Modular Statically Typed Multimethods

Todd Millstein and Craig Chambers

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

Abstract

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

1 Introduction

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

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

We have designed Dubious, a simple core language supporting both symmetric multimethods and separately typechecked modules. We have identified two sets of restrictions on module importers to achieve modular type safety. These restrictions represent the endpoints of a spectrum trading off flexibility and expressiveness for modularity. We also identify an intermediate point on this spectrum that provides programmers with fine-grained control over this tradeoff.

```
P  ::=  M1 ... Mn import I in E end
M  ::=  module I imports I1,...,Im { D1 ... Dn }
D  ::=  [abstract] object I [R] isa I1,...,In
     |  I has method(F1,...,Fn) { E }
E  ::=  E0(E1,...,En)  |  I
R  ::=  (T1,...,Tn)→T
T  ::=  I  |  R
F  ::=  I1 [@ I2]
I  ::=  identifier
```

Figure 1: Syntax of Dubious

Section 2 describes the language's features, along with a simple global typechecking algorithm. Section 2 also presents a set of important programming idioms that we use as expressiveness benchmarks for our various modular typechecking algorithms. Section 3 focuses on the challenges and solutions for modular typechecking, describing the two sets of restrictions that define the modularity spectrum as well as an interesting intermediate point. Section 4 discusses related work, and section 5 concludes.

2 The Dubious Language

2.1 Syntax and Semantics

Dubious's syntax appears in figure 1.* Formal dynamic semantics (evaluation rules) and a base static semantics (typechecking rules), which will be extended for each type system we discuss, appear in appendices A.1 and B.1, respectively. Because we focus on the issue of modular typechecking of symmetric multimethods, we omit many useful but less relevant features. Dubious includes:

- a classless object model with explicitly declared objects and inheritance (but omits dynamically created objects and state),
- first-class generic functions (but omits lexically nested closures),
- explicitly declared function types (but omits explicitly declared object types and subtyping independent of objects and inheritance†), and
- modules to support separate typechecking (but omits nested modules, parameterized modules, and encapsulation mechanisms).

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 directly inherits (its parents). As in other classless languages [LaLonde *et al.* 86, Lieberman 86, Ungar & Smith 87, Chambers 95, Abadi & Cardelli 96], objects play the roles of both classes and instances, and the **`isa`** relation accordingly plays the roles of both inheritance and instantiation. For example, in the modules of figure 2, the `colorPoint` object represents a subclass of `point` that has a color, while the `origin` object represents a particular `point` instance. If an object is declared **`abstract`**, it is used solely as a template for other objects and may not be referred to in expressions.

* We use brackets to delimit optional parts of the syntax.

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

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

Figure 2: A simple Dubious program fragment

For example, the abstract `graphic` object in figure 2 defines an interface for objects that can be drawn, which is then implemented by the `point` object and descendants.

An object can also act like a (generic) function by *attaching* methods to it, via the **`has method`** declaration. This is a destructive operation, like the method update construct in the calculi of Abadi and Cardelli [Abadi & Cardelli 95, Abadi & Cardelli 96]. In the `PointMod` module example in figure 2, the method added to the `draw` object has two formal parameters, named `p` and `d`. The first formal is *specialized* by providing the `@point` suffix, which specifies the object on which the argument is dynamically dispatched. While this method is specialized only on its first argument, the method added to the `equal` object is a multimethod, dynamically dispatching on both of its arguments.

To evaluate a generic function application (i.e., a message send) $E_0(E_1, ..., E_n)$, we evaluate each expression E_i to some object o_i and then invoke the *most-specific applicable method* attached to object o_0 for $(o_1, ..., o_n)$. For ease of description, we consider each unspecialized formal to be specialized on a distinguished `any` object from which every other object inherits. A method in o_0 is *applicable* to $(o_1, ..., o_n)$ if the method has n arguments and if $(o_1, ..., o_n)$ pointwise inherits from the tuple of the method's specializers. The *most-specific* applicable method is the unique applicable method whose specializers pointwise inherit 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)`. Of the three declared `equal` methods in figure 2, the (`point` × `point`) and (`colorPoint` × `colorPoint`) methods are applicable, and the (`colorPoint` × `colorPoint`) method is the most-specific one.

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

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

Dubious's static type system ensures that legal programs do not have message-not-understood or message-ambiguous errors. Each generic function object is declared with an *arrow type* that defines the legal arguments and return values of the function. For example, the `equal` generic function in the `PointMod` module in figure 2 is declared to accept two objects each of which is `point` or a descendant, and to return `bool` or a descendant. Descendants of `equal` inherit this arrow type. For simplicity, each generic function may directly conform to at most one arrow type. The ordinary contravariant subtyping rule relates arrow types [Cardelli 84].

In the Dubious type system, a *type* is either an object or an arrow type, and an *abstract type* is either an abstract object or an arrow type. We say that an object o *conforms* to a type t if either o inherits from t or o has an arrow type that is a subtype of t.

Ruling out message-not-understood and message-ambiguous errors involves two kinds of checks: *client-side* and *implementation-side* [Chambers & Leavens 95]. Client-side checks are local checks on declarations and expressions. The most important client-side check is that for each message send expression $E(E_1, ..., E_n)$ in the program, E conforms to an arrow type $(T_1, ..., T_n) \rightarrow T$ and each E_i conforms to T_i. The message send is then given the type T. Implementation-side checks ensure that each generic function o correctly implements the arrow type $(T_1, ..., T_n) \rightarrow T$ to which it conforms. For every object tuple $(o_1, ..., o_n)$ such that each o_i conforms to T_i and o_i is not abstract, there must exist a most-specific applicable method in o. For example, consider implementation-side checks on the `equal` generic function. `equal` is declared to conform to (`point`, `point`)→`bool`. Nine argument tuples pointwise conform to (`point`, `point`):

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.

This description suggests a straightforward typechecking algorithm. Client-side checks on the declarations in a module require only type information from imported modules, and they can therefore be performed on a module-by-module basis. The implementation-side checks as described above assume global knowledge of the program's objects and methods. Therefore, implementation-side typechecking on all generic functions is deferred until link-time, when all modules are present. This link-time typechecking approach is used for typechecking previous languages with symmetric multimethods, including Kea [Mugridge *et al*. 91], Cecil [Chambers 95], $ML_{\leq}$ [Bourdoncle & Merz 97], and Tuple [Leavens & Millstein 98]. We refer to this global typechecking algorithm as *System G*. Appendix B.2 shows the extensions to the base static semantics for System G. We have proven this static semantics sound for Dubious, and our proof technique is sketched in appendix C.

2.2 Example Programming Idioms

There are several flexible programming idioms that we would like to be able to express and statically typecheck in Dubious. This subsection introduces these idioms. We will use these idioms as "benchmarks" for evaluating the modular typechecking algorithms of section 3. The global typechecking of System G supports all these idioms.

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

We also wish to allow a module to define an abstract object, whose operations are not required to be implemented for the abstract object. Abstract objects are used solely as templates for other objects. For example, the `GraphicMod` module in figure 2 defines an interface for graphical objects. Because the `graphic` object is abstract, it is ignored during implementation-side typechecking, allowing its `draw` generic function to be incompletely 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 the `graphic` interface. Clients of the abstract interface need not be aware of the various concrete implementations.

Several more 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, *binary (multi)methods* result, which enable a simple solution to the "binary method" problem [Bruce *et al.* 95]. For example, the `equal` generic function in figure 2 is a binary method, accepting two arguments of the same type. Binary multimethods allow easy programming of the natural semantics for equality on various combinations of points. At the same time, `colorPoint` and `origin` remain safe subtypes of `point`, so subtype polymorphism over the point hierarchy is perfectly legal.

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

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

If we remove all restrictions on method specializers, we obtain what we call *arbitrary multimethods*. Using arbitrary multimethods, we can program special behavior for equality on one `point` and one `colorPoint` (in that order), without modifying either the `PointMod` or `ColorPointMod` modules, with the following code:

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

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

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

Open objects arise naturally in languages based on multimethods, but they are useful even for singly dispatched methods, as in the case of the `distance` generic function above. Open objects support simpler programming of the "visitor" design pattern [Gamma *et al.* 95] where client-specific visitors are programmed directly without needing to build a special visitor infrastructure for each object hierarchy. In the `lineseg` example, the `distance` generic function is a kind of client-specific visitor of the `point` hierarchy. New descendants of `point` can be added in other modules without modifying or breaking the `distance` method (as long as the implementations of `distance` inherited by new descendants are still appropriate), and new operations can be added to existing object hierarchies without modifying or breaking those hierarchies.

3 Modular Typechecking

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

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

Figure 3: Ambiguity problems with arbitrary multimethods

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

Figure 4: Incompleteness problems with multimethods on abstract objects

```
module RotateMod imports ColorPointMod, OriginMod {
  object rotateAboutOrigin (point, int)→point
  rotateAboutOrigin has method(p@point, i) { ... }
  rotateAboutOrigin has method(cp@colorPoint, i) { ... }
  rotateAboutOrigin has method(o@origin, i) { o }
}
module ColorOriginMod imports ColorPointMod, OriginMod {
  object colorOrigin isa colorPoint, origin
}
```

Figure 5: Ambiguity problems combining open objects with multiple inheritance

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

Figure 6: Incompleteness problems combining open objects with abstract objects

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

Unfortunately, applying such a modular typechecking scheme to the unrestricted Dubious language is unsound. It is possible for two importers of a module to pass these checks in isolation but still cause message-not-understood or message-ambiguous errors when combined in a single program [Chambers & Leavens 95]. Subsection 3.1 describes the situations that can lead to such errors. Subsections 3.2, 3.3, and 3.4 describe three different sets of restrictions that support safe modular typechecking, representing different tradeoffs between expressiveness, modularity, and complexity. Subsection 3.5 summarizes the key features of the various type systems discussed in this paper.

