# Modular Statically Typed Multimethods

Todd Millstein and Craig Chambers

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

Technical Report UW-CSE-99-03-02
March 1999

This technical report presents the formal details of the Dubious language and its various static type systems, as described in [Millstein & Chambers 99]. Section 1 presents the syntax of Dubious, sections 2 and 3 present its formal dynamic and static semantics, respectively, and section 4 sketches the soundness proofs for each of Dubious's type systems. Refer to [Millstein & Chambers 99] for an informal description of the semantics of the language and its static type restrictions.

$P$ ::= $M_1$ ... $M_n$ **import** $I$ **in** $E$ **end**
$M$ ::= **module** $I$ **imports** $I_1, ..., I_m$ { $D_1$ ... $D_n$ }
$D$ ::= [**abstract** | **interface**] **object** $I$ **isa** $O_1, ..., O_n$
| $I$ **has method**($F_1, ..., F_n$) { $E$ }
$E$ ::= $E_0(E_1, ..., E_n)$ | $I$
$O$ ::= $I$ | $(O_1, ..., O_n) \rightarrow O$
$F$ ::= $I_1$ [@ $I_2$]
$I$ ::= *identifier*

**Figure 1-1: Syntax of Dubious**

# 1 Syntax

Dubious's syntax appears in figure 1-1. This syntax is used in conjunction with the static type rules corresponding to System M. The static type restrictions of Systems E and ME require a few modifications and additions to the syntax, as follows:

$M$ ::= **module** $I$ **imports** $I_1, ..., I_m$ **extends** $I_1', ..., I_r'$ { $D_1$ ... $D_n$ }
$O$ ::= $I$ | $(S_1, ..., S_n) \rightarrow O$
$S$ ::= [#]$O$

# 2 Dynamic Semantics

This section presents the dynamic semantics of Dubious. Subsection 2.1 presents the dynamic semantics corresponding to the syntax in figure 1-1, and subsection 2.2 presents the modifications to the dynamic semantics for the augmented syntax necessary for Systems E and ME.

## 2.1 Dynamic Semantics for Systems G and M

### 2.1.1 Preliminaries

Figure 2-1 defines the necessary domains for the dynamic semantics. *Env* represents the dynamic environment, mapping identifiers to values. *Store* maintains information on the inheritance relation among objects and the methods contained in each generic function. *ModEnv* is a mapping from each module name to an environment representing the values defined in that module. *Isa* is the domain representing the declared inheritance relationship. The *GFMethods* domain contains the

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

**Figure 2-1: Domains for the dynamic semantics**

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

$r_1 \,\&\, r_2 = \{(x,y_1) \mid (x,y_1) \in r_2 \text{ or } [(x,y_1) \in r_1 \text{ and } \sim\exists\, y_2.(x,y_2) \in r_2]\}$

$(isa_1, gfms_1) + (isa_2, gfms_2) = (isa_1 \cup isa_2, gfms_1 \cup gfms_2)$

$name(I_1@I_2) = I_1 \qquad name(I_1) = I_1$

$disjoint(x_1,...,x_n) = \forall\ 1 \le i \le n.\ \forall\ 1 \le j \le n.(x_i = x_j) \Rightarrow (i = j)$

Each static occurrence of "object" in the program is subscripted with a unique integer greater than one.

**Figure 2-2: Definitions and functions**

relevant information about each method in a generic function object. *MethodHeader* is the list of specializer objects of a method. *MethodBody* is the list of formal parameter names, method body, and lexical environment of a method. *Obj* is the domain of objects in the program. *Arrow* is the domain of arrow objects, containing a list of argument objects and a result object. *Val* is the domain of non-arrow objects. Each element of this domain is given a different natural number, which serves as the object's unique identity.

Figure 2-2 makes several useful definitions and functions. Many of these definitions are self-explanatory. + is the pointwise union of two *Store*s. & is a *shadowing union* operator, favoring $r_2$. The *disjoint* function returns true if and only if all of its arguments are distinct. Since each textual occurrence of "object" maps to a unique run-time value, the subscripts are a simple way to provide each run-time value with a unique identity.

