModTypeEnv} \times \text{ModTypeStore} \times \text{Extends} \\
sp &\in \text{Specializers} &= \wp(\text{Const} \times \text{Spec}^*) \\
ext &\in \text{Extends} &= \wp(I \times I) \\
lk &\in \text{LocalTypeStore} &= \text{Locals} \times \text{Extendees} \times \text{LocalGFMethods} \\
ex &\in \text{Extendees} &= \wp(\text{Const}) \\
s &\in \text{Spec} &= \{\mathbf{s}, \mathbf{u}\}
\end{aligned}$$

FIG. B.16. Modifications and additions to the domains for the static semantics.

We define names for the components of a *TypeStore* and *Module*, and associated accessor functions with those names:  $(isa, cnc, abs, gfms, sp, (ls, ex, lgf))$  and  $(mp, mk, ext)$

$$import-store((isa, cnc, abs, gfms, sp, lk)) = (isa, cnc, abs, gfms, sp, (\{\}, \{\}, \{\}))$$

$$extend-store((isa, cnc, abs, gfms, sp, (ls, ex, lgf))) = (isa, cnc, abs, gfms, sp, (\{\}, ls \cup ex, \{\}))$$

$$s_1 - s_2 = \{x \mid x \in s_1 \wedge x \notin s_2\}$$

$$type(HO) = O$$

$$spec(O) = \mathbf{u} \quad spec(\#O) = \mathbf{s}$$

FIG. B.17. Modifications and additions to the definitions and functions.

The *Modules* domain now includes a component that records the declared extension relation between modules. This component will be used to ensure that each module has a most extending module in the program. Finally, the *LocalTypeStore* domain now includes a component that records all nonarrow objects that were declared in (transitive) extendees of the current module. This allows us to distinguish extended objects from imported objects.

Figure B.17 makes modifications and additions to the definitions and functions. First the appropriate accessor functions are defined on the modified *TypeStore* and *Module* domains. The *import-store* function takes a *TypeStore* for a module and returns the information needed by importers of the module. This information will then be added to the importer's type store. The *extend-store* function is similar, but for extenders of the module. In particular, the *extend-store* function appropriately updates the extender's list of objects created in extended modules. The  $-$  operator performs ordinary set difference. The *type* function extracts the object component of a possibly marked object. The *spec* function returns the appropriate *Spec* element, depending on whether or not the given object is marked.

Figure B.18 shows the updated rules for typechecking module lists and modules. The rule for module lists simply takes into account the fact that the *Modules* domain now has one more component than it used to have. The [mod-s] rule is revised in order to use both imported and extended modules as context in the evaluation of a module's declarations.

Figure B.19 shows the modified rule for typechecking generic functions, as well as the rule for typechecking arrow objects with possibly marked argument positions. The [gf-s] rule is similar to the original rule for typechecking generic functions, except that the new rule must record which argument positions of the generic function are marked and which are unmarked. The rule simply uses whatever markings are on the generic function's most specific arrow object for this purpose. The complexity in this augmented rule comes from the need to track the markings of each inherited arrow object. The rule for typechecking arrow objects is identical to the old rule, except that the new rule also extracts and returns the argument position markings of the arrow object.

#### B.4.2. Additions to the Base Static Semantics

Now we provide the inference rules implementing System E's modular typechecking restrictions. The global typechecking for System E is shown in Fig. B.20. The *most-extending-modules-exist* function

$$\begin{array}{c}
 \text{[md*-s]} \quad \frac{
 \begin{array}{c}
 (\{\}, \{\}, \{\}) \vdash M_1 \succ m_1 \\
 m_1 \vdash M_2 \succ m_2 \quad \dots \quad m_1 + m_2 + \dots + m_{n-1} \vdash M_n \succ m_n \\
 k, m \vdash \text{program-has-safe-modules}
 \end{array}
 }{
 \vdash M_1 \dots M_n \succ k, m
 }
 \quad
 \begin{array}{c}
 \text{where } n \geq 0, \\
 m = m_1 + m_2 + \dots + m_n \\
 \text{name}(M_1) = I_1, \dots, \text{name}(M_n) = I_n, \\
 \text{disjoint}(I_1, \dots, I_n), \\
 I_1 \in \text{domair}(mk(m)), \dots, I_n \in \text{domair}(mk(m)), \\
 k = mk(m)(I_1) + \dots + mk(m)(I_n)
 \end{array}
 \end{array}$$

$$\begin{array}{c}
 \text{[mod-s]} \quad \frac{
 \begin{array}{c}
 p, k \vdash D_1 \dots D_n \succ p_0, k_0 \\
 k + k_0 \vdash \text{module-has-safe-objects} \\
 k + k_0 \vdash \text{module-has-safe-methods} \\
 k + k_0 \vdash \text{module-has-safe-gfs} \\
 k + k_0 \vdash \text{module-has-safe-imported-gfs}
 \end{array}
 }{
 m \vdash \text{module } l \text{ imports } I_1, \dots, I_t \text{ extends } I'_1, \dots, I'_r \\
 \{D_1 \dots D_n\} \succ (\{(l, p_0)\}, \{(l, k_0)\}, \{(I, I'_1), \dots, (I, I'_r)\})
 }
 \quad
 \begin{array}{c}
 \text{where } n \geq 0, t \geq 0, r \geq 0, \\
 I_1 \in \text{domair}(mp(m)), \dots, I_t \in \text{domair}(mp(m)), \\
 I_1 \in \text{domair}(mk(m)), \dots, I_t \in \text{domair}(mk(m)), \\
 I'_1 \in \text{domair}(mp(m)), \dots, I'_r \in \text{domair}(mp(m)), \\
 I'_1 \in \text{domair}(mk(m)), \dots, I'_r \in \text{domair}(mk(m)), \\
 p = mp(m)(I_1) \& \dots \& mp(m)(I_t) \& \\
 \quad mp(m)(I'_1) \& \dots \& mp(m)(I'_r) \\
 k = \text{import-store}(mk(m)(I_1)) + \dots + \\
 \quad \text{import-store}(mk(m)(I_t)) + \\
 \quad \text{extend-store}(mk(m)(I'_1)) + \dots + \\
 \quad \text{extend-store}(mk(m)(I'_r))
 \end{array}
 \end{array}$$

FIG. B.18. Updated inference rules for module lists and modules.