3.1 Challenges for Modular Typechecking

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

First, the ability to add arbitrary multimethods anywhere in the program can cause undetected ambiguities. Figure 3 shows a simple example of the problems that can occur. Each of the `ColorPointMod` and `OriginMod` modules from figure 2 is augmented with a second `equal` method. Each module typechecks in isolation, given only information about its imported modules. From the `ColorPointMod` module's point of view, every visible legal argument tuple to the `equal` generic function has a single, most-specific method implementation, and similarly for the `OriginMod` module. However, when the two modules are combined in a single

program, a run-time message-ambiguous error will occur if the message `equal(colorPoint,origin)` is ever sent, since neither of the methods in the example is more specific than the other.

One way to solve this problem is to break the symmetry of the dispatching semantics. For example, if we linearized the specificity of argument positions, comparing specializers lexicographically left-to-right (rather than pointwise) as is done in Common Lisp [Steele 90, Paepcke 93] and Polyglot [Agrawal *et al.* 91], then the `(colorPoint × point)` method would become more specific than the `(point × origin)` method. However, one of our major design goals for Dubious is to retain the symmetric multimethod dispatching semantics, which we believe is more natural and less error-prone, since it reports potential ambiguities rather than silently resolving them. The symmetric semantics is used in the languages Cecil [Chambers 92, Chambers 95], Dylan [Shalit 97, Feinberg *et al.* 97], the λ&-calculus [Castagna *et al.* 92, Castagna 97], Kea [Mugridge *et al*. 91], $ML_{\leq}$ [Bourdoncle & Merz 97], and Tuple [Leavens & Millstein 98].

A second unsafe scenario involves the combination of abstract objects with multimethods. For example, suppose we want to make the `point` object in figure 2 be abstract, so that it need not be fully implemented. Figure 4 shows the relevant parts of the revised modules. Each module passes implementation-side typechecks on the `equal` generic function in isolation, as all legal tuples of concrete objects have a single, most-specific implementation from each module's point of view. However, at run-time a message-not-understood error can 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 `RotateMod` module extends the interface of points from figure 2 with a function for rotating a point about the origin. From this module's point of view, the `rotateAboutOrigin` generic function is completely and unambiguously implemented. Independently, the `ColorOriginMod` module creates a new point object that multiply inherits from `colorPoint` and `origin`, introducing the possibility of ambiguities. Since this module does not know about the `rotateAboutOrigin` generic function, it has no way of statically detecting the ambiguity `rotateAboutOrigin(colorOrigin,i)`, which can therefore cause a run-time error.

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

The final unsafe scenario involves the combination of open object idioms with abstract objects. In figure 6, the `EraseMod` module extends the `graphic` interface from figure 2 with an operation for erasing the graphical object. Since there are no concrete descendants of `graphic` from this module's point of view, the `erase` generic function appears to be completely implemented.[*] The `MyGraphicMod` creates a concrete implementation of a graphic object. From its point of view, the `graphic` interface is completely implemented for `myGraphic`. However, at run-time there will be a message-not-understood error if the message send `erase(myGraphic, d)` occurs (where `d` is some `display` descendant).

3.2 System M: Maximizing Modularity

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

We say that an object or method is *local* if it is declared in the current module, and otherwise it is *non-local*. An arrow type is local if all of its argument types and result type are local, and otherwise it is non-local. A concrete object *o* is an *orphan* if *o* conforms to some non-local type *t* but does not inherit from any other concrete conformer to *t*. Finally, two types are *related* if one is an object that conforms to the other, and otherwise they are *unrelated*. (In the case when both types are objects, being related means that one object inherits from the other.)

Informally, System M removes multimethod ambiguities by forcing methods of a generic function to agree on a designated argument position on which to have a local specializer, ensuring that a method cannot conflict with an unseen method. Multiple inheritance ambiguities are removed by disallowing multiple inheritance across module boundaries. Potential incompletenesses are removed by forcing the existence of appropriate default methods. More precisely, System M imposes the following four restrictions on each module:

- (***M1***) Each method added to a non-local generic function must specialize on its designated argument position, and the specializer must be a local object. (All independent extensions to a generic function will add encapsulated-style methods with disjoint tuples of applicable arguments, in the absence of unseen multiple inheritance.)
- (***M2***) If a local object conforms to two non-local types, those types must be related. (No non-trivial multiple inheritance/conformance across module boundaries is allowed.)
- (***M3***) If a local generic function has an argument position *i* with a non-local argument type, then implementation-side typechecks must include a dummy concrete conformer to position *i*'s type in the set of objects checked during implementation-side typechecking of that generic function. (This dummy acts as a placeholder for all unseen concrete conformers, ensuring that appropriate default methods exist.)
- (***M4***) If a local generic function has a non-designated position *i* that has an abstract argument type, then implementation-side typechecks must include a dummy concrete conformer to position *i*'s type in the set of objects checked during implementation-side typechecking of that generic function. (This dummy acts as a placeholder for all unseen concrete conformers, ensuring that appropriate default methods exist.)

By imposing the above restrictions, we can ensure safety in Dubious while meeting the modularity goals described at the

[*] Even if there were visible concrete children of `graphic`, complete implementation of these visible children does not imply complete implementation of unseen concrete children such as `myGraphic`.

beginning of the section. Each module implementation-side typechecks its local generic functions given only the interfaces of its importees. In addition, there are two scenarios in which a module must re-check importees:

- If the module adds methods to a non-local generic function, then this generic function is implementation-side typechecked. However, only legal argument tuples that apply to a local method need be checked.
- If the module creates an object *o* which is an orphan, then all visible non-local generic functions accepting *o* as an argument on the designated position are implementation-side typechecked. However, only legal argument tuples containing *o* at that position need be checked.

For simplicity, in the rest of this paper we will assume that the designated argument position for all generic functions is the first argument. Appendix B.3 contains the formal description of these rules, which are sufficient to ensure the absence of run-time type errors, regardless of how a module is eventually linked with other modules. We have proven this type system sound for Dubious, and our proof technique is sketched in appendix C.

We can use the restrictions to resolve the problems in the examples of the previous subsection. In figure 3, the `equal` method in the `OriginMod` would cause a static type error, because it violates restriction ***M1*** by using a non-local first specializer. The restriction would be satisfied if the method instead had the form

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

and indeed this removes the multimethod ambiguity. Forcing both methods in figure 3 to have a local first specializer ensures that no argument tuple will be applicable to both of them (unless an object multiply inherits from both first specializers, which is disallowed by restriction ***M2***).

In figure 4, both the `colorPoint` and `origin` objects are orphans. Because `equal` accepts the `colorPoint` and `origin` objects on its designated position, it is re-checked by the `ColorPointMod` and `OriginMod` modules. By restriction ***M4***, these checks must consider a dummy conformer to the abstract `point` type on the non-designated argument position. Therefore, re-checks from the `ColorPointMod` module will find an incompleteness for the argument tuple `(colorPoint,pointDummy)`, and similarly for the `OriginMod` module's checks. Therefore, methods must be created to cover these cases safely. For example, the `ColorPointMod` module could include a method declaration

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

and similarly for the `OriginMod` module.

In figure 5, the `colorOrigin` object fails restriction ***M2*** because it inherits from two unrelated non-local objects, so such an object cannot be programmed. Because of the conflict between (only partially ordered) multiple 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 multiple inheritance across import boundaries because of the importance of open objects. Arbitrary multiple inheritance is still possible within a module.[*]

In figure 6, the `erase` generic function's first argument position has the non-local `graphic` type. By restriction ***M3***, implementation-side typechecks on this generic function must assume that some dummy conformer to `graphic` may be sent to that position. Therefore, an incompleteness is found, which is fixed by adding a default method implementation specializing on `graphic` in the first position (or having no specializers). Adding such a method allows the module to typecheck, and ensures that the message send `erase(myGraphic, d)` now has a single, most-specific method implementation to invoke.

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

3.3 System E: Maximizing Expressiveness

Although the set of restrictions in System M provides completely modular typechecking for Dubious, there are certain idioms that cannot be expressed. Notable among these is multiple inheritance across modules and arbitrary multimethods, where the designated argument position does not have a local specializer. In this section, we present *System E*, whose goal is to express all idioms, at the cost of a small link-time check. Subject to this constraint, we would like to retain modular typechecking for as many idioms as possible.

A previous work informally introduced the idea of *extending* a module as an alternative to importing it [Chambers & Leavens 95]. If each module rechecks all the generic functions from modules that it extends and at link-time each module is checked for a unique most-extending module, then the program is guaranteed to be safe. The most-extending module statically sees all relevant parts of the program, so its checks are enough to ensure run-time safety of all the modules it extends. In this way, groups of extenders partition the program into regions that must eventually be typechecked together, but still separately from all other regions. As a result, extenders can safely be given much more flexibility than importers.