### 2.1.2 Judgements

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

Figure 2-3 contains the judgements for programs and modules. The program rule evaluates each module in the program, returning a global store and an environment for each module. The rule then

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

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

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

**Figure 2-3: Judgements for programs and modules**

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

$$\text{[obj-d]}\quad \frac{\begin{array}{c} e \vdash O_1 \Rightarrow o_1 \quad \ldots \quad e \vdash O_n \Rightarrow o_n \\ v = \mathbf{obj}(i) \end{array}}{\begin{array}{c} e,s \vdash \texttt{[abstract|interface] object}_i\ I\ \texttt{isa}\ O_1,\ldots,O_n \\ \succ \{(I,v)\},(\{(v,o_1),\ldots,(v,o_n)\},\{\}) \end{array}}$$

$$\text{[has-d]}\quad \frac{\begin{array}{c} e \vdash I \Rightarrow v \\ e,s \vdash (v,[F_1,\ldots,F_n]) \Rightarrow (o_1,\ldots,o_n) \\ mb = ([name(F_1),\ldots,name(F_n)],E,\ e) \end{array}}{e,s \vdash I\ \texttt{has method}(F_1,\ldots,F_n)\ \{E\} \succ \{\},(\{\},\{((v,[o_1,\ldots,o_n]),mb)\})}$$

**Figure 2-4: Judgements for declarations**

evaluates the given expression in the context of the global store and the environment of the imported module. A module is evaluated in the context of the shadowing union of the environments of each module it imports.

The judgements for declarations are shown in figures 2-4. Each declaration in a declaration block is evaluated in an environment which includes any name bindings from previous declarations in the block. Although declaration blocks are not recursive, it is still possible to write (mutually) recursive methods because generic functions are declared separately from their methods. For example, the body of a method *m* may refer to the same generic function in which *m* is contained, since the generic function was declared previously, thereby creating a recursive method. The **`object`** declaration is evaluated by evaluating each inheritance parent of the new object and storing the new inheritance pairs in the resulting evaluation context. Methods are evaluated by evaluating each formal argument and recording the appropriate information in the resulting evaluation context.

Figure 2-5 contains the judgements for expressions and objects. A generic function application is evaluated by evaluating the generic function expression and the actual argument expressions to objects. The most-specific method for the argument objects is extracted from the generic function object, and its method body is then evaluated in the context of the method's lexical environment, augmented with bindings from the formal to the actual parameters. An identifier is evaluated simply by looking up the identifier's binding in the current environment. An arrow object is evaluated by evaluating each of its object components.

The judgements for evaluating formals are given in figure 2-6. If the formal parameter is specialized, the specializer object is evaluated and returned. If the formal parameter is unspecialized, we consider the formal to be implicitly specialized on the associated object in an arrow object of the generic function (the static semantics will ensure that there is at most one such arrow object for each generic function).

$$\text{[app-d]}\quad \frac{e,s \vdash E_0 \Rightarrow v_0 \quad \ldots \quad e,s \vdash E_n \Rightarrow v_n \qquad s \vdash v_0 \textit{ lookup-method } [v_1,\ldots,v_n] \succ (mh,([I_1,\ldots,I_n],E,e_0)) \qquad e_0 \,\&\, \{(I_1,v_1),\ldots,(I_n,v_n)\},s \vdash E \Rightarrow v}{e,s \vdash E_0(E_1,\ldots,E_n) \Rightarrow v}$$

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

$$\text{[arr-d]}\quad \frac{e \vdash O_1 \Rightarrow o_1 \quad \ldots \quad e \vdash O_n \Rightarrow o_n \qquad e \vdash O \Rightarrow o}{e \vdash (O_1,\ldots,O_n) \rightarrow O \Rightarrow \textbf{arrow}([o_1,\ldots,o_n],o)}$$

**Figure 2-5: Judgements for expressions and objects**

$$\text{[form-d]}\quad \frac{e,s \vdash (o,F_1,1) \succ o_1 \quad \ldots \quad e,s \vdash (c,F_n,n) \succ o_n}{e,s \vdash (o,[F_1,\ldots,F_n]) \succ [o_1,\ldots,o_n]}$$

$$\text{[spe-d]}\quad \frac{e \vdash I_2 \Rightarrow o_2}{e,s \vdash (o,\, I_1@I_2,\, i) \Rightarrow o_2}$$

$$\text{[uns-d]}\quad \frac{1 \le i \le n \qquad (o,\textbf{arrow}([o_1,\ldots,o_n],o_0)) \in isa(s)}{e,s \vdash (o,\, I,\, i) \succ o_i}$$

**Figure 2-6: Judgements for formal arguments**

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

**Figure 2-7: Judgement for method lookup**

The method lookup rule appears in figure 2-7. The rule first extracts the tuple of specializers of all *applicable* methods for $[o_1,\ldots,o_m]$ from the generic function $o$. Then it finds the unique such tuple of specializers that is more-specific than all other tuples of specializers, and it returns this tuple along with the associated formal arguments, method body, and lexical environment.