$$\begin{array}{c}
\text{[gf-s]} \quad \frac{p, k \vdash I_1 : c_1 \dots p, k \vdash I_m : c_m \quad p, k \vdash O_1' : t_1 \triangleright s_1^* \dots p, k \vdash O_o' : t_o \triangleright s_o^* \quad k \vdash \text{art}_r \leq_{isa} \text{art}_1 \dots k \vdash \text{art}_r \leq_{isa} \text{art}_q}{p, k \vdash \text{interface object}; I \text{ isa } O_1, \dots, O_n \triangleright \{(I, c)\}, \text{isa} = \{(c, c_1), \dots, (c, c_m), (c, \text{art}_r)\}, k = \{c\}, sp = \{(c, s_r^*)\}} \\
\text{[art-s]} \quad \frac{p, k \vdash O_1 : t_1 \dots p, k \vdash O_n : t_n \quad p, k \vdash O : t}{p, k \vdash (S_1, \dots, S_n) \rightarrow O : \text{arrow}([t_1, \dots, t_n], t) \triangleright [s_1, \dots, s_n]}
\end{array}$$

where  $n \geq 0, m \geq 0, o \geq 0, q \geq 0,$   
 $\{I_1, \dots, I_m\} = \{O_j \mid 1 \leq j \leq n \wedge O_j \in I\},$   
 $\{O_1', \dots, O_o'\} = \{O_1, \dots, O_n\} - \{I_1, \dots, I_m\},$   
 $\{(art_1, s_1^*), \dots, (art_q, s_q^*)\} =$   
 $\{(t, s^*) \mid 1 \leq j \leq m \wedge (c_j, t) \in \text{isa}(k) \wedge t \in \text{ArrowType}$   
 $\wedge (c_j, s^*) \in \text{sp}(k)\} \cup \{(t_1, s_1^*), \dots, (t_n, s_n^*)\},$   
 $\exists r. 1 \leq r \leq q, \quad c = \text{object}(i)$

where  $n \geq 0,$   
 $O_1 = \text{type}(S_1) \dots O_n = \text{type}(S_n)$   
 $s_j = \text{spec}(S_j) \dots s_n = \text{spec}(S_n)$

FIG. B.19. Updated inference rules for generic functions and arrow objects.

checks that each module has a unique most extending module in the program. For this purpose, the  $\leq_{ext}$  relation is defined as the reflexive, transitive closure of the declared extension relation. The check for most extending modules is the only global check needed by System E.

The rest of the rules implement restrictions **E1–E5** as well as the two kinds of reimplementationside typechecking required for importers. The restrictions on methods and objects appear in Fig. B.21. The *module-has-safe-methods* rule implements restrictions **E1a** and **E1b**. The function for **E1a** ensures that methods do not specialize on unmarked positions of their associated generic function. The function for **E1b** ensures that methods added to imported generic functions are all-local multimethods. The *restriction-E1b* function also requires such methods to have at least one marked argument position. The *module-has-safe-objects* rule implements restriction **E2**. In particular, it is checked that if a local noninterface object  $c$  descends from an imported object and from two nonlocal, noninterface objects  $c_1$  and  $c_2$ , then either  $c_1$  and  $c_2$  are related or there exists a noninterface object  $c_3$  that is an ancestor of  $c$  and a descendant of both  $c_1$  and  $c_2$ .

The System E typechecking restrictions for generic functions appear in Fig. B.22. The *E-impl-side-typecheck-gfs* function ensures that, for each concrete generic function, all of the argument tuples to be checked have a most specific applicable method. The *E-is-legal-arg-tuple* function defines which argument tuples must be checked for each generic function. The function implements restrictions **E3** and **E4**. In particular, let  $\text{arrow}([t'_1, \dots, t'_n], t)$  be the arrow object of the generic function being checked. For any argument position  $i$ , all descendants of  $t'_i$  are checked, regardless of whether these descendants are concrete. However, if the generic function is singly dispatched and  $q$  is the single marked position, descendants of  $t'_q$  are checked only if they are concrete or if they are nonlocal and the generic function is local or extended. The *E-has-most-specific-method-for* function checks for the presence of a most specific applicable method for an argument tuple. However, if the conditions of restriction **E4b** hold, then the tuple is only checked for method ambiguity, not for incompleteness. Finally, the *restriction-E5* function rules out open object idioms in importers. In particular, the function checks that if any descendant of an argument object in a generic function's arrow object is imported, then the associated argument position is unmarked. Note that *restriction-E5* checks all arrow objects from which the generic function descends, which by contravariance has the effect of checking all descendants of the argument objects in the generic function's most specific arrow object.

$$\begin{array}{c}
\text{[mdsf-s]} \quad km \vdash \text{program-has-safe-modules} \quad \text{where } \text{most-extending-modules-exists}(m) \\
\text{most-extending-modules-exists}(m) = \\
\quad \forall I \in \text{domain}(mp(m)). \exists I_1 \in \text{domain}(mp(m)). \forall I_2 \in \text{domain}(mp(m)). (m \vdash I_2 \leq_{ext} I \Rightarrow m \vdash I_1 \leq_{ext} I_2) \\
\text{[exb-s]} \quad m \vdash I_1 \leq_{ext} I_2 \quad \text{where } (I_1, I_2) \in \text{ex}(m) \\
\text{[exr-s]} \quad m \vdash I \leq_{ext} I \\
\text{[ext-s]} \quad \frac{m \vdash I_1 \leq_{ext} I_2 \quad m \vdash I_2 \leq_{ext} I_3}{m \vdash I_1 \leq_{ext} I_3}
\end{array}$$

FIG. B.20. Global typechecking for System E.



[igfsf-s]  $k \vdash \text{module-has-safe-imported-gfs}$  where  $c^* = \{c \mid c \notin \text{ls}(k) \cup \text{ex}(k) \wedge c \in \text{cnd}(k) \wedge (c, t) \in \text{isa}(k) \wedge t \in \text{ArrowType}\}$ ,  
 $E\text{-impl-side-typecheck-imported-gfs}(k, c^*)$

$E\text{-impl-side-typecheck-imported-gfs}(k, c^*) =$   
 $\forall c \in c^*. \forall [t_1, \dots, t_n]. (n \geq 0 \wedge (c, t) \in \text{isa}(k) \wedge t \in \text{ArrowType} \wedge$   
 $(E\text{-is-local-recheck1}(k, [t_1, \dots, t_n], c, t) \vee E\text{-is-local-recheck2}(k, [t_1, \dots, t_n], c, t)))$   
 $\Rightarrow \text{has-most-specific-method-for}(k, c, [t_1, \dots, t_n])$

$E\text{-is-local-recheck1}(k, [t_1, \dots, t_n], c, \text{art}) =$   
 $n \geq 0 \wedge E\text{-is-legal-arg-tuple}(k, [t_1, \dots, t_n], c, \text{art}) \wedge (c, mh) \in \text{lg}(k) \wedge k \vdash [t_1, \dots, t_n] \leq_{\text{isa}} * mh$

$E\text{-is-local-recheck2}(k, [t_1, \dots, t_n], c, \text{art}) =$   
 $n \geq 0 \wedge E\text{-is-legal-arg-tuple}(k, [t_1, \dots, t_n], c, \text{art}) \wedge \text{is-singly-dispatched}(k, c)$   
 $\wedge (c, [s_1, \dots, s_n]) \in \text{sp}(k) \wedge \exists q (1 \leq q \leq n \wedge s_q = s) \wedge \text{is-orphan}(k, t_q)$

FIG. B.23. Reimplementation-side typechecking for System E.