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

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

This revised syntax requires a slight modification to the module evaluation rule in the dynamic semantics. The new rule is shown in appendix A.2.

In the context of System E, we say that an object or method is *local* if it was declared in the current module, *extended* if it was declared in an extended module, and *imported* otherwise. A *non-local* object is either imported or extended, and similarly for a *non-imported* and *non-extended* object. An arrow type is local if its argument and result types are local, extended if it has only local or extended argument and result types, and otherwise it is imported. A concrete object *o* is an *orphan* if *o* conforms to some imported type *t* but does not inherit from any other concrete conformer to *t*. A module m_1 is the most-extending

[*] If Dubious had a notion of *interface* like that of Java [Arnold & Gosling 98, Gosling *et al.* 96], then multiple conformance to interfaces across import boundaries would still be safe, because an interface cannot be an argument specializer.

```
module ResolveColorPointAndOriginMod extends ColorPointMod, OriginMod {
  equal has method(cp@colorPoint, o@origin) { … }
}
module ResolveRotateModAndColorOriginMod extends RotateMod, ColorOriginMod {
  rotateAboutOrigin has method(co@colorOrigin, i) { … }
}
```

Figure 7: Extension modules

module of m_2 if m_1 extends every module that extends m_2, using the reflexive, transitive closure of the module extension relation.

In the typechecking rules, a module is unrestricted in its interactions with extended objects, generic functions, and types. Consequently, all of our benchmark idioms are available to extenders. However, because of the greater potential effects of unseen extenders, importers must obey stricter restrictions than those of System M. To ensure disjointness of applicable argument tuples across independent importers, all specializers of methods added to imported generic functions must be non-imported objects (not just the specializer of a designated argument, as in System M). In addition, multiple inheritance and open objects are disallowed in importers, in order to allow extenders to safely program these idioms.

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

R ::= $(S_1, \ldots, S_n) \rightarrow T$
S ::= $[\#]T$

A **#** marker denotes a *marked argument position* of a generic function, meaning that methods may specialize at that position. We impose the rule that extenders may not specialize on arguments lacking a **#** marker, which then relieves importers of the onerous requirement of specializing on unmarked arguments. Given this language extension, System E imposes the following four restrictions on modules, analogous to the three restrictions of System M:

- (***E1a***) If a method is added to an imported generic function, the method must have a non-imported specializer at all positions marked with **#** in the generic function's arrow type, and (***E1b***) a method added to any generic function (local, extended, or imported) may not specialize at any position not marked with **#** in the generic function's arrow type. (All independent additions to an imported generic function will be binary-style methods having disjoint tuples of applicable arguments, in the absence of unseen multiple inheritance.)
- (***E2***) If a local object o conforms to an imported type t, then all non-local types to which o_1 conforms must be related to one another. (Conformance to an imported type disallows multiple inheritance/conformance across module boundaries.)
- (***E3a***) A marked argument position to a local generic function cannot have an imported type. (Imported objects are not "open.") (***E3b***) If a marked argument position of a local generic function has an extended type, then implementation-side typechecks must include a dummy concrete conformer to position i's type in the set of objects checked during implementation-side typechecking of that generic function. (This dummy acts as a placeholder for all unseen concrete conformers, ensuring that appropriate default methods exist.)
- (***E4***) If a local generic function has more than one marked argument position and such a position i has an abstract argument type, then implementation-side typechecks must include a dummy concrete conformer to position i's type in the set of objects checked during implementation-side typechecking of that generic function. (Multiply dispatched generic functions with abstract types must have appropriate default methods.)

To typecheck a module, we need only the interfaces of extended and imported modules. All extended generic functions are re-implementation-side typechecked. In addition, the two conditions for re-checking imported generic functions from System M apply here as well, with one modification: the second re-check holds for any marked argument position, rather than the designated position.

A simple 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. Appendix B.4 contains the formal semantics for extenders and importers.

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

However, the more expressive programming idioms must use extenders, which can now resolve the problems from section 3.1 that System M could not solve. In figure 3, the revised `ColorPointMod` and `OriginMod` modules must use the clause **`extends`** `PointMod` instead of **`imports`** `PointMod` because they each violate restriction ***E1a*** by having an imported specializer on the `equal` method (the `ColorPointMod` module still need only import the `ColorMod` module). Further, if the two modules are combined in the same program, an additional module must be written that extends them both, in order for the `PointMod` module to have a unique most-extending module. In order for checks on the `equal` generic function to succeed in this new module, it must include the necessary declarations to fix the ambiguity, as

shown in the `ResolveColorPointAndOriginMod` module in figure 7.

An analogous solution is used to resolve the problem illustrated in figure 5. Assuming the arrow type for `rotateAboutOrigin` has a **#** marker on its first argument position, `rotateAboutOrigin` violates restriction ***E3a*** by having an imported type on a marked argument position, and `colorOrigin` violates restriction ***E2*** by multiply inheriting from non-local objects. Therefore, both modules must extend `point` rather than import it, requiring the existence of a most-extending module to fix the ambiguity. This module is shown in figure 7. Further, if both of the modules in figure 7 were combined in a single program, a module extending both of these would be required, in order to provide the `PointMod` module with a single most-extending module. (Because there are no conflicts, this module could be empty.)

.In figure 4, both the `colorPoint` and `origin` objects are orphans of type `point`. Because `equal` accepts these objects on a specialized position, it is re-checked by the `ColorPointMod` and `OriginMod` modules. By restriction ***E4***, these checks must consider a dummy conformer to `point` on both argument positions. In particular, re-checks from the `ColorPointMod` module will find an incompleteness for the argument tuples `(colorPoint, pointDummy)` and `(pointDummy,colorPoint)`, and similarly for the `OriginMod` module's checks. Therefore, a default method implementation such as

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

must be created to cover these cases safely.

In figure 6, the `erase` generic function, assuming that it has a **#** marker on its first argument position, violates restriction ***E3a*** by having the imported `graphic` type on the first argument. Therefore, the `EraseMod` module must extend the `GraphicMod`. Then by restriction ***E3b***, implementation-side checks on `erase` must assume the existence of a dummy conformer to `graphic`, requiring a method declaration to handle this confomer, such as

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

This method safely handles the unseen `myGraphic` object.

In summary, 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 checking is still performed in a modular fashion, but each module re-checks its extended modules. The modular checks in the most-extending module of a region ensure the safety of that region, and each region's checking is separate from the others'. A simple link-time check ensures that each module has a most-extending module. By sacrificing complete modularity, System E provides all of our expressive programming idioms, including arbitrary multimethods and multiple inheritance.

3.4 System ME: Combining Modularity and Expressiveness

Systems M and E define a spectrum of possible modular type systems. System M maximizes the language's modularity of typechecking. The cost for this modularity is a loss of certain expressive programming idioms, including arbitrary multimethods and multiple inheritance. System E maximizes the language's expressiveness, at the cost of some regional typechecking, a simple link-time check for extenders, and more restrictions on what can be expressed in importers.

By selecting different subsets of restrictions from each of Systems M and E, it is possible to design safe modular type systems with intermediate abilities between these two extremes. In this subsection, we describe one interesting point in this spectrum. In particular, we define *System ME* to be essentially the union of the restrictions in Systems M and E. Each generic function uses System M's restrictions by default, but a generic function may be given an arrow type with **#** markers on argument positions, in which case System E's rules apply to it. These restrictions then allow extenders to add arbitrary multimethods to the generic function.

More precisely, System ME includes restrictions ***M1-M4*** and ***E1-E4***, with a few modifications. The generic functions referred to in restrictions ***M1***, ***M3***, and ***M4*** are only those generic functions whose associated arrow type has no **#** markers, while the generic functions referred to in restrictions ***E1***, ***E3***, and ***E4*** are only those generic functions whose associated arrow type has at least one **#** marker on an argument position. Restriction ***M2*** subsumes restriction ***E2***, disallowing multiple inheritance across module boundaries. Appendix B.5 contains the formal semantics for System ME.

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

3.5 Summary

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

In summary, we have presented several alternatives for modular typechecking of multimethods, ranging from a completely modular approach that sacrifices certain programming idioms, to a completely expressive approach that requires some

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

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

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 require regional checking, and which kinds of restrictions are simpler or more intuitive than others. A practical evaluation of these systems is necessary to better understand these issues. We plan to undertake such an evaluation in the context of Diesel, a full programming language succeeding Cecil that will incorporate Dubious's work on modular type systems for symmetric multimethods.

4 Related Work

Chambers and Leavens made the first effort toward modular typechecking of symmetric multimethods [Chambers & Leavens 95], in the Cecil language [Chambers 92, Chambers 95]. They defined client-side and implementation-side typechecking and sketched informal ideas for modular typechecking, including the notions of extension modules and unique most-extending modules. In this sketch, objects could not conform to an imported type, forcing much of a program into extension modules. They did not formalize the static or dynamic semantics of their modular language nor prove any soundness results for modular typechecking.

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

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

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

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

Encapsulated multimethods [Castagna 95, Bruce *et al.* 95] are an attempt to solve the modularity problem for multimethods by embedding multimethods in the traditional object-oriented class model. An encapsulated multimethod first dispatches on the receiver argument, then dispatches on the remaining arguments. All the multimethods with a given receiver are encapsulated within the receiver class and are not extensible or inheritable outside this class. 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 the open object idiom and abstract methods. 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. Parasitic multimethods [Boyland & Castagna 97] are a variant of encapsulated multimethods adapted to Java. Parasitic multimethods are additionally complicated by the use of the textual order of methods to resolve ambiguities, the inheritance

of parasitic methods in the presence of this textual ordering, and the need to retain backward compatibility with Java's blend of dynamic single dispatching and static overloading on arguments.