The rules for extending the direct inheritance relation to form the descendant relation are given in figure 2-8. The $\le_{isa*}$ relation is a pointwise extension of the $\le_{isa}$ relation. The $\le_{isa}$ relation is the reflexive, transitive closure of the declared inheritance relation, along with the standard contravariant rule for relating arrow objects.

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

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

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

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

[isar-s] $$\frac{\begin{array}{c} s \vdash o_1' \leq_{isa} o_1 \ldots s \vdash o_n' \leq_{isa} o_n \\ k \vdash o \leq_{isa} o' \end{array}}{s \vdash \mathbf{arrow}([o_1,\ldots,o_n],o) \leq_{isa} \mathbf{arrow}([o_1',\ldots,o_n'],o')}$$

**Figure 2-8: Judgements for the descendant relation**

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

[arr-d] $$\frac{e \vdash O_1 : o_1 \ldots e \vdash O_n : o_n \qquad e \vdash O : o}{e \vdash ([\#]O_1,\ldots,[\#]O_n) \rightarrow O : \mathbf{arrow}([o_1,\ldots,o_n],o)}$$

**Figure 2-9: Judgement modifications for Systems E and ME**

## 2.2 Modifications for Systems E and ME

Two minor modifications of these semantics are needed to accomodate the syntax extensions for Systems E and ME. These modified rules are shown in figure 2-9. The modified rule for modules treats extended modules identically to imported modules, using their environments as context for the evaluation of the new module. The rule for evaluating arrow objects simply ignores the optional # markers on argument objects.

# 3 Static Semantics

This section presents the static semantics of Dubious. Subsection 3.1 presents the base static semantics, which is independent of the particular modular typechecking system used. Subsections 3.2 through 3.5 present the modifications and additions to the base static semantics for Systems G, M, E, and ME, respectively.

$$
\begin{array}{llll}
p & \in & \mathit{TypeEnv} & = & I \to \mathit{ObjType} \\
k & \in & \mathit{TypeStore} & = & \mathit{Isa} \times \mathit{Concrete} \times \mathit{Abstract} \times \mathit{GFMethodTypes} \\
 & & & & \times \mathit{LocalTypeStore} \\
m & \in & \mathit{Modules} & = & \mathit{ModTypeEnv} \times \mathit{ModTypeStore} \\
\mathit{cnc} & \in & \mathit{Concrete} & = & \mathit{Const}* \\
\mathit{abs} & \in & \mathit{Abstract} & = & \mathit{Const}* \\
\mathit{isa} & \in & \mathit{Isa} & = & (\mathit{Const} \times \mathit{ConstType})* \\
\mathit{gfms} & \in & \mathit{GFMethodTypes} & = & (\mathit{Const} \times \mathit{MethodHeaderType})* \\
\mathit{mh} & \in & \mathit{MethodHeaderType} & = & \mathit{ConstType}* \\
\mathit{lk} & \in & \mathit{LocalTypeStore} & = & \mathit{Locals} \times \mathit{LocalGFMethods} \\
\mathit{ls} & \in & \mathit{Locals} & = & \mathit{Const}* \\
\mathit{lgf} & \in & \mathit{LocalGFMethods} & = & \mathit{GFMethodTypes} \\
\mathit{mp} & \in & \mathit{ModTypeEnv} & = & I \to \mathit{TypeEnv} \\
\mathit{mk} & \in & \mathit{ModTypeStore} & = & I \to \mathit{TypeStore} \\
t & \in & \mathit{ConstType} & = & \mathit{Const} + \mathit{ArrowType} \\
\mathit{art} & \in & \mathit{ArrowType} & = & \textbf{arrow}(\mathit{ConstType}*, \mathit{ConstType}) \\
\mathit{ot} & \in & \mathit{ObjType} & = & \mathit{Const} + \textbf{unk}(\mathit{ConstType}) \\
c & \in & \mathit{Const} & = & \textbf{object}(\mathit{Nat})
\end{array}
$$