We use the rules in System M for defining when an object is nonlocal (Fig. B.15). Figure B.24 shows the rules defining when an object is imported. An object is imported if it is nonlocal and has an imported object in a positive position.

### B.5. System ME

This section presents the typing restrictions for System ME. The full set of inference rules for System ME is the union of the rules presented in this section, the base static semantics in Section B.1, the rules for System E in Section B.4, and the rules for System M in Section B.3 that are not overridden by rules of the same name in System E.

The main typing rules for System ME are presented in Fig. B.25. The rules simply invoke the appropriate restrictions from Systems M and E. In particular, the generic functions are partitioned into those that use System M's restrictions and those that use System E's restrictions. A System M generic function obeys restrictions **M1**, **M3**, and **M4**, and a System E generic function obeys restrictions **E1**, **E3**, **E4**, and **E5**. All objects must obey both restrictions **M2** and **E2**, which greatly limit multiple implementation inheritance across module boundaries, even in extenders.

Finally, Fig. B.26 shows the functions that define which typing restrictions are used for each generic function. A generic function with no marked positions uses System M's restrictions, and a generic function with at least one marked position uses System E's restrictions.

## C. TYPE SOUNDNESS

This section sketches our proof that Dubious's static semantics is sound with respect to its dynamic semantics. Section C.1 overviews our proof method, which is based on prior work on type soundness by Wright and Felleisen 1994. Section C.2 describes the key lemma for each of Systems G, M, E, and ME.

$\text{is-imported}(k, t) = \text{is-non-local}(k, t) \wedge \text{has-imported-in-positive-position}(k, t)$

$\text{has-imported-in-positive-position}(k, \text{object}(i)) = i \geq 0 \wedge \text{object}(i) \notin \text{ls}(k) \cup \text{ex}(k)$

$\text{has-imported-in-positive-position}(k, \text{arrow}([t_1, \dots, t_n], t)) =$   
 $n \geq 0 \wedge (\exists 1 \leq i \leq n. \text{has-imported-in-negative-position}(k, t_i)) \vee \text{has-imported-in-positive-position}(k, t)$

$\text{has-imported-in-negative-position}(k, \text{object}(i)) = \text{false}$

$\text{has-imported-in-negative-position}(k, \text{arrow}([t_1, \dots, t_n], t)) =$   
 $n \geq 0 \wedge (\exists 1 \leq i \leq n. \text{has-imported-in-positive-position}(k, t_i)) \vee \text{has-imported-in-negative-position}(k, t)$

FIG. B.24. Functions for determining if an object is imported.

[mtsf-s]	$k \vdash$ <i>module-has-safe-methods</i>	where $c_1^* = \{c \mid (c, mh) \in \text{lgf}(k) \wedge \text{is-}M\text{-gfl}(k, c)\}$ , $c_2^* = \{c \mid (c, mh) \in \text{lgf}(k) \wedge \text{is-}E\text{-gfl}(k, c)\}$ , <i>restriction-M1</i> ( $k, c_1^*$ ), <i>restriction-E1a</i> ( $k, c_2^*$ ), <i>restriction-E1b</i> ( $k, c_2^*$ )
[obsf-s]	$k \vdash$ <i>module-has-safe-objects</i>	where <i>restriction-M2</i> ( $k$ ), <i>restriction-E2</i> ( $k$ )
[gfsf-s]	$k \vdash$ <i>module-has-safe-gfs</i>	where $c_1^* = \{c \mid c \in \text{ls}(k) \wedge \text{is-}M\text{-gfl}(k, c)\}$ , <i>M-impl-side-typecheck-gfs</i> ( $k, c_1^*$ ), $c_2^* = \{c \mid c \in \text{ls}(k) \cup \text{ex}(k) \wedge \text{is-}E\text{-gfl}(k, c)\}$ , <i>E-impl-side-typecheck-gfs</i> ( $k, c_2^*$ ), $c_3^* = \{c \mid c \in \text{ls}(k) \wedge \text{is-}E\text{-gfl}(k, c)\}$ , <i>restriction-E5</i> ( $k, c_3^*$ )
[igfsf-s]	$k \vdash$ <i>module-has-safe-imported-gfs</i>	where $c_1^* = \{c \mid c \notin \text{ls}(k) \wedge \text{is-}M\text{-gfl}(k, c)\}$ , $c_2^* = \{c \mid c \notin \text{ls}(k) \cup \text{ex}(k) \wedge \text{is-}E\text{-gfl}(k, c)\}$ , <i>M-impl-side-typecheck-imported-gfs</i> ( $k, c_1^*$ ), <i>E-impl-side-typecheck-imported-gfs</i> ( $k, c_2^*$ )

FIG. B.25. Inference rules for System ME.

### C.1. Proof Outline

We begin by extending the static typing rules in order to relate elements of the *Obj* domain in the dynamic semantics to their static counterparts in the *ConstType* domain:

$$\begin{array}{l}
 \text{[val-s]} \quad p, k \vdash \mathbf{obj}(i) : \mathbf{object}(i) \\
 \text{[valart-s]} \quad \frac{p, k \vdash o_1 : t_1 \dots p, k \vdash o_n : t_n \quad p, k \vdash o : t}{p, k \vdash \mathbf{arrow}([o_1, \dots, o_n], o) : \mathbf{arrow}([t_1, \dots, t_n], t)} \quad \text{where } n \geq 0
 \end{array}$$

We consider elements of the *Obj* domain to be members of the *E* syntactic domain, thereby allowing the use of the subsumption rules [smpc-s] and [smpu-s] to raise the type of such objects. Since none of the rules for typechecking elements of the *Obj* domain make use of the given *TypeEnv*  $p$ , we often omit this context to the judgment.

Now we define two correspondences that relate static and dynamic environments and stores:

DEFINITION 1. Given a *TypeEnv*  $p$ , *TypeStore*  $k$ , and *Env*  $e$ , we say that  $p \models_k e$  if  $\text{domain}(p) = \text{domain}(e)$  and for every pair  $(I, ot) \in p$ , there exists a pair  $(I, v) \in e$  such that  $p, k \vdash v : ot$ .