Tuple [Leavens & Millstein 98] is a language that provides dispatching on tuples in order to add 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.

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

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

5 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 objects, inheritance, and first-class generic functions. We have defined a spectrum of modular type systems for Dubious, ranging from complete modularity at the expense of giving up certain programming idioms, to complete expressiveness at the cost of regional typechecking and a simple link-time check on regions. Dubious represents the first formalization and the first proof of soundness of separate typechecking for symmetric multimethods. We are working on soundness proofs for Systems E and ME to augment our proofs for Systems G and M.

There are several language features that are less central to the problem of separate typechecking but would be useful to add to Dubious. These include simple features like dynamic object creation, instance variables holding (mutable) object state, and local variable declarations in an expression. More interesting extensions include supporting polymorphic types in the presence of separate typechecking; module types, first-class modules, and parameterized modules; block structure with arbitrarily nested declarations; a more sophisticated dispatching scheme [Ernst *et al.* 98]; and encapsulation. Finally, we are using the ideas in Dubious as a foundation for the design of Diesel, a practical programming language succeeding Cecil.

Acknowledgments

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Wright & Felleisen 94] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, **115**(1):38–94, November 1994.

Appendix A Dynamic Semantics

A.1 Systems G and M

Domains for the Dynamic Semantics:

$e \in Env = (I \times Obj)^*$
$s \in Store = Isa \times GFMethods$
$me \in ModEnv = (I \times Env)^*$
$isa \in Isa = (Obj \times Obj)^*$
$gfms \in GFMethod = ((Obj \times MethodHeader) \times MethodBody)^*$
$mh \in MethodHeader = Obj^*$
$mb \in MethodBody = I^* \times E \times Env$
$o \in Obj = \mathbf{obj}(Nat)$

Definitions and functions:

We define accessor functions on the components of a *Store,* with the following names: (*isa,gfms*).
$r_1 \,\&\, r_2 = \{(x,y_1) \mid (x,y_1) \in r_2$ or $[(x,y_1) \in r_1$ and $\sim\exists\, y_2.(x,y_2) \in r_2]\}$
$s_1 + s_2$ = pointwise union on two *Stores*
$name(I_1 \ \mathtt{@}\ I_2) = I_1 \quad name(I_1) = I_1$
Each static occurrence of "object" in the program is subscripted with a unique integer greater than one, acting as the object's unique identity.
$disjoint(x_1,...,x_n) = \forall\ 1 \le i \le n.\ \forall\ 1 \le j \le n.\ (x_i = x_j) \Longrightarrow (i = j)$

[prg-d]
$$\frac{\vdash M_1 \ldots M_n \succ s,me \qquad e = me(I) \qquad e,s \vdash E \Rightarrow o}{\vdash M_1 \ldots M_n\ \mathbf{import}\ I\ \mathbf{in}\ E\ \mathbf{end} \Rightarrow o \succ s}$$

[md*-d]
$$\frac{\{\} \vdash M_1 \succ s_1,me_1 \quad me_1 \vdash M_2 \succ s_2,me_2 \ldots \quad me_1 \& me_2 \& \ldots \& me_{n-1} \vdash M_n \succ s_n,me_n}{\vdash M_1 \ldots M_n \succ s_1+s_2+\ldots+s_n,\ me_1 \& me_2 \& \ldots \& me_n}$$

[mod-d]
$$\frac{e = me(I_1)\&\ldots\&me(I_t) \qquad e \vdash D_1 \ldots D_n \succ e_0,s_0}{me \vdash \mathbf{module}\ I\ \mathbf{imports}\ I_1,\ldots,I_t\ \{D_1 \ldots D_n\} \succ s_0,\ \{(I,e\&e_0)\}}$$

[dc*-d]
$$\frac{e \vdash D_1 \succ e_1,s_1 \quad e\&e_1 \vdash D_2 \succ e_2,s_2 \quad \ldots \quad e\&e_1\&e_2\&\ldots\&e_{n-1} \vdash D_n \succ e_n,s_n}{e \vdash D_1 \ldots D_n \succ e_1\&e_2\&\ldots\&e_n,\ s_1+s_2+\ldots+s_n}$$

[obj-d]
$$\frac{e \vdash I_1 \Rightarrow o_1 \ldots e \vdash I_n \Rightarrow o_n \qquad o = \mathbf{obj}(i)}{e \vdash [\mathbf{abstract}]\ \mathbf{object}_i\ I\ [R]\ \mathbf{isa}\ I_1,\ldots,I_n \succ \{(I,o)\},(\{(o,o_1),\ldots,(o,o_n)\},\{\})}$$

[has-d]
$$\frac{e,s \vdash I \Rightarrow o \quad e \vdash F_1 \Rightarrow o_1 \ldots e \vdash F_n \Rightarrow o_n \quad mb = ([name(F_1),\ldots,name(F_n)],E,\ e)}{e \vdash I\ \mathbf{has\ method}\ (F_1,\ldots,F_n)\ \{E\} \succ \{\},(\{\},\{((o,[o_1,\ldots,o_n]),mb)\})}$$

[app-d]
$$\frac{e,s \vdash E_0 \Rightarrow o_0 \ldots e,s \vdash E_n \Rightarrow o_n \quad s \vdash o_0\ lookup\text{-}method\ [o_1,\ldots,o_n] \succ (mh,([I_1,\ldots,I_n],E,e_0)) \quad e_0\ \&\ \{(I_1,o_1),\ldots,(I_n,o_n)\},s \vdash E \Rightarrow o}{e,s \vdash E_0(E_1,\ldots,E_n) \Rightarrow o}$$

[id-d]
$$\frac{e(I) = o}{e,s \vdash I \Rightarrow\ o}$$

[spe-d]
$$\frac{e \vdash I_2 \Rightarrow o}{e \vdash I_1\mathtt{@}I_2\ \Rightarrow\ o}$$

[uns-d]
$$e \vdash I\ \Rightarrow\ \mathbf{obj}(1)$$

[look-d]
$$\frac{[mh_1,\ldots,mh_n] = [mh \mid ((o,mh),mb) \in gfms(s) \wedge s \vdash [o_1,\ldots,o_m] \le_{isa^*} mh] \quad disjoint(mh_1,\ldots,mh_n) \quad 1 \le i \le n \quad s \vdash mh_i \le_{isa^*} mh_1 \ldots s \vdash mh_i \le_{isa^*} mh_n \quad ((o,mh_i),mb_i) \in gfms(s)}{s \vdash o\ lookup\text{-}method\ [o_1,\ldots,o_m] \succ (mh_i,mb_i)}$$

[isa*-d]
$$\frac{s \vdash o_1 \le_{isa} o_1' \ldots s \vdash o_n \le_{isa} o_n'}{s \vdash [o_1,\ldots,o_n] \le_{isa^*} [o_1',\ldots,o_n']}$$

[isa-d]
$$s \vdash o \le_{isa} \mathbf{obj}(1)$$

[isb-d]
$$\frac{(o_1,o_2) \in isa(s)}{s \vdash o_1 \le_{isa} o_2}$$

[isr-d]
$$s \vdash o \le_{isa} o$$

[ist-d]
$$\frac{s \vdash o_1 \le_{isa} o_2 \quad s \vdash o_2 \le_{isa} o_3}{s \vdash o_1 \le_{isa} o_3}$$

A.2 Modification for Systems M and ME

[mod-d]
$$\frac{e = me(I_1)\&\ldots\&me(I_t)\&me(I_1')\&\ldots\&me(I_r') \qquad e \vdash D_1 \ldots D_n \succ e_0,s_0}{me \vdash \mathbf{module}\ I\ \mathbf{imports}\ I_1,\ldots,I_t\ \mathbf{extends}\ I_1',\ldots,I_r'\ \{D_1 \ldots D_n\} \succ s_0,\ \{(I,e\&e_0)\}}$$

Appendix B Static Semantics

B.1 Base Static Semantics

Domains for the static semantics:

$p \in TypeEnv = (I \times ObjType)^*$
$k \in TypeStore = Conforms \times Isa \times Abstract \times GFMethods \times LocalStore$
$m \in Modules = ModEnv \times ModStore$
$ot \in ObjType = Const + \mathbf{unk}(Type)$
$c \in Const = \mathbf{object}(Nat)$
$t \in Type = Const + Arrow$
$art \in Arrow = \mathbf{arrow}(Type^*, Type)$
$con \in Conforms = (Const \times Arrow)^*$
$abs \in Abstract = Const^*$
$isa \in Isa = (Const \times Const)^*$
$gfms \in GFMethod = (Const \times MethodHeader)^*$
$mh \in MethodHeader = Const^*$
$lk \in LocalStore = Locals \times LocalGFMethods$
$ls \in Locals = Const^*$
$lgf \in LocalGFMethods = GFMethods$
$mp \in ModEnv = (I \times TypeEnv)^*$
$mk \in ModStore = (I \times TypeStore)^*$

Definitions and functions:

We define names for the components of a *TypeStore* and *Module*, and associated accessor functions with those names: $(con,isa,abs,gfms,(ls,lgf))$ and (mp,mk)
Let $()$ denote the empty type store and
$(x_1 = v_1,...,x_n = v_n)$ denote the empty type store augmented with component x_i equal to v_i.
$r_1 \,\&\, r_2 = \{(x,y_1) \mid (x,y_1) \in r_2$ or $[(x,y_1) \in r_1$ and $\sim\exists\, y_2.(x,y_2) \in r_2]\}$
k_1+k_2 = pointwise union on a tuple of sets.
$name(\mathbf{module}\ I\ \mathbf{exports}\ (I^*)\ \{D^*\}) = I$
$name(I_1@I_2) = I_1 \quad name(I) = I$
$disjoint(x_1,...,x_n) = \forall\ 1 \le i \le n.\ \forall\ 1 \le j \le n.\ (x_i = x_j) \Longrightarrow (i = j)$
$length([x_1,...,x_n]) = n$
Each static occurrence of "object" in the program is subscripted with a unique integer greater than one, acting as the object's unique identity.