**Figure 3-1: Domains for the static semantics**

## 3.1 Base Static Semantics

### 3.1.1 Preliminaries

Figure 3-1 defines the necessary domains for the static semantics. *TypeEnv* represents the static environment, mapping object identifiers to their types. *TypeStore* maintains information on the inheritance relation among objects, which objects are concrete, which objects are abstract, the methods contained in each generic function, and local information about the current module being typechecked. *Modules* maintains a mapping from each module name to its associated type environment, and a mapping from each module name to its associated type store. The *GFMethodTypes* domain maintains, for each method in the program, a mapping from the method's generic function to the method's tuple of specializers. *Locals* keeps track of all non-arrow objects created in the current module being typechecked, and *LocalGFMethods* keeps track of all methods created in the current module being typechecked. A *ConstType* is the type of statically known objects. This is either a non-arrow object with a known object identity, or an arrow object. *ObjType* is the type of all values in the program. A value may either have a *Const* type, in which case the value has a statically known object identity, or a *unknown* type of the form **unk**(*Type*), which means that the value is not statically known but is known to inherit from *Type*.

Figure 3-2 makes several useful definitions and functions. As in the dynamic semantics, we define accessor functions for easy manipulation of the *TypeStore* and *Modules* domains. In addition, we define a record-like syntax for representing sparse *TypeStore*s, where most of the components are empty. The *add-imp* function is used to obtain "interface" information from the type context of an imported module. This information is then added to the type context of the importer. As in the dynamic semantics, integer subscripts on occurrences of "object" are used as object identities. We assume that the subscripts used in the dynamic and static semantics are identical.

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

Let () denote the empty type store and $(x_1 = v_1,...,x_n = v_n)$ denote the empty type store augmented with component $x_i$ equal to $v_i$.

$r_1 \,\&\, r_2 = \{(x,y_1) \mid (x,y_1) \in r_2 \text{ or } [(x,y_1) \in r_1 \text{ and } \sim\exists\, y_2.(x,y_2) \in r_2]\}$

$(s_{11},...,s_{1n}) + (s_{21},...,s_{2n}) = (s_{11} \cup s_{21}, \ldots, s_{1n} \cup s_{2n})$

*add-imp*((*isa*,*cnc*,*abs*,*gfms*,*lk*)) = (*isa*,*cnc*,*abs*,*gfms*,({},{}))

*name*(**module** *I* **imports** $I_1,...I_t$ {*D**}) = *I*

$name(I_1@I_2) = I_1 \qquad name(I) = I$

$disjoint(x_1,...,x_n) = \forall\ 1 \le i \le n.\ \forall\ 1 \le j \le n.(x_i = x_j) \Rightarrow (i = j)$

$length([x_1,...,x_n]) = n$

Each static occurrence of "object" in the program is subscripted with a unique integer greater than one.

**Figure 3-2: Definitions and functions**

### 3.1.2 Judgements

Now we present the judgements for the base static semantics. In general, a judgement may have the form $d_1,...,d_m \vdash X : t \succ d_1',...,d_n'$. Such a judgement is interpreted as follows: "Given the information in domain elements $d_1,...,d_m$ as the context for typechecking, the program fragment *X* has type *t* and produces $d_1',...,d_n'$ as additions to the current type context." The : *t* or $\succ d_1',...,d_n'$ parts may be omitted.

Figure 3-3 contains the judgements for programs and modules. The program rule typechecks each module in the program, returning a global type store and a type environment and type store for each module. The rule then typechecks the given expression in the context of the global type store and the type environment of the imported module. A block of modules is typechecked one-by-one, and the typechecking of each module returns that module's type environment and store. The type store of the program is the union of the type stores of each module. The *program-has-safe-modules* rule will be supplied by the particular type system (G, M, E, or ME) being added to these base semantics, performing any necessary global typechecking. The declarations in a module are typechecked in the context of the type environments and type stores of each imported module. The four *module-has-safe-x* rules will be supplied by each particular type system, performing any necessary local and regional typechecking.