DEFINITION 2. Given a *TypeStore*  $k$  and *Store*  $s$ , we say that  $k \approx s$  if the following two conditions hold:

1.  $|\text{isa}(k)| = |\text{isa}(s)|$  and for every pair  $(t_1, t_2) \in \text{isa}(k)$ , there exists a pair  $(o_1, o_2) \in \text{isa}(s)$  such that  $k \vdash o_1 : t_1$  and  $k \vdash o_2 : t_2$ .
2.  $|\text{gfms}(k)| = |\text{gfms}(s)|$  and for every pair  $(t_0, [t_1, \dots, t_n]) \in \text{gfms}(k)$ , where  $n \geq 0$ , there exists a pair  $((o_0, [o_1, \dots, o_n]), mb) \in \text{gfms}(s)$  such that  $\forall 0 \leq i \leq n. k \vdash o_i : t_i$ .

Next we prove a subject reduction lemma:

LEMMA 1 (Subject reduction). Suppose we are given a module list  $M_1, \dots, M_n$ , *TypeStore*  $k$ , *Modules*  $m$ , *Store*  $s$ , and *ModEnv*  $me$  such that  $\vdash M_1, \dots, M_n \triangleright_k, m$  and  $\vdash M_1, \dots, M_n \triangleright_s, me$ . Let  $I$  be an identifier such that  $I \in \text{domain}(mp(m))$  and  $I \in \text{domain}(me)$ , and let  $p = mp(m)(I)$  and  $e = me(I)$ . Given an expression  $E$ , if  $p, k \vdash E : ot$  and  $e, s \vdash E \Rightarrow o$ , then  $p, k \vdash o : ot$ .

$$\begin{array}{l}
 \text{is-}M\text{-gfl}(k, c) = n \geq 0 \wedge c[s_1, \dots, s_n] \in \mathcal{F}(k) \wedge \forall I \leq i \leq n. (s_i = \mathbf{u}) \\
 \text{is-}E\text{-gfl}(k, c) = n \geq 0 \wedge c[s_1, \dots, s_n] \in \mathcal{F}(k) \wedge \exists I \leq i \leq n. (s_i = \mathbf{s})
 \end{array}$$

FIG. B.26. Functions for partitioning generic functions.

*Proof.* First we prove that  $p \models_k e$  and  $k \approx s$ . Given these correspondences, subject reduction is then proven by induction on the length of the derivation in the dynamic semantics that  $e, s \vdash E \Rightarrow o$ . ■

The previous lemma says that types are preserved throughout the evaluation of an expression, in the context of well-typed modules. To complete the proof of type soundness, we need to show that the evaluation of well-typed expressions does not get stuck, in the context of well-typed modules. We start with a notion of *faulty* expressions, which the following definition formalizes:

**DEFINITION 3 (Faulty expressions).** An expression  $E$  is faulty with respect to environment  $e$  and store  $s$  if one of the following conditions holds:

1.  $E$  is an identifier and  $E \notin \text{domain}(e)$
2.  $E = E_0(E_1, \dots, E_n)$  and  $\exists i. (0 \leq i \leq n \text{ and } E_i \text{ is faulty with respect to } e \text{ and } s)$
3.  $E = E_0(E_1, \dots, E_n)$ ,  $\forall i. (0 \leq i \leq n \Rightarrow e, s \vdash E_i \Rightarrow v_i)$ , and there is no most specific applicable method for  $v_0(v_1, \dots, v_n)$  in  $s$ .
4.  $E = E_0(E_1, \dots, E_n)$ ,  $\forall i. (0 \leq i \leq n \Rightarrow e, s \vdash E_i \Rightarrow v_i)$ ,  $((v_0, mh), ([I_1, \dots, I_n], E', e')) \in \text{gfps}(s)$  is the most-specific applicable method for  $v_0(v_1, \dots, v_n)$  in  $s$ , and  $E'$  is faulty with respect to  $e' \ \& \ \{(I_1, v_1), \dots, (I_n, v_n)\}$  and  $s$ .

This definition of faulty expression is validated by the following lemma, which says that the faulty expressions are a conservative approximation of the stuck expressions:

**LEMMA 2 (Every stuck expression is faulty).** *Suppose we are given a module list  $M_1, \dots, M_n$ , Store  $s$ , and  $\text{ModEnv } me$  such that  $\vdash M_1, \dots, M_n \succ s, me$ . Let  $I$  be an identifier such that  $I \in \text{domain}(me)$ , and let  $e = me(I)$ . Given an expression  $E$ , if  $E$  is not faulty with respect to  $e$  and  $s$  and  $e, s \vdash E$  does not diverge, then there exists a  $\text{Val } v$  such that  $e, s \vdash E \Rightarrow v$ .*

*Proof.* By induction on the length of the derivation in the dynamic semantics of  $e, s \vdash E$ . ■

The final lemma shows that well-typed expressions are not faulty, in the context of well-typed modules.

**LEMMA 3 (Well-typed expressions are not faulty).** *Suppose we are given a module list  $M_1, \dots, M_n$ ,  $\text{TypeStore } k$ ,  $\text{Modules } m$ ,  $\text{Store } s$ , and  $\text{ModEnv } me$  such that  $\vdash M_1, \dots, M_n \succ k, m$ , and  $\vdash M_1, \dots, M_n \succ s, me$ . Let  $p$  be a  $\text{TypeEnv}$  and  $e$  be an  $\text{Env}$  such that  $p \models_k e$ . Given an expression  $E$ , if there exists an  $\text{ObjType } ot$  such that  $p, k \vdash E : ot$ , then  $E$  is not faulty with respect to  $e$  and  $s$ .*

*Proof.* We prove that none of the four cases in the definition of faulty expression holds. First suppose  $E$  is some identifier. Then the derivation of  $p, k \vdash E : ot$  ensures that  $E \in \text{domain}(p)$ . Since  $p \models_k e$ , by Definition 1 we know that  $E \in \text{domain}(e)$  as well, thereby ruling out case one in the definition of faulty expressions. Now suppose  $E = E_0(E_1, \dots, E_n)$ . We rule out case two by induction. Ruling out case three is the only part of the entire soundness proof that depends on which type system ( $G, M, E$ , or  $ME$ ) is used. For each of these systems, we prove that every legal argument tuple to each generic function in the program has a most specific applicable method in  $k$ . This is the key lemma of the soundness proof, and Section C.2 sketches it for each of the four type systems. Ruling out case three is then completed by showing that  $k \approx s$ , so by Definition 2 we can show that method specificity in  $k$  is isomorphic to method specificity in  $s$ . In particular, if a message send has a most specific applicable method in  $k$ , the message send also has a most specific applicable method in  $s$ . Finally, we rule out case four. Let  $E'$  be the method body that is invoked upon the message send  $E_0(E_1, \dots, E_n)$ . Let  $p'$  be the  $\text{TypeEnv}$  used to typecheck  $E'$  during the typechecking of modules  $M_1, \dots, M_n$ , and let  $e'$  be the  $\text{Env}$  used in evaluating  $E'$  upon the message send  $E_0(E_1, \dots, E_n)$ . We show that  $p' \models_k e'$  and that there exists an  $\text{ObjType } ot'$  such that  $p', k \vdash E' : ot'$ , so this case can be ruled out by induction. ■