[prg-s]
$$\frac{\vdash M_1 \ldots M_n \succ k,m \qquad p = (mp(m))(I) \qquad p,k \vdash E : \mathbf{unk}(t)}{\vdash M_1 \ldots M_n\ \mathbf{import}\ I\ \mathbf{in}\ E\ \mathbf{end} : \mathbf{unk}(t) \succ k}$$

[md*-s]
$$\frac{(\{\},\{\}) \vdash M_1 \succ m_1 \quad m_1 \vdash M_2 \succ m_2 \ldots m_1+m_2+\ldots+m_{n-1} \vdash M_n \succ m_n \quad disjoint(name(M_1),\ldots,name(M_n)) \quad k = mk(m_1)(name(M_1))+\ldots+mk(m_n)(name(M_n)) \quad m = m_1+m_2+\ldots+m_n \quad k,m \vdash program\text{-}has\text{-}safe\text{-}modules}{\vdash M_1 \ldots M_n \succ k,m}$$

[mod-s]
$$\frac{p = mp(m)(I_1)\&\ldots\&mp(m)(I_t) \quad k = add\text{-}imp(mk(m)(I_1))\&\ldots\&add\text{-}imp(mk(m)(I_t)) \quad p,k \vdash D_1 \ldots D_n \succ p_0,k_0 \quad k+k_0 \vdash module\text{-}has\text{-}safe\text{-}gfs \quad k+k_0 \vdash module\text{-}has\text{-}safe\text{-}imported\text{-}gfs \quad k+k_0 \vdash module\text{-}has\text{-}safe\text{-}objects \quad k+k_0 \vdash module\text{-}has\text{-}safe\text{-}methods}{m \vdash \mathbf{module}\ I\ \mathbf{imports}\ I_1,\ldots,I_t\ \{D_1 \ldots D_n\} \succ (\{(I,p\&p_0)\},\{(I,k+k_0)\},)}$$

[dc*-s]
$$\frac{p,k \vdash D_1 \succ p_1,k_1 \quad p\&p_1,k+k_1 \vdash D_2 \succ p_2,k_2 \quad \ldots \quad p\&p_1\&p_2\&\ldots\&p_{n-1},k+k_1+k_2+\ldots+k_{n-1} \vdash D_n \succ p_n,k_n}{p,k \vdash D_1 \ldots D_n \succ p_1\&p_2\&\ldots\&p_n,k_1+k_2+\ldots+k_n}$$

[obj-s]
$$\frac{p,k \vdash I_1 : c_1 \ldots p,k \vdash I_n : c_n \quad \{t_1,\ldots,t_q\} = \{t \mid 1 \le j \le n \wedge (c_j,t) \in con(k)\} \quad (q=0 \vee q=1) \quad k_0 = (con = \{(\mathbf{object}(i),t_1),\ldots,(\mathbf{object}(i),t_q)\}, isa = \{(\mathbf{object}(i),c_1),\ldots,(\mathbf{object}(i),c_n)\}, ls = \{\mathbf{object}(i)\})}{p,k \vdash \mathbf{object}_i\ I\ \mathbf{isa}\ I_1,\ldots,I_n \succ \{(I,\ \mathbf{object}(i))\},k_0}$$

[abso-s]
$$\frac{p,k \vdash \mathbf{object}_i\ I\ \mathbf{isa}\ I_1,\ldots,I_n \succ p_0,k_0}{p,k \vdash \mathbf{abstract\ object}_i\ I\ \mathbf{isa}\ I_1,\ldots,I_n \succ p_0,k_0 + (abs = \{\mathbf{object}(i)\})}$$

[gf-s]
$$\frac{p,k \vdash \mathbf{object}_i\ I\ \mathbf{isa}\ I_1,\ldots,I_n \succ p_0,k_0 \quad \{\} = \{t_0 \mid (\mathbf{object}(i),t_0) \in con(k)\} \quad p,k \vdash R : t}{p,k \vdash \mathbf{object}_i\ I\ R\ \mathbf{isa}\ I_1,\ldots,I_n \succ \{(I,\ \mathbf{object}(i))\},k_0 + (con = \{(\mathbf{object}(i),t)\})}$$

[absf-s]
$$\frac{p,k \vdash \mathbf{object}_i\ I\ R\ \mathbf{isa}\ I_1,\ldots,I_n \succ p_0,k_0}{p,k \vdash \mathbf{abstract\ object}_i\ I\ R\ \mathbf{isa}\ I_1,\ldots,I_n \succ p_0,k_0 + (abs = \{\mathbf{object}(i)\})}$$

[has-s]
$$\frac{p,k \vdash I : c \quad (c,\mathbf{arrow}([t_1',\ldots,t_n'],t)) \in con(k) \quad p,k \vdash (c,[F_1,\ldots,F_n]) \succ [c_1,\ldots,c_n],[t_1,\ldots,t_n] \quad \forall\ 1 \le i \le n.\ I_i = name(F_i) \quad disjoint(I_1,\ldots,I_n) \quad p\&\{(I_1,\mathbf{unk}(t_1)),\ldots,(I_n,\mathbf{unk}(t_n)),k \vdash E : \mathbf{unk}(t)}{p,k \vdash I\ \mathbf{has\ method}(F_1,\ldots,F_n) : T\ \{E\} \succ \{\}, (gfms = \{(c,[c_1,\ldots,c_n])\}, lgf = \{(c,[c_1,\ldots,c_n])\})}$$

$$[\text{app-s}]\quad \frac{p,k \vdash E_0 : ot_0 \ldots p,k \vdash E_n : ot_n \quad ot_0 \notin abs(k) \ldots ot_n \notin abs(k) \quad k \vdash ot_0 \leq_{con} \mathbf{arrow}([t_1,\ldots,t_n],t) \quad k \vdash ot_1 \leq_{con} t_1 \ldots k \vdash ot_n \leq_{con} t_n}{p,k \vdash E_0(E_1,\ldots,E_n) : \mathbf{unk}(t)}$$

$$[\text{id-s}]\quad \frac{p(I) = c}{p,k \vdash I : c}$$

$$[\text{smp-s}]\quad \frac{p,k \vdash E : ot \quad ot \notin abs(k) \quad k \vdash ot \leq_{con} t}{p,k \vdash E : \mathbf{unk}(t)}$$

$$[\text{form-s}]\quad \frac{p,k \vdash (c,F_1,I) \succ c_1,t_1 \ldots p,k \vdash (c,F_n,n) \succ c_n,t_n}{p,k \vdash (c,[F_1,\ldots,F_n]) \succ [c_1,\ldots,c_n],[t_1,\ldots,t_n]}$$

$$[\text{spe-s}]\quad \frac{p,k \vdash I_2 : c_0}{p,k \vdash (c, I_1@I_2, i) \succ c_0,c_0}$$

$$[\text{uns-s}]\quad \frac{(c,\mathbf{arrow}([t_1,\ldots,t_n],t)) \in con(k)}{p,k \vdash (c, I, i) \succ \mathbf{object}(1),t_i}$$

$$[\text{look-s}]\quad \frac{[mh_1,\ldots,mh_n] = [mh \mid (c,mh) \in gfms(k) \wedge k \vdash [c_1,\ldots,c_m] \leq_{isa*} mh] \quad disjoint(mh_1,\ldots,mh_n) \quad 1 \leq i \leq n \quad k \vdash mh_i \leq_{isa*} mh_1 \ldots k \vdash mh_i \leq_{isa*} mh_n}{k \vdash c \; lookup\text{-}method \; [c_1,\ldots,c_m]}$$

$$[\text{art-s}]\quad \frac{p,k \vdash T_1 : t_1 \ldots p,k \vdash T_n : t_n \quad p,k \vdash T : t}{p,k \vdash (T_1,\ldots,T_n) \rightarrow T : \mathbf{arrow}([t_1,\ldots,t_n],t)}$$

$$[\text{isa*-s}]\quad \frac{k \vdash c_1 \leq_{isa} c_1' \ldots k \vdash c_n \leq_{isa} c_n'}{k \vdash [c_1,\ldots,c_n] \leq_{isa*} [c_1',\ldots,c_n']}$$

$$[\text{isa-s}]\quad k \vdash c \leq_{isa} \mathbf{object}(1)$$

$$[\text{isb-s}]\quad \frac{(c_1,c_2) \in isa(k)}{k \vdash c_1 \leq_{isa} c_2}$$

$$[\text{isr-s}]\quad k \vdash c \leq_{isa} c$$

$$[\text{ist-s}]\quad \frac{k \vdash c_1 \leq_{isa} c_2 \quad k \vdash c_2 \leq_{isa} c_3}{k \vdash c_1 \leq_{isa} c_3}$$

$$[\text{isar-s}]\quad \frac{k \vdash t_1' \leq_{isa} t_1 \ldots k \vdash t_n' \leq_{isa} t_n \quad k \vdash t \leq_{isa} t'}{k \vdash \mathbf{arrow}([t_1,\ldots,t_n],t) \leq_{isa} \mathbf{arrow}([t_1',\ldots,t_n'],t')}$$

$$[\text{cob-s}]\quad \frac{(c,t) \in con(k)}{k \vdash c \leq_{con} t}$$

$$[\text{coi-s}]\quad \frac{k \vdash c_1 \leq_{isa} c_2}{k \vdash c_1 \leq_{con} c_2}$$

$$[\text{cos-s}]\quad \frac{k \vdash c \leq_{con} t_1 \quad k \vdash t_1 \leq_{isa} t_2}{k \vdash c \leq_{con} t_2}$$

$$[\text{cou-s}]\quad \frac{k \vdash t_1 \leq_{isa} t_2}{k \vdash \mathbf{unk}(t_1) \leq_{con} t_2}$$