The judgements for declarations are shown in figure 3-4. Each declaration in a declaration block is typechecked in the context of the name bindings and type store produced from typechecking all preceding declarations in the block. The [obj-s] rule typechecks objects that are not acting as generic functions. In particular, the rule checks that none of the object's parents (either declared or inherited) are arrow objects. The rule simply evaluates each declared inheritance parent and stores these parents in the *Isa* component of the type context. It also adds the new object to *Locals*, indicating that the object was created in the current module. The [gf-s] rule typechecks generic functions. It is similar to the [obj-s] rule, except that it ensures that the new generic function has exactly one most-specific parent arrow object. The [abso-s] and [cnco-s] rules are used as wrappers around the previous two rules, in order to correctly note in the type context if the new object is

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

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

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

**Figure 3-3: Judgements for programs and modules**

abstract or concrete (if the new object is an interface, nothing is added to the type context). Methods are typechecked by typechecking each formal argument position, returning an associated specializer type. Then the method body is typechecked in the context of the current lexical type environment, augmented with type bindings for the formals. The body must have the declared result type of the method's associated generic function.

Figure 3-5 contains the judgements for expressions and types. The [app-s] rule is the client-side typechecking rule for generic function applications. The result of a generic function application cannot be statically known, so the result type is always of the form **unk**(*t*). An identifier is typechecked simply by looking up the identifier's binding in the current type environment. There are two subsumption rules, which allow the types of expressions to be "raised." The [smpc-s] rule ensures that only concrete objects can have their types raised, which has the effect of disallowing reference to non-concrete objects in expressions. An arrow object is typechecked by typechecking each of its components.

The judgements for typechecking formals are given in figure 3-6. If the formal parameter is specialized, the specializer object is typechecked and returned. The specializer must not be an interface and it must be a descendant of the associated argument object of the generic function's arrow object. If the formal parameter is unspecialized, we consider the formal to be implicitly specialized on the associated object in the generic function's arrow object.

Figure 3-7 shows the static analog of the dynamic method lookup rule. The rule checks that generic function *c* has a most-specific method for the argument tuple $[t_1,\ldots,t_m]$. This is the main subroutine in the implementation-side typechecking of a generic function (each type system will fill in the other details of implementation-side typechecking).

The rules for extending the direct inheritance relation to form the descendant relation are given in figure 3-8. These are simply the analogs of the associated rules in the dynamic semantics.

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

$$\text{[obj-s]}\quad \frac{\begin{array}{c}p,k \vdash O_1 : t_1 \quad \ldots \quad p,k \vdash O_n : t_n \\ \{c_1,\ldots,c_m\} = \{c \mid 1 \le j \le n \wedge c = t_j\} \\ \{\} = \{art \mid 1 \le j \le m \wedge (c_j,art) \in isa(k)\} \cup \{art \mid 1 \le j \le n \wedge art = t_j\} \\ k_0 = (isa = \{(\mathbf{object}(i),c_1),\ldots,(\mathbf{object}(i),c_m)\}, ls=\{\mathbf{object}(i)\})\end{array}}{p,k \vdash \mathbf{interface\ object}_i\ I\ \mathbf{isa}\ O_1,\ldots,O_n \succ \{(I, \mathbf{object}(i))\},k_0}$$

$$\text{[gf-s]}\quad \frac{\begin{array}{c}p,k \vdash O_1 : t_1 \quad \ldots \quad p,k \vdash O_n : t_n \\ \{c_1,\ldots,c_m\} = \{c \mid 1 \le j \le n \wedge c = t_j\} \\ \{art_1,\ldots,art_q\} = \{art \mid 1 \le j \le m \wedge (c_j,art) \in isa(k)\} \cup \{art \mid 1 \le j \le n \wedge art = t_j\} \\ 1 \le r \le q \quad k \vdash art_r \le_{isa} art_1 \quad \ldots \quad k \vdash art_r \le_{isa} art_q \\ k_0 = (isa = \{(\mathbf{object}(i),c_1),\ldots,(\mathbf{object}(i),c_m), (\mathbf{object}(i),art_r)\}, ls=\{\mathbf{object}(i)\})\end{array}}{p,k \vdash \mathbf{interface\ object}_i\ I\ \mathbf{isa}\ O_1,\ldots,O_n \succ \{(I, \mathbf{object}(i))\},k_0}$$