Finally, all three lemmas are combined to prove the main result:

**THEOREM 1 (Type soundness).** *Given a program  $P = M_1, \dots, M_n \text{ import } I \text{ in } E \text{ end}$ , suppose there exists an  $\text{ObjType } ot$ ,  $\text{TypeStore } k$ , and  $\text{Modules } m$  such that  $\vdash P : ot \succ k, m$ , and  $\vdash P$  does not diverge in the dynamic semantics. Then  $I \in \text{domain}(mp(m))$  and there exist an  $\text{Obj } o$ ,  $\text{Store } s$ , and  $\text{ModEnv } me$  such that  $\vdash P \Rightarrow o \succ s, me$  and  $p, k \vdash o : ot$ , where  $p = mp(m)(I)$ .*

*Proof.* Since  $\vdash P : ot \succ k, m$ , by the [prg-s] rule in the static semantics we have  $\vdash M_1, \dots, M_n \succ k, m, I \in \text{domain}(mp(m))$ , and  $p, k \vdash E : ot$ . First we show that, since no message sends are evaluated during the evaluation of modules, successful typechecking of modules implies successful evaluation of modules. In particular,  $\vdash M_1, \dots, M_n \succ k, m$  and  $I \in \text{domain}(mp(m))$  imply that  $\vdash M_1, \dots, M_n \succ s, me$  in the dynamic semantics, for some *Store*  $s$  and *ModEnv*  $me$  such that  $I \in \text{domain}(me)$  and  $p \models_k e$ , where  $e = me(I)$ . Then by Lemma 3,  $E$  is not faulty with respect to  $e$  and  $s$ . Since  $\vdash P$  does not diverge in the dynamic semantics, by the [prg-d] rule neither does  $e, s \vdash E \Rightarrow$ . Therefore by Lemma 2,  $e, s \vdash E \Rightarrow o$ , so by the [prg-d] rule again,  $\vdash P \Rightarrow o \succ s, me$ . Finally by Lemma 1,  $p, k \vdash o : ot$ . ■

## C.2. Key Lemma

This Section provides details on the key lemma in the soundness proof for each of our type systems. The lemma says that, after all modules have been typechecked, for each concrete generic function  $f$  in the program, every legal argument tuple to  $f$  has a most specific method implementation to invoke. Formally, the lemma is defined as follows:

LEMMA 4 (Key lemma). *Suppose we are given a module list  $M_1, \dots, M_n$ , TypeStore  $k$ , and Modules  $m$  such that  $\vdash M_1, \dots, M_n \succ k, m$ . Further suppose we are given Const objects  $c, c_1, \dots, c_n$ , where  $n \geq 0$ . If there exists a pair  $(c, \mathbf{arrow}([t_1, \dots, t_n], t))$  in  $\text{isa}(k)$  and for  $1 \leq i \leq n, c_i \in \text{cnc}(k)$  and  $k \vdash c_i \leq_{\text{isa}} t_i$ , then has-most-specific-method-for  $(k, c, [c_1, \dots, c_n])$ .*

We now sketch the proof of this lemma for each of our type systems.

### C.2.1. System $G$

Since  $\vdash M_1, \dots, M_n \succ k, m$ , by rule [md\*-s] we know that  $k, m \vdash \text{program-has-safe-modules}$  holds. Therefore, by rule [mdsf-s] in System  $G$ , global implementation-side typechecks on the program succeed, which is precisely the requirement of this lemma. In particular, by the  $G\text{-impl-side-typecheck-gfs}$  function we know that for each concrete generic function in the program, each legal argument tuple has a most specific method to invoke.

### C.2.2. System $M$

We say that an object (method) is *visible* in a module  $M$  if the object (method) was created in a (reflexive, transitive) importee of  $M$ . Let  $c$  be the generic function object in the statement of the lemma, and suppose it was created in module  $M$ . We divide the proof into several cases:

1. The generic function  $c$  accepts no arguments (that is,  $n = 0$ ). Then implementation-side typechecks in module  $M$  (rule [gfsf-s]) ensure the existence of a most specific method implementation visible in  $M$ . This case is finished by showing that all applicable methods to the argument tuple  $[\ ]$  must be visible in  $M$ , so module  $M$ 's typechecks are sufficient to ensure global safety of  $c$  for  $[\ ]$ . In particular, all methods added to  $c$  outside of module  $M$  must be first-local methods, so by restriction  $MI$  these methods must have at least one argument. Therefore, these methods are not applicable to the argument tuple  $[\ ]$ .

2. The generic function  $c$  accepts at least one argument (that is,  $n > 0$ ). There are several cases:
  - a. The first argument object,  $c_1$ , is visible in  $M$ . First we show that all methods of  $c$  applicable to  $[c_1, \dots, c_n]$  must be visible in  $M$ . In particular, any method of  $c$  that is not visible in  $M$  must be created in some importer  $M_m$  of  $M$ . Then by restriction  $MI$ , the method must be first-local. Therefore, the method's first argument specializer, call it  $c_m$ , is created in  $M_m$  so  $c_m$  is not visible in  $M$ . If the method is applicable to  $[c_1, \dots, c_n]$ , then  $c_1$  must descend from  $c_m$ . But since  $c_1$  is visible in  $M$ , this means that  $c_m$  must also be visible in  $M$ , so we have a contradiction.

Therefore, we only need to consider methods of  $c$  that are visible in module  $M$ . Our strategy is to build an argument tuple  $[t'_1, \dots, t'_n]$  that is considered during implementation-side typechecks of  $c$  in module,  $M$  and then show that the methods of  $c$  that are applicable to  $[c_1, \dots, c_n]$  are precisely the methods of  $c$  that are applicable to  $[t'_1, \dots, t'_n]$ . If we show this, then this case is proven, because implementation-side typechecks in module  $M$  ensure that  $[t'_1, \dots, t'_n]$  has a most specific applicable method, which implies that  $[c_1, \dots, c_n]$  has a most specific applicable method as well.

if  $c_i$  has no non-interface ancestors (including itself) that descend  $t_i$  and are visible in  $M$ , then  $t'_i = t_i$  else  $t'_i = c'_i$ , where  $c'_i$  is the most-specific non-interface ancestor of  $c_i$  that descends  $t_i$  and is visible in  $M$

FIG. C.1. Constructing the argument tuple  $[t'_1, \dots, t'_n]$ .