B.2 System G -- Extensions to the Base Static Semantics

Definitions and functions:

$$add\text{-}imp((con,isa,abs,gfms,lk)) = (con,isa,abs,\{\},(\{\},\{\}))$$

[mdsf-s]
$$\frac{k \vdash G\text{-}impl\text{-}side\text{-}typecheck\text{-}gfs\ ls(k)}{k,m \vdash program\text{-}has\text{-}safe\text{-}modules}$$

[Ggfs-s]
$$\frac{\begin{array}{c}\forall\ c \in \{c_1',...,c_r'\}.\ \forall\ art.\forall\ [c_1,...,c_n].\\ k \vdash c \leq_{con} art \wedge\\ [c_1,...,c_n]\ G\text{-}is\text{-}legal\text{-}tuple\text{-}for\ (c,art) \Rightarrow\\ c\ lookup\text{-}method\ [c_1,...,c_n]\end{array}}{k \vdash G\text{-}impl\text{-}side\text{-}typecheck\text{-}gfs\ \{c_1',...,c_r'\}}$$

[Gleg-s]
$$\frac{\forall\ 1 \leq i \leq n.\ c_i \notin abs(k) \wedge k \vdash c_i \leq_{con} t_i}{k \vdash [c_1,...,c_n]\ G\text{-}is\text{-}legal\text{-}arg\text{-}tuple\text{-}for\ (c, \mathbf{arrow}([t_1,...,t_n],t))}$$

[gfsf-s] $k \vdash module\text{-}has\text{-}safe\text{-}gfs$

[igfsf-s] $k \vdash module\text{-}has\text{-}safe\text{-}imported\text{-}gfs$

[obsf-s] $k \vdash module\text{-}has\text{-}safe\text{-}objects$

[mtsf-s] $k \vdash module\text{-}has\text{-}safe\text{-}methods$

B.3 System M -- Extensions to the Base Static Semantics

Definitions and functions:

$$add\text{-}imp((con,isa,abs,gfms,lk)) = (con,isa,abs,gfms,(\{\},\{\}))$$

[mdsf-s] $k,m \vdash program\text{-}has\text{-}safe\text{-}modules$

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

[Mgfs-s]
$$\frac{\begin{array}{c}\forall\ c \in \{c_1',...,c_r'\}.\ \forall\ art.\forall\ [c_1,...,c_n].\\ k \vdash c \leq_{con} art \wedge\\ [c_1,...,c_n]\ M\text{-}is\text{-}legal\text{-}tuple\text{-}for\ (c,art) \Rightarrow\\ c\ lookup\text{-}method\ [c_1,...,c_n]\end{array}}{k \vdash M\text{-}impl\text{-}side\text{-}typecheck\text{-}gfs\ \{c_1',...,c_r'\}}$$

[Mleg-s]
$$\frac{\begin{array}{c}\forall\ 1 \leq i \leq n.(k \vdash c_i \leq_{con} t_i \vee\\ (t_i = art \wedge c_i = \mathbf{object}(1)))\\ (c_1 \notin abs(k)\ \vee c_1 \notin ls(k))\\ c_1 = \mathbf{object}(1) \Rightarrow t_1\ is\text{-}non\text{-}local\end{array}}{k \vdash [c_1,...,c_n]\ M\text{-}is\text{-}legal\text{-}arg\text{-}tuple\text{-}for\ (c, \mathbf{arrow}([t_1,...,t_n],t))}$$

[igfsf-s]
$$\frac{\begin{array}{c}c^* = \{c \mid c \notin ls(k) \wedge (c,art) \in con(k)\}\\ k \vdash M\text{-}impl\text{-}side\text{-}typecheck\text{-}imported\text{-}gfs\ c^*\end{array}}{k \vdash module\text{-}has\text{-}safe\text{-}imported\text{-}gfs}$$

[Migfs-s]
$$\frac{\begin{array}{c}\forall\ c \in c^*.\forall\ art.\forall\ [c_1,...,c_n].\ k \vdash c \leq_{con} art \wedge\\ ([c_1,...,c_n]\ is\text{-}local\text{-}recheck1\text{-}for\ (c,art) \vee\\ [c_1,...,c_n]\ is\text{-}local\text{-}recheck2\text{-}for\ (c,art)) \Rightarrow\\ c\ lookup\text{-}method\ [c_1,...,c_n]\end{array}}{k \vdash M\text{-}impl\text{-}side\text{-}typecheck\text{-}imported\text{-}gfs\ c^*}$$

[re1-s]
$$\frac{\begin{array}{c}k \vdash [c_1,...,c_n]\ M\text{-}is\text{-}legal\text{-}arg\text{-}tuple\text{-}for\ (c, \mathbf{arrow}([t_1,...,t_n],t))\\ (c,mh) \in lgf(k)\\ [c_1,...,c_n] \leq_{isa^*} mh\end{array}}{k \vdash [c_1,...,c_n]\ is\text{-}local\text{-}recheck1\text{-}for\ (c, \mathbf{arrow}([t_1,...,t_n],t))}$$

[re2-s]
$$\frac{\begin{array}{c}k \vdash [c_1,...,c_n]\ M\text{-}is\text{-}legal\text{-}arg\text{-}tuple\text{-}for\ (c, \mathbf{arrow}([t_1,...,t_n],t))\\ c_1 \in ls(k) \qquad c_1\ is\text{-}orphan\end{array}}{k \vdash [c_1,...,c_n]\ is\text{-}local\text{-}recheck2\text{-}for\ (c, \mathbf{arrow}([t_1,...,t_n],t))}$$

[orph-s]
$$\frac{\begin{array}{c}t\ is\text{-}non\text{-}local\\ \{c\} = \{c_0 \mid c_0 \notin abs(k) \wedge k \vdash c_0 \leq_{con} t \wedge k \vdash c \leq_{isa} c_0\}\end{array}}{k \vdash c\ is\text{-}orphan}$$

$$\text{[obsf-s]}\quad \frac{k \vdash \textit{restriction-M2}}{k \vdash \textit{module-has-safe-objects}}$$

$$\text{[M2-s]}\quad \frac{\begin{array}{c}\forall\, c \in ls(k).(k \vdash c \le_{con} t_1 \wedge k \vdash c \le_{con} t_2 \\ \wedge\, k \vdash t_1 \textit{ is-non-local} \wedge k \vdash t_2 \textit{ is-non-local}) \Rightarrow \\ (k \vdash t_1 \le_{con} t_2 \vee k \vdash t_2 \le_{con} t_1)\end{array}}{k \vdash \textit{restriction-M2}}$$

$$\text{[mtsf-s]}\quad \frac{\begin{array}{c}c^* = \{c \mid (c,[c_1,\ldots,c_n]) \in lgf(k)\} \\ k \vdash \textit{restriction-M1 } c^*\end{array}}{k \vdash \textit{module-has-safe-methods}}$$

$$\text{[M1-s]}\quad \frac{\begin{array}{c}\forall\, c \in c^*.\ \forall [c_1,\ldots,c_n].\ (c,[c_1,\ldots,c_n]) \in lgf(k) \Rightarrow \\ c \in ls(k) \vee (n > 0 \wedge c_1 \in ls(k))\end{array}}{k \vdash \textit{restriction-M1 } c^*}$$

$$\text{[inlb-s]}\quad \frac{c \notin ls(k)}{k \vdash c \textit{ is-non-local}}$$

$$\text{[inla-s]}\quad \frac{0 \le i \le n \qquad t_i \textit{ is-non-local}}{k \vdash \textbf{arrow}([t_1,\ldots,t_n],t_0) \textit{ is-non-local}}$$

B.4 System E -- Modifications and Extensions to the Base Static Semantics

Domains for the static semantics:

$k \in \textit{TypeStore} = \textit{Conforms} \times \textit{Isa} \times \textit{Abstract} \times \textit{GFMethods} \times \textit{Specializers} \times \textit{LocalStore}$
$m \in \textit{Modules} = \textit{ModEnv} \times \textit{ModStore} \times \textit{Extends}$
$\textit{ext} \in \textit{Extends} = (I \times I)^*$
$\textit{sp} \in \textit{Specializers} = (\textit{Const} \times \textit{Spec}^*)^*$
$\textit{lk} \in \textit{LocalStore} = \textit{Locals} \times \textit{Extendees} \times \textit{LocalGFMethods}$
$\textit{ex} \in \textit{Extendees} = \textit{Const}^*$
$s \in \textit{Spec} = \mathbf{s} \mid \mathbf{u}$

Definitions and functions:

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

$\textit{add-imp}((con,isa,abs,gfms,sp,lk)) = (con,isa,abs,gfms,sp,(\{\},\{\},\{\}))$
$\textit{add-ext}((con,isa,abs,gfms,sp,(ls,ex,lgf))) = (con,isa,abs,gfms,sp,(\{\},ls \cup ex,\{\}))$
$\textit{type}(T) = T \quad \textit{type}(\#T) = T$
$\textit{spec}(T) = \mathbf{u} \quad \textit{spec}(\#T) = \mathbf{s}$

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

$$\text{[mod-s]}\quad \frac{\begin{array}{c}p = mp(m)(I_1)\&\ldots\&mp(m)(I_t)\& \\ mp(m)(I_1')\&\ldots\&mp(m)(I_r') \\ k = \textit{add-imp}(mk(m)(I_1))\&\ldots\& \\ \textit{add-imp}(mk(m)(I_t))\&\textit{add-ext}(mk(m)(I_1'))\&\ldots\& \\ \textit{add-ext}(mk(m)(I_r')) \\ p,k \vdash D_1 \ldots D_n \succ p_0,k_0 \\ k{+}k_0 \vdash \textit{module-has-safe-gfs} \\ k{+}k_0 \vdash \textit{module-has-safe-imported-gfs} \\ k{+}k_0 \vdash \textit{module-has-safe-objects} \\ k{+}k_0 \vdash \textit{module-has-safe-methods}\end{array}}{\begin{array}{c}m \vdash \textbf{module}\ I\ \textbf{imports}\ I_1,\ldots,I_t \\ \textbf{extends}\ I_1',\ldots,I_r'\ \{D_1 \ldots D_n\} \\ \succ (\{(I,p\&p_0)\},\{(I,k{+}k_0)\},\{(I,I_1'),\ldots,(I,I_r')\})\end{array}}$$

$$\text{[gf-s]}\quad \frac{\begin{array}{c}p,k \vdash \textbf{object}_i\ I\ \textbf{isa}\ I_1,\ldots,I_n \succ p_0,k_0 \\ p,k \vdash R : t \succ sp_1 \\ k_1 = (con = \{(\textbf{object}(i),t)\},\ sp = \{(\textbf{object}(i),sp_1)\})\end{array}}{\begin{array}{c}p,k \vdash \textbf{object}_i\ I\ R\ \textbf{isa}\ I_1,\ldots,I_n \\ \succ \{(I,\ \textbf{object}(i))\},k_0 + k_1\end{array}}$$

[rart-s] $$\frac{T_1 = type(S_1) \ldots T_n = type(S_n) \quad s_1 = spec(S_1) \ldots s_n = spec(S_n) \quad p,k \vdash (T_1,\ldots,T_n)\to T : t}{p,k \vdash (S_1,\ldots,S_n)\to T : t \succ [s_1,\ldots,s_n]}$$

[mdsf-s] $$\frac{m\vdash \textit{each-module-has-a-most-extending-module}}{k,m\vdash \textit{program-has-safe-modules}}$$

[me-s] $$\frac{\forall\, I \in \textit{domain}(mp(m)).\ \exists\, I_1.\forall\, I_2.\ (m \vdash I_2 \le_{ext} I \Rightarrow m \vdash I_1 \le_{ext} I_2)}{m\vdash \textit{each-module-has-a-most-extending-module}}$$

[exb-s] $$\frac{(I_1,I_2) \in ext(m)}{m \vdash I_1 \le_{ext} I_2}$$

[exr-s] $$m \vdash I \le_{ext} I$$

[ext-s] $$\frac{m \vdash I_1 \le_{ext} I_2 \quad m \vdash I_2 \le_{ext} I_3}{m \vdash I_1 \le_{ext} I_3}$$

[obsf-s] $$\frac{k\vdash \textit{restriction-E2}}{k\vdash \textit{module-has-safe-objects}}$$

[E2-s] $$\frac{\forall\, c \in ls(k).(k \vdash c \le_{con} t \wedge k \vdash t\ \textit{is-imported}) \Rightarrow ((k \vdash c \le_{con} t_1 \wedge k \vdash c \le_{con} t_2 \wedge k \vdash t_1\ \textit{is-non-local} \wedge k \vdash t_2\ \textit{is-non-local}) \Rightarrow (k \vdash t_1 \le_{con} t_2 \vee k \vdash t_2 \le_{con} t_1))}{k\vdash \textit{restriction E2}}$$

[mtsf-s] $$\frac{c^* = \{c \mid (c,[c_1,\ldots,c_n]) \in lgf(k)\} \quad k\vdash \textit{restriction-E1a}\ c^* \quad k\vdash \textit{restriction-E1b}\ c^*}{k\vdash \textit{module-has-safe-methods}}$$

[E1a-s] $$\frac{\forall\, c \in c^*.\ \forall [c_1,\ldots,c_n].\ (c,[c_1,\ldots,c_n]) \in lgf(k) \Rightarrow (k \vdash c \in ls(k) \cup ex(k)) \vee ((c,[s_1,\ldots,s_n]) \in sp(k) \wedge \exists\, 1 \le i \le n.\ s_i = \mathbf{s} \wedge \forall 1 \le i \le n.\ (s_i = \mathbf{s}) \Rightarrow c_i \in ls(k))}{k\vdash \textit{restriction-E1a}\ c^*}$$

[E1b-s] $$\frac{\forall\, c \in c^*.\ \forall [c_1,\ldots,c_n].\ (c,[c_1,\ldots,c_n]) \in lgf(k) \Rightarrow (c,[s_1,\ldots,s_n]) \in sp(k) \wedge \forall\ 1 \le i \le n.(s_i = \mathbf{u}) \Rightarrow (c_i = \mathbf{object}(1))}{k\vdash \textit{restriction-E1b}\ c^*}$$

[gfsf-s] $$\frac{k\vdash \textit{E-impl-side-typecheck-gfs}\ ls(k) \cup ex(k) \quad k\vdash \textit{restriction-E3a}\ ls(k)}{k\vdash \textit{module-has-safe-gfs}}$$

[Egfs-s] $$\frac{\forall\, c \in \{c_1',\ldots,c_r'\}.\ \forall\, art.\forall\, [c_1,\ldots,c_n].\ (k \vdash c \le_{con} art \wedge ([c_1,\ldots,c_n]\ \textit{E-is-legal-tuple-for}\ c\ \textit{via}\ art) \Rightarrow c\ \textit{lookup-method}\ [c_1,\ldots,c_n])}{k\vdash \textit{E-impl-side-typecheck-gfs}\ \{c_1',\ldots,c_r'\}}$$

[Eleg-s] $$\frac{(c,[s_1,\ldots,s_n]) \in sp(k) \quad \forall\ 1 \le i \le n.(s_i = \mathbf{s} \wedge (k \vdash c_i \le_{con} t_i \vee (t_i = art \wedge c_i = \mathbf{object}(1)))) \vee (s_i = \mathbf{u} \wedge c_i = \mathbf{object}(1)) \quad \forall\ 1 \le q,r \le n.((s_q = s_r = \mathbf{s}) \Rightarrow (q = r)) \Rightarrow (c_q \notin abs(k) \vee c_q \notin ls(k) \wedge (c_1 = \mathbf{object}(1)) \Rightarrow t_1\ \textit{is-non-local}))}{k \vdash [c_1,\ldots,c_n]\ \textit{E-is-legal-arg-tuple-for}\ (c,\mathbf{arrow}([t_1,\ldots,t_n],t))}$$

[E3a-s] $$\frac{\forall\, c \in c^*.(c,[s_1,\ldots,s_n]) \in sp(k) \wedge (c,\mathbf{arrow}([t_1,\ldots,t_n],t)) \in con(k) \Rightarrow \forall 1 \le i \le n.(t_i\ \textit{is-imported} \Rightarrow s_i = \mathbf{u})}{k\vdash \textit{restriction-E3a}\ c^*}$$

[igfsf-s] $$\frac{c^* = \{c \mid c \notin ls(k) \cup ex(k) \wedge (c,art) \in con(k)\} \quad k\vdash \textit{E-impl-side-typecheck-imported-gfs}\ c^*}{k\vdash \textit{module-has-safe-imported-gfs}}$$

[Eigfs-s] $$\frac{\forall\, c \in c^*.\forall\, art.\forall\, [c_1,\ldots,c_n].\ k \vdash c \le_{con} art \wedge ([c_1,\ldots,c_n]\ \textit{is-local-recheck1-for}\ (c,art) \vee [c_1,\ldots,c_n]\ \textit{is-local-recheck2-for}\ (c,art)) \Rightarrow c\ \textit{lookup-method}\ [c_1,\ldots,c_n]}{k\vdash \textit{E-impl-side-typecheck-imported-gfs}\ c^*}$$