$$\text{[abso-s]}\quad \frac{p,k \vdash \mathbf{interface\ object}_i\ I\ \mathbf{isa}\ O_1,\ldots,O_n \succ p_0,k_0}{p,k \vdash \mathbf{abstract\ object}_i\ I\ \mathbf{isa}\ O_1,\ldots,O_n \succ p_0,k_0+(abs = \{\mathbf{object}(i)\})}$$

$$\text{[cnco-s]}\quad \frac{p,k \vdash \mathbf{interface\ object}_i\ I\ \mathbf{isa}\ O_1,\ldots,O_n \succ p_0,k_0}{p,k \vdash \mathbf{object}_i\ I\ \mathbf{isa}\ O_1,\ldots,O_n \succ p_0,k_0 + (cnc = \{\mathbf{object}(i)\})}$$

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

**Figure 3-4: Judgements for declarations**

$$\text{[look-s]}\quad \frac{\begin{array}{c}[mh_1,\ldots,mh_n] = [mh \mid (c,mh) \in gfms(k) \wedge k \vdash [t_1,\ldots,t_m] \le_{isa*} mh] \\ disjoint(mh_1,\ldots,mh_n) \qquad 1 \le i \le n \\ k \vdash mh_i \le_{isa*} mh_1 \quad \ldots \quad k \vdash mh_i \le_{isa*} mh_n\end{array}}{k \vdash c\ lookup\text{-}method\ [t_1,\ldots,t_m]}$$

**Figure 3-7: Judgement for method lookup**

[app-s] $$\frac{p,k \vdash E_0 : \mathbf{unk}(\mathbf{arrow}([t_1,...,t_n],t)) \quad p,k \vdash E_1 : \mathbf{unk}(t_1) \quad ... \quad p,k \vdash E_n : \mathbf{unk}(t_n)}{p,k \vdash E_0(E_1,...,E_n) : \mathbf{unk}(t)}$$

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

[smpc-s] $$\frac{p,k \vdash E : c \qquad k \vdash c \leq_{isa} t \qquad c \in cnc(k)}{p,k \vdash E : \mathbf{unk}(t)}$$

[smpu-s] $$\frac{p,k \vdash E : \mathbf{unk}(t_1) \qquad k \vdash t_1 \leq_{isa} t_2}{p,k \vdash E : \mathbf{unk}(t_2)}$$

[art-s] $$\frac{p,k \vdash O_1 : t_1 \quad ... \quad p,k \vdash O_n : t_n \qquad p,k \vdash O : t}{p,k \vdash (O_1,...,O_n) \rightarrow O : \mathbf{arrow}([t_1,...,t_n],t)}$$

**Figure 3-5: Judgements for expressions and types**

[form-s] $$\frac{p,k \vdash (c,F_1,1) \succ t_1 \quad ... \quad p,k \vdash (c,F_n,n) \succ t_n}{p,k \vdash (c,[F_1,...,F_n]) \succ [t_1,...,t_n]}$$

[spe-s] $$\frac{(c,\mathbf{arrow}([t_1,...,t_n],t)) \in con(k) \quad p,k \vdash I_2 : c_0 \quad c_0 \in cnc(k) \cup abs(k) \quad k \vdash c_0 \leq_{isa} t_i}{p,k \vdash (c, I_1@I_2, i) \succ c_0}$$

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

**Figure 3-6: Judgements for formal arguments**

## 3.2 System G

This subsection provides additions to the base static semantics that enforce the typing restrictions of System G. In particular, we provide typing rules for the *has-safe* "hooks" in the base static semantics of section 3.1.

The typing rules for *program-has-safe-modules* are shown in figure 3-9. The rules globally perform implementation-side typechecking on each concrete generic function in the program. In particular, for each concrete generic function accepting $n$ arguments, all possible argument tuples $[c_1,...,c_n]$ are formed such that each $c_i$ is concrete and descends from the corresponding object in the generic function's arrow object. It is checked that each of these argument tuples has a most-specific method in the generic function.