The tuple  $[t'_1, \dots, t'_n]$  is built by the construction in Fig. C.1, given  $[c_1, \dots, c_n]$ , **arrow**  $([t_1, \dots, t_n], t)$ , and  $M$ . The construction in Fig. C.1 assumes that if there exists an ancestor of  $c_i$  that descends  $t_i$  and is visible in  $M$ , then there is a most specific such ancestor. We show that this is in fact the case, by the limitations on nonlocal multiple inheritance imposed by restriction **M2**. It is straightforward to show that the tuple  $[t'_1, \dots, t'_n]$  satisfies the requirements of the previous paragraph.

b. The first argument object,  $c_1$ , is not visible in  $M$ ; further,  $c_1$  was created in some module  $M_1$ , and  $c$  is not visible in  $M_1$ . This is the case when  $c_1$  is an open object. We prove this case precisely as in case 2a above. First, we can show that no methods created in importers of  $M$  are applicable to  $[c_1, \dots, c_n]$  or else we would be able to show that  $c$  is visible in  $M_1$ , contradicting the assumptions of this case. Then we build  $[t'_1, \dots, t'_n]$  as in Fig. C.1, given  $[c_1, \dots, c_n]$ , **arrow**  $([t_1, \dots, t_n], t)$ , and  $M$ . Finally, we show that  $[t'_1, \dots, t'_n]$  is considered during implementation-side typechecking of  $c$  in  $M$  and that the methods of  $c$  that are applicable to  $[t'_1, \dots, t'_n]$  are precisely the methods of  $c$  that are applicable to  $[c_1, \dots, c_n]$ .

c. The first argument object,  $c_1$ , is not visible in  $M$ ; further,  $c_1$  was created in some module  $M_1$ ,  $c$  is visible in  $M_1$ , and at least one of the two conditions for reimplementation-side typechecking of  $c$  is satisfied in  $M_1$ . That is, either  $M_1$  adds a method to  $c$  that is applicable to  $[c_1, \dots, c_n]$  or  $c_1$  is an orphan. The proof of this case is similar to the proofs of cases 2a and 2b, but with respect to module  $M_1$  instead of  $M$ . In particular, we first prove that only methods of  $c$  that are visible in  $M_1$  are applicable to  $[c_1, \dots, c_n]$ . Then we build  $[t'_1, \dots, t'_n]$  as in Fig. C.1, given  $[c_1, \dots, c_n]$ , **arrow**  $([t_1, \dots, t_n], t)$ , and  $M_1$ . We show that  $[t'_1, \dots, t'_n]$  is considered by the rechecks of  $c$  performed in module  $M_1$  and that the methods of  $c$  that are applicable to  $[t'_1, \dots, t'_n]$  are precisely the methods of  $c$  that are applicable to  $[c_1, \dots, c_n]$ .

d. The first argument object,  $c_1$ , is not visible in  $M$ ; further,  $c_1$  was created in some module  $M_1$ ,  $c$  is visible in  $M_1$ , and neither of the two conditions for reimplementation-side typechecking of  $c$  is satisfied in module  $M_1$ . The key to proving this case is induction on the number of concrete ancestors of  $c_1$  that descend from  $t_1$ . The base case, when  $c_1$  has one such ancestor (itself), is covered by cases 2a–2c above. Since neither recheck condition is met, we know that  $c_1$  is not an orphan, so it must have at least one concrete ancestor other than itself that descends from  $t_1$ . We prove this case by finding one such ancestor  $c'_1$  such that the methods of  $c$  that are applicable to  $[c_1, \dots, c_n]$  are precisely the methods of  $c$  that are applicable to  $[c'_1, c_2, \dots, c_n]$ . By the inductive hypothesis,  $[c'_1, \dots, c_n]$  has a most specific applicable method in  $c$ , which means that  $[c_1, \dots, c_n]$  does as well.

### C.2.3. System E

In System E, we say that an object (method) is visible in module  $M$  if the object (method) was created in a (reflexive, transitive) importee or extender of  $M$ . Let  $c$  be the generic function object in the statement of the Lemma. Since  $\vdash M_1, \dots, M_n \succ k, m$ , we know that  $k, m \vdash \text{program-has-safe-modules}$  holds, so by [mdsf-s] in System E, we know that each module has a most extending module. Let  $M$  be the most extending module of  $M_c$ , the module that created  $c$ .

In this sketch, we assume that all argument positions are marked. It is easy to extend the proof to include unmarked argument positions. In particular, because of restriction **E1a**, unmarked positions may not be specialized upon. Therefore, the unmarked argument position of methods cannot be the causes of method ambiguities. We can show that if an argument tuple has a most specific applicable method implementation when we ignore the unmarked positions in the argument tuple and in all methods of the generic function, the tuple must also have a most specific applicable method implementation when the unmarked positions are taken into consideration.

There are several cases:

1.  $c$  has zero arguments. Implementation-side typechecks in module  $M$  (rule [gfsf-s]) ensure the existence of a most specific method for the tuple  $[\ ]$ . Since  $M$  is the most extending module of  $M_c$  any

method not visible in  $M$  must have been added by an importer of  $M_c$ . Therefore, by the restriction **E1b** (as implemented by the *restriction-E1b* function), the generic function must have at least one marked position, so the method is not applicable to  $[\ ]$ . Therefore, only methods visible in  $M$  can be applicable to  $[\ ]$ , so the checks by module  $M$  are enough to ensure global safety of  $c$  for  $[\ ]$ .

2.  $c$  has one argument. There are several cases:

a.  $c_1$  is visible by  $M$ . Then implementation-side typechecks in module  $M$  ensure the existence of a most specific method for the tuple  $[c_1]$ . Suppose some method not visible in  $M$  is applicable to  $[c_1]$ . Since the method is not visible in  $M$ , it was added in some importer  $M_m$  of  $M_c$ . Therefore, by restriction **E1b**, the method must be an all-local method, so its specializer, call it  $c'_1$ , is local to  $M_m$  and is therefore not visible in  $M$ . Since the method is applicable to  $[c_1]$ , we know that  $c_1$  descends  $c'_1$ . Since  $c_1$  is visible in  $M$  and  $c_1$  descends  $c'_1$ , we can show that  $c'_1$  must also be visible in  $M$ , and we have a contradiction. Therefore, only methods visible in  $M$  can be applicable to  $[c_1]$ , so  $M$ 's checks are enough to ensure global safety of  $c$  for  $[c_1]$ .

b.  $c_1$  is not visible by  $M$ ,  $c_1$  was created in module  $M_1$ , and  $c$  is not visible in  $M_1$ . This is the case when  $c_1$  is an open object. First, we can show that no methods created in importers of  $M_c$  are applicable to  $[c_1]$ , or else we would be able to show that  $c$  is visible in  $M_1$ , contradicting the assumptions of this case. Finally, we find an object  $t'_1$  visible in  $M$  such that the methods of  $c$  that are applicable to  $[c_1]$  are precisely the methods of  $c$  that are applicable to  $[t'_1]$ . We use the definition of  $[t'_1]$  in Fig. C.1, given  $[c_1]$ , **arrow**  $([t_1], t)$ , and  $M$ . By a combination of the restrictions on open objects of **E5** and the restrictions on multiple inheritance of **E2**, we can show that this construction of  $t'_1$  is well founded. That is, if there exists an ancestor of  $c_1$  that descends  $t_1$  and is visible in  $M$ , then there is a most specific such ancestor.