[re1-s] $$\frac{k \vdash [c_1,\ldots,c_n]\ \textit{E-is-legal-arg-tuple-for}\ (c,\mathbf{arrow}([t_1,\ldots,t_n],t)) \quad (c,mh) \in lgf(k) \quad [c_1,\ldots,c_n] \le_{inh^*} mh}{k \vdash [c_1,\ldots,c_n]\ \textit{is-local-recheck1-for}\ (c,\mathbf{arrow}([t_1,\ldots,t_n],t))}$$

[re2-s] $$\frac{k \vdash [c_1,\ldots,c_n]\ \textit{E-is-legal-arg-tuple-for}\ (c,\mathbf{arrow}([t_1,\ldots,t_n],t)) \quad 1 \le i \le n \quad c_i \in ls(k) \quad c_i\ \textit{is-orphan-of-type}\ t_i}{k \vdash [c_1,\ldots,c_n]\ \textit{is-local-recheck2-for}\ (c,\mathbf{arrow}([t_1,\ldots,t_n],t))}$$

[orph-s] $$\frac{t\ \textit{is-imported} \quad \{c\} = \{c_0 \mid c_0 \notin abs(k) \wedge k \vdash c_0 \le_{con} t \wedge k \vdash c \le_{isa} c_0\}}{k\vdash c\ \textit{is-orphan}}$$

[iimb-s] $$\frac{c \notin ls(k) \cup ex(k)}{k \vdash c \text{ is-imported}}$$

[iima-s] $$\frac{0 \le i \le n \qquad t_i \text{ is-imported}}{k \vdash \mathbf{arrow}([t_1,...,t_n],t_0) \text{ is-imported}}$$

[inlb-s] $$\frac{c \notin ls(k)}{k \vdash c \text{ is-non-local}}$$

[inla-s] $$\frac{0 \le i \le n \qquad t_i \text{ is-non-local}}{k \vdash \mathbf{arrow}([t_1,...,t_n],t_0) \text{ is-non-local}}$$

B.5 System ME -- Modifications and Extensions to the Static Semantics of System E

[gfsf-s] $$\frac{\begin{array}{c} c_1^* = \{c \mid c \in ls(k) \wedge k \vdash c \text{ is-M-gf}\} \\ c_2^* = \{c \mid c \in ls(k) \cup ex(k) \wedge k \vdash c \text{ is-E-gf}\} \\ k \vdash \text{M-impl-side-typecheck-gfs } c_1^* \\ k \vdash \text{E-impl-side-typecheck-gfs } c_2^* \\ k \vdash \text{restriction-E3a } c_2^* \end{array}}{k \vdash \text{module-has-safe-gfs}}$$

[igfsf-s] $$\frac{\begin{array}{c} c_1^* = \{c \mid c \notin ls(k) \wedge (c,art) \in con(k)\} \\ c_2^* = \{c \mid c \notin ls(k) \cup ex(k) \wedge (c,art) \in con(k)\} \\ c_3^* = \{c \mid c \in c_1^* \wedge k \vdash c \text{ is-M-gf}\} \\ c_4^* = \{c \mid c \in c_2^* \wedge k \vdash c \text{ is-E-gf}\} \\ k \vdash \text{M-impl-side-typecheck-imported-gfs } c_3^* \\ k \vdash \text{E-impl-side-typecheck-imported-gfs } c_4^* \end{array}}{k \vdash \text{module-has-safe-imported-gfs}}$$

[obsf-s] $$\frac{k \vdash \text{restriction-M2}}{k \vdash \text{module-has-safe-objects}}$$

[mtsf-s] $$\frac{\begin{array}{c} c_1^* = \{c \mid k \vdash c \text{ is-M-gf}\} \\ c_2^* = \{c \mid k \vdash c \text{ is-E-gf}\} \\ k \vdash \text{restriction-M1 } c_1^* \\ k \vdash \text{restriction-E1a } c_2^* \\ k \vdash \text{restriction-E1b } c_2^* \end{array}}{k \vdash \text{module-has-safe-methods}}$$

[isM-s] $$\frac{(c,[s_1,...,s_n]) \in sp(k) \wedge \forall\ 1 \le i \le n.(s_i = \mathbf{u})}{k \vdash c \text{ is-M-gf}}$$

[isE-s] $$\frac{(c,[s_1,...,s_n]) \in sp(k) \quad 1 \le i \le n \quad (s_i = \mathbf{s})}{k \vdash c \text{ is-E-gf}}$$

Appendix C Sketch of the Type Soundness Approach for Dubious

This section sketches our proof that Dubious's static semantics is sound with respect to its dynamic semantics. We have proofs for Systems G and M and are working on proofs for Systems E and ME. For more details, see http://www.cs.washington.edu/homes/todd/dubious.txt.

The type soundness results are presented in terms of the dynamic and static semantics in appendices A and B, respectively. In the dynamic semantics, $\vdash M^* \succ s,me$ denotes the evaluation of a set of modules M^*, producing a store s and module environment me. Also, $e,s \vdash E \Rightarrow o$ denotes the evaluation of an expression E to an object o in the context of environment e and store s. Finally, $me(I)$ returns the environment associated with module I in module environment me. In the static semantics, $\vdash M^* \succ k,m$ denotes the typechecking of a set of modules M^*, producing a type store k and module context m. Also, $p,k \vdash E : ot$ denotes the typechecking of an expression E to have type ot in the context of type environment p and type store k. Further, $mp(m)(I)$ returns the type environment associated with module I in module environment m.

Our proof method is based on prior work on type soundness by Wright and Felleisen [Wright & Felleisen 94]. First we prove a subject reduction lemma:

Lemma 1. (Subject Reduction) Suppose $\vdash M^* \succ k,m$ and $\vdash M^* \succ s,me$. Let $p = mp(m)(I)$ and $e = me(I)$. If $p,k \vdash E : ot$ and $e,s \vdash E \Rightarrow o$ then $p,k \vdash o : ot$.

Proof: We first show two simple correspondences, one between p and e and the other between k and s. The result then follows by induction on the length of the derivation in the dynamic semantics that $e,s \vdash E \Rightarrow o$.

This 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 (see appendix A.1 for the definitions of the *GFMethod* domain and the & function, used in condition 4):

Definition 1. (Faulty Expressions) An expression E is *faulty with respect to environment e and store s* if one of the following conditions holds:

1. $E = I$ *and* $I \notin domain(e)$
2. $E = E_0(E_1,...,E_n)$ *and* $\exists i.(0 \le i \le n$ *and* E_i *is faulty with respect to e and s*)
3. $E = E_0(E_1,...,E_n)$, $\forall i.(0 \le i \le n \Rightarrow e,s \vdash E_i \Rightarrow o_i)$, *and there is no most-specific applicable method for* $o_0(o_1,...,o_n)$ *in s.*
4. $E = E_0(E_1,...,E_n)$, $\forall i.(0 \le i \le n \Rightarrow e,s \vdash E_i \Rightarrow o_i)$, $((o_0,mh),mb)$ *is the most-specific applicable method for* $o_0(o_1,...,o_n)$ *in s, where* $mb = ([I_1,...,I_n],E',e')$, *and* E' *is faulty with respect to* $e' \& \{(I_1,o_1),...,(I_n,o_n)\}$ *and s.*

This definition of faulty expressions 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 $\vdash M_1,...,M_n \succ s,m$ and let $e = me(I)$. If E is not faulty with respect to e and s and $e,s \vdash E$ does not diverge, then $e,s \vdash E \Rightarrow o$.
Proof: By induction on the length of the derivation in the dynamic semantics of $e,s \vdash E \Rightarrow o$.

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 $\vdash M^* \succ k,m$ and $\vdash M^* \succ s,me$. Let $p = mp(m)(I)$ and $e = me(I)$. If $p,k \vdash E : ot$ then E is not faulty with respect to e and s.
Proof: We need to prove that none of the four cases in the definition of faulty expressions holds. The correspondence between p and e referred to above is sufficient to prove that the typechecks on identifiers rule out case one. We rule out cases two and four by induction. Ruling out case three is the main technical challenge in the proof. We prove that every legal message send has a most-specific applicable method in k. This case is completed by showing that the correspondence between k and s referred to above is sufficient to prove that method specificity in k translates into method specificity in s.

Finally, all three lemmas are combined to prove the main result (see appendices A.1 and B.1 for the definitions of the [prg-d] rule for evaluating programs and the [prg-s] rule for typechecking programs, respectively):

Theorem 1. (Type Soundness) Let $P = M^*$ **import** I **in** E **end**. If $\vdash P : ot \succ k$ and $\vdash P$ does not diverge in the dynamic semantics, then $\vdash P \Rightarrow o \succ s$ and $p,k \vdash o : ot$, where $p = mp(m)(I)$.
Proof: Since $\vdash P : ot \succ k$, by the [prg-s] rule in the static semantics we have $\vdash M^* \succ k,m$ and $p,k \vdash E : ot$. After showing a correspondence between the static and dynamic environments, it is easy to show that $\vdash M^* \succ k,m$ in the static semantics implies $\vdash M^* \succ s,me$ in the dynamic semantics, for some store s in the appropriate correspondence with k. Let $e = me(I)$. 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 $e,s \vdash E$. Therefore by lemma 2, $e,s \vdash E \Rightarrow o$. By the [prg-d] rule again, $\vdash P \Rightarrow o \succ s$. Finally by lemma 1, $p,k \vdash o : ot$.