[isa*-s] $$\frac{k \vdash t_1 \leq_{isa} t_1' \quad \ldots \quad k \vdash t_n \leq_{isa} t_n'}{k \vdash [t_1,\ldots,t_n] \leq_{isa*} [t_1',\ldots,t_n']}$$

[isb-s] $$\frac{(t_1,t_2) \in isa(k)}{k \vdash t_1 \leq_{isa} t_2}$$

[isr-s] $$k \vdash t \leq_{isa} t$$

[ist-s] $$\frac{k \vdash t_1 \leq_{isa} t_2 \quad k \vdash t_2 \leq_{isa} t_3}{k \vdash t_1 \leq_{isa} t_3}$$

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

**Figure 3-8: Judgements for the descendant relation**

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

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

[Gleg-s] $$\frac{\forall\, 1 \leq i \leq n.\ c_i \in cnc(k) \wedge k \vdash c_i \leq_{isa} t_i}{k \vdash [c_1,\ldots,c_n]\ \textit{G-is-legal-tuple-for}\ (c,\mathbf{arrow}([t_1,\ldots,t_n],t))}$$

**Figure 3-9: Global typechecking for System G**

The local and regional typechecking rules for System G are shown in figure 3-10. These rules are all trivially satisfied. Because of the global implementation-side typechecking, there is no need for any local or regional typechecking.

## 3.3 System M

This subsection provides additions to the base static semantics that enforce the typing restrictions of System M. The rule for *program-has-safe-modules* is shown in figure 3-11. Since System M requires no global typechecking, this rule is trivially satisfied.

The rest of the rules implement restrictions ***M1-M4*** as well as the two kinds of re-implementation-side typechecking required for importers. The restrictions on methods and objects appear in figure 3-12. The *module-has-safe-methods* rule implements restriction ***M1***, which ensures that, for each

[gfsf-s] $k \vdash$ *module-has-safe-gfs*

[igfsf-s] $k \vdash$ *module-has-safe-imported-gfs*

[obsf-s] $k \vdash$ *module-has-safe-objects*

[mtsf-s] $k \vdash$ *module-has-safe-methods*

**Figure 3-10: Local and regional typechecking for System G**

[mdsf-s] $k,m \vdash$ *program-has-safe-modules*

**Figure 3-11: Global typechecking for System M**

$$\text{[mtsf-s]}\quad \frac{c^* = \{c \mid (c,[t_1,\ldots,t_n]) \in lgf(k)\} \quad k \vdash \textit{restriction-M1 } c^*}{k \vdash \textit{module-has-safe-methods}}$$

$$\text{[M1-s]}\quad \frac{\forall (c,[t_1,\ldots,t_n]) \in lgf(k).\ c \in c^* \Rightarrow (c \in ls(k) \vee (n > 0 \wedge t_1 \in ls(k)))}{k \vdash \textit{restriction-M1 } c^*}$$

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

$$\text{[M2-s]}\quad \frac{\begin{array}{c}\forall\ c \in ls(k).\ \forall\ c_1 \in cnc(k) \cup abs(k).\ \forall\ c_2 \in cnc(k) \cup abs(k)\ .\\ (c \in cnc(k) \cup abs(k) \wedge k \vdash c \leq_{isa} c_1 \wedge k \vdash c \leq_{isa} c_2 \wedge c_1 \notin ls(k) \wedge c_2 \notin ls(k)) \Rightarrow \\ ((k \vdash c_1 \leq_{isa} c_2 \vee k \vdash c_2 \leq_{isa} c_1) \vee \\ (c_3 \in cnc(k) \cup abs(k) \wedge k \vdash c \leq_{isa} c_3 \wedge k \vdash c_3 \leq_{isa} c_1 \wedge k \vdash c_3 \leq_{isa} c_2))\end{array}}{k \vdash \textit{restriction-M2}}$$

**Figure 3-12: Restrictions *M1* and *M2***

method created in the current module, either the method was added to a local generic function or the method is encapsulated-style (the method's first specializer is a local object). The *module-has-safe-objects* rule implements restriction ***M2***, which disallows unanticipated multiple code inheritance across module boundaries.