Since  $c_1$  is an open object, we can show that  $t'_1$  must be nonlocal to module  $M$ , so by restriction **E4** we know that  $[t'_1]$  is considered in implementation-side checks of  $c$  in  $M$ , even if  $t'_1$  is nonconcrete. Therefore,  $[t'_1]$  has a most specific applicable method, so  $[c_1]$  does as well. One caveat is that, if  $c$  is extended by module  $M$ , rather than local to  $M$ , by restriction **E4b** we know that  $[t'_1]$  may be checked only for ambiguity and not for incompleteness. However, we can show that there must exist at least one visible method that is applicable to  $[t'_1]$ , so there cannot be an incompleteness problem.

c.  $c_1$  is not visible in  $M$ ,  $c_1$  was created in module  $M_1$ ,  $c$  is visible in  $M_1$ , and  $M_1$  adds a method to  $c$  that is applicable to  $[c_1]$ . Therefore, the first condition for reimplementation-side typechecking  $c$  in  $M_1$  is satisfied, so rechecks ensure that there is a most specific method implementation for  $[c_1]$  of the methods visible in  $M_1$ . This case is finished by proving that the methods created in module  $M_1$  that are applicable to  $[c_1]$  (we know there is at least one such method by the assumptions of this case) are strictly more specific than any other applicable method in the entire program. Therefore, the rechecks from module  $M_1$  are enough to ensure global safety of  $c$  for  $[c_1]$ .

d.  $c_1$  is not visible in  $M$ ,  $c_1$  was created in module  $M_1$ ,  $c$  is visible in  $M_1$ , and  $c_1$  is an orphan. Therefore, the second condition for reimplementation-side typechecking  $c$  in  $M_1$  is met, so rechecks ensure that there is a most specific method implementation for  $[c_1]$  of the methods visible in  $M_1$ . Interestingly, there may be more specific method implementations that are not visible in  $M_1$ . However, because we can show that  $M_1$  must import the module creating  $t_1$ ,  $c_1$  is subject to the multiple inheritance restrictions of **E2**. Therefore, we can show that even if there are methods applicable to  $[c_1]$  that are more specific than the most specific method visible in  $M_1$ , there will still be a most specific such method.

e.  $c_1$  is not visible by  $M$ ,  $c_1$  was created in module  $M_1$ ,  $c$  is visible in  $M_1$ , and neither of the conditions for reimplementation-side typechecking  $c$  in  $M_1$  holds. The key to proving this case is induction on the number of concrete ancestors of  $c_1$  that descend from  $t_1$ . The base case, when  $c_1$  has one such ancestor (itself), is covered by cases 2a–2d above. Since neither recheck condition is met, we know that  $c_1$  is not an orphan, so it must have at least one concrete ancestor other than itself that descends from  $t_1$ . We prove this case by finding such an ancestor  $c'_1$  such that the methods of  $c$  that are applicable to  $[c_1]$  are precisely the methods of  $c$  that are applicable to  $[c'_1]$ . By the inductive hypothesis,  $[c'_1]$  has a most specific applicable method in  $c$ , which means that  $[c_1]$  does as well.

3.  $c$  has at least two arguments. In this case, we actually prove a slightly stronger lemma, replacing  $c_i \in cnc(k)$  with  $c_i \in cnc(k) \cup abs(k)$  in the statement of Lemma 4. That is, argument tuples that contain

abstract objects must now also be shown to have a most specific applicable method. By restriction **E3**, abstract objects are always considered concrete when implementation-side typechecking a generic function with more than one marked argument position, so the modified lemma is not much harder to prove than the original one. However, this allows us to use a stronger inductive hypothesis in case 3e below.

a. Each argument is visible in  $M$ . This is just the generalization of case 2a above to multiple arguments, and the same proof technique applies here.

b. At least one argument,  $c_i$ , is visible in  $M$ , but some other argument,  $c_j$ , is not visible in  $M$ . Since  $c_i$  is visible in  $M$ , we can show that only methods visible in  $M$  can be applicable to  $[c_1, \dots, c_n]$ . Then we build the tuple  $[t'_1, \dots, t'_n]$  as in Fig. C.1, given  $[c_1, \dots, c_n]$ , **arrow**  $([t_1, \dots, t_n]t)$ , and  $M$ . We show that  $[t'_1, \dots, t'_n]$  is considered by the rechecks of  $c$  performed in module  $M$  and that the methods of  $c$  that are applicable to  $[t'_1, \dots, t'_n]$  are precisely the methods of  $c$  that are applicable to  $[c_1, \dots, c_n]$ .

c. No argument is visible in  $M$ ; further, all arguments were created in the same module,  $M_1$ , and  $M_1$  adds a method to  $c$  that is applicable to  $[c_1, \dots, c_n]$ . This is just the generalization of case 2c above to multiple arguments, and the same proof technique applies here.

d. No argument is visible in  $M$ ; further, at least one of the conditions on case 3c above does not hold, and each  $c_i$  has zero non-local non-interface ancestors that descend from  $t_i$ . We show that the methods applicable to  $[c_1, \dots, c_n]$  are precisely the methods applicable to  $[t_1, \dots, t_n]$ . Since each  $c_i$  descends  $t_i$ , it is clear that every method applicable to  $[t_1, \dots, t_n]$  is applicable to  $[c_1, \dots, c_n]$ . Suppose there were some method  $m$  applicable to  $[c_1, \dots, c_n]$  but not to  $[t_1, \dots, t_n]$ . We show that this implies that each  $c_i$  was created in the same module, call it  $M_1$ , and that method  $m$  was created in  $M_1$  as well. Therefore, all the conditions on case 3c above hold, contradicting our initial assumption.

So the methods applicable to  $[c_1, \dots, c_n]$  are precisely the methods applicable to  $[t_1, \dots, t_n]$ . Therefore, we can show that all applicable methods are visible in  $M$ . By restriction **E3b**, we know that  $[t_1, \dots, t_n]$  is considered during implementation-side typechecks of  $c$  in  $M$ , even if some  $t_i$  is non-concrete. Therefore, we know that  $[t_1, \dots, t_n]$  has a most specific method in  $c$ , and hence  $[c_1, \dots, c_n]$  does as well.

e. No argument is visible in  $M$ ; further, at least one of the conditions on each of cases 3c and 3d above does not hold. As in the final case of the previous proofs, we prove this by induction. However, the induction in this case is more complicated than in the previous cases. In particular, induction is performed on the number of ancestor tuples  $[c'_1, \dots, c'_n]$  of  $[c_1, \dots, c_n]$  according to the  $\leq_{isa^*}$  relation, such that each  $c'_i$  is a noninterface object that descends from  $t_i$ . The base case is covered by cases 3a–3d above. By induction, each of these ancestor tuples has a most specific applicable method in  $c$ . This case is proven by showing that the most specific applicable method of one of these ancestor tuples must also be the most specific applicable method of  $[c_1, \dots, c_n]$ .

#### C.2.4. System ME

The proof for System ME follows immediately from the proofs for Systems M and E. In particular, System ME partitions its generic functions into those that use System M's rules and those that use System E's rules. Suppose the generic function  $c$  in the lemma uses System M's rules. By the rules for System ME, this generic function obeys restrictions **M1**, **M3**, and **M4**, and the entire program obeys restriction **M2**. Therefore, our proof of Lemma 4 for System M implies that every legal argument tuple to  $c$  has a most specific applicable method. A similar argument is used if  $c$  uses System E's rules.

## 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 Lyle Ramshaw for suggesting the term “first-local.” 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 (CCR-9970986), an NSF Young Investigator Award (CCR-9457767), and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software.

## REFERENCES

- Abadi, M., and Cardelli, L. (1995), An Imperative Object Calculus, *Theory and Practice of Object Systems* 1(3), 151–166.
- Abadi, M., and Cardelli, L. (1996), *A Theory of Objects*. Springer-Verlag, New York.
- Agrawal, R., DeMichiel, L. G., and Lindsay, B. G. (1991), Static Type Checking of Multi-Methods. *OOPSLA'91 Conference Proceedings*, Phoenix, AZ, volume 26, number 11 of *ACM SIGPLAN Notices*, pp. 113–128. ACM, New York.
- Arnold, K., and Gosling, J. (1998), *The Java Programming Language Second Edition*, Addison-Wesley, Reading, Mass.
- Baumgartner, G., Läufer, K., and Russo, V. F. (1996), On the Interaction of Object-Oriented Design Patterns and Programming Languages. Technical Report CSD-TR-96-020, Department of Computer Science, Purdue University.
- Bourdoncle, F., and Merz, S. (1997), 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.
- Boyland, J., and Castagna, G. (1997), 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.
- Bracha, G., and Cook, W. (1990), 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*.
- Bruce, K., Cardelli, L. (1995), 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.
- Bruce, K., Petersen, L., and Fiech, A. (1997), Subtyping is not a good “Match” for object-oriented languages. In proceedings of the *Eleventh European Conference on Object-Oriented Programming* Finland. Springer-Verlag LNCS 1241.
- Canning, P., Cook, W., Hill, W., Olthoff, W., and Mitchell, J. C. (1989), F-Bounded Polymorphism for Object-Oriented Programming. *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280.
- Cardelli, L. (1984), A Semantics of Multiple Inheritance. *Information and Computation* 76(2/3), 138–164, February-March, 1988. An earlier version appeared in *Semantics of Data Types Symposium*, LNCS 173, pp. 51–66, Springer-Verlag.
- Castagna, G., Ghelli, G., and Longo, G. (1992), 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.
- Castagna, G. (1995), Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems* 17(3), 431–447.
- Castagna, G. (1997), *Object-Oriented Programming A Unified Foundation*, Birkhäuser, Boston.
- Chambers, C. (1992), 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.
- Chambers, C. (1995), The Cecil Language: Specification and Rationale: Version 2.0. Department of Computer Science and Engineering, University of Washington, December. <http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html>.
- Chambers, C., and Leavens, G. T. (1995), Typechecking and Modules for Multi-Methods *ACM Transactions on Programming Languages and Systems* 17(6), 805–843.
- Chambers, C., and Leavens, G. T. (1997), BeCecil A Core Object-Oriented Language with Block Structure and Multimethods: *The Fourth International Workshop on the Foundations of Object-oriented Languages*, Paris, France.
- Chambers, C. (1998), Towards Diesel, a Next-Generation OO Language after Cecil. Invited talk, *The Fifth Workshop on Foundations of Object-oriented Languages*, San Diego, California.
- Cook, W. (1991), 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.
- Ernst, M. D., Kaplan, C., and Chambers, C. (1998), Predicate Dispatching: A Unified Theory of Dispatch. *Twelfth European Conference on Object-Oriented Programming*, Brussels, Belgium, pp. 186–211.
- Feinberg, N., Keene, S. E., Mathews, R. O., and Withington, P. T. (1997), *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass.
- Findler, R. B., and Flatt, M. (1998), Modular Object-Oriented Programming with Units and Mixins. *International Conference on Functional Programming*, Baltimore, Maryland.
- Flatt, M., Krishnamurthi, S., and Felleisen, M. (1998), 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.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass.
- Gosling, J., Joy, B., Steele, G. (1996), *The Java Language Specification*, Addison-Wesley, Reading, Mass.
- Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997), Aspect-Oriented Programming. In proceedings of the *Eleventh European Conference on Object-Oriented Programming*, Finland. Springer-Verlag LNCS 1241.
- LaLonde, W. R., Thomas, D. A., and Pugh, J. R. (1986), An Exemplar Based Smalltalk. *OOPSLA '86 Conference Proceedings*, pp. 322–330, Portland, OR, September, 1986. Published as *SIGPLAN Notices* 21(11).
- Leavens, G. T., and Millstein, T. D. (1998), Multiple Dispatch as Dispatch on Tuples. *Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia.

- Lieberman, H. (1986), 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).
- Mugridge, W. B., Hamer, J., and Hosking, J. G. (1991), 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.
- Odersky, M., and Wadler, P. (1997), 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.
- Paepcke, A. (1993), *Object-Oriented Programming: The CLOS Perspective* MIT Press.
- Reenskaug, T., Anderson, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nordhagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A., and Stenslet, P. (1992), OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming* 5(6), pp. 27–41.
- Reynolds, J. C. (1980), 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.
- Shalit, A. (1997), *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass.
- Smaragdakis, Y., and Batory, D. (1998), Implementing Layered Designs with Mixin Layers. *Twelfth European Conference on Object-Oriented Programming* Brussels, Belgium, pp. 550–570.
- Steele, G. L., Jr. (1990), *Common Lisp: The Language (second edition)*. Digital Press, Bedford, MA.
- Ungar, D., and Smith, R. B. (1987), 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.
- VanHilst, M., and Notkin, D. (1996), Using C++ Templates to Implement Role-Based Designs. *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, pp. 22–37.
- Wright, A. K., and Felleisen, M. (1994), A Syntactic Approach to Type Soundness. *Information and Computation* 115(1), 38–94.