The rules for implementation-side typechecking of a module appear in figure 3-13. The rules ensure that, for each generic function created in the current module, each legal argument tuple $[t_1,\ldots,t_n]$ has a most-specific method to invoke. The *M-is-legal-tuple-for* rule defines which argument tuples are checked for a most-specific method. This rule implements restrictions ***M3*** and ***M4.*** In particular, let **arrow**$([t_1',\ldots,t_n'],t)$ be the arrow object of the generic function being checked. For any argument position $i$ other than the first, all descendants of $t_i'$ are checked, regardless of whether or

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

$$\text{[Mgfs-s]}\quad \frac{\begin{array}{c}\forall\, c \in \{c_1', \ldots, c_r'\}.\, \forall\, [t_1, \ldots, t_n].\\ (c \in cnc(k) \wedge (c, art) \in isa(k) \wedge k \vdash [t_1, \ldots, t_n]\ \textit{M-is-legal-tuple-for}\ (c, art)) \Rightarrow\\ k \vdash c\ \textit{lookup-method}\ [t_1, \ldots, t_n]\end{array}}{k \vdash \textit{M-impl-side-typecheck-gfs } \{c_1', \ldots, c_r'\}}$$

$$\text{[Mleg-s]}\quad \frac{\forall\, 1 \le i \le n.k \vdash t_i \le_{isa} t_i' \qquad (t_1 \in cnc(k) \ \vee\ t_1\ \textit{is-non-local})}{k \vdash [t_1, \ldots, t_n]\ \textit{M-is-legal-tuple-for}\ (c, \textbf{arrow}([t_1', \ldots, t_n'], t))}$$

**Figure 3-13: Implementation-side typechecking for System M**

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

$$\text{[Migfs-s]}\quad \frac{\begin{array}{c}\forall\, c \in c^*.\, \forall\, [t_1, \ldots, t_n].\ ((c, art) \in isa(k) \wedge\\ (k \vdash [t_1, \ldots, t_n]\ \textit{is-local-recheck1-for}\ (c, art) \vee\\ k \vdash [t_1, \ldots, t_n]\ \textit{is-local-recheck2-for}\ (c, art))) \Rightarrow\\ k \vdash c\ \textit{lookup-method}\ [t_1, \ldots, t_n]\end{array}}{k \vdash \textit{M-impl-side-typecheck-imported-gfs } c^*}$$

$$\text{[re1-s]}\quad \frac{\begin{array}{c}k \vdash [t_1, \ldots, t_n]\ \textit{M-is-legal-tuple-for}\ (c, \textbf{arrow}([t_1', \ldots, t_n'], t))\\ (c, mh) \in lgf(k) \qquad k \vdash [t_1, \ldots, t_n] \le_{isa^*} mh\end{array}}{k \vdash [t_1, \ldots, t_n]\ \textit{is-local-recheck1-for}\ (c, \textbf{arrow}([t_1', \ldots, t_n'], t))}$$

$$\text{[re2-s]}\quad \frac{\begin{array}{c}k \vdash [t_1, \ldots, t_n]\ \textit{M-is-legal-tuple-for}\ (c, \textbf{arrow}([t_1', \ldots, t_n'], t))\\ k \vdash t_1\ \textit{is-orphan}\end{array}}{k \vdash [t_1, \ldots, t_n]\ \textit{is-local-recheck2-for}\ (c, \textbf{arrow}([t_1', \ldots, t_n'], t))}$$

$$\text{[orph-s]}\quad \frac{\begin{array}{c}k \vdash t\ \textit{is-non-local} \qquad c \in ls(k)\\ \{c\} = \{c_0 \mid c_0 \in cnc(k) \wedge k \vdash c_0 \le_{isa} t \wedge k \vdash c \le_{isa} c_0\}\end{array}}{k \vdash c\ \textit{is-orphan}}$$

**Figure 3-14: Re-implementation-side typechecking for System M**

not these descendants are concrete. Concrete descendants of $t_1'$ are checked, as well as any non-local descendants of $t_1'$.

The rules for the two kinds of re-implementation-side typechecking of imported generic functions are shown in figure 3-14. The [Migfs-s] rule ensures that, for each imported generic function, all

[isnloc-s] $$\frac{k\vdash t \textit{ has-no-local-in-positive-position}}{k\vdash t \textit{ is-non-local}}$$

[nlposb-s] $$\frac{c \notin ls(k)}{k\vdash c \textit{ has-no-local-in-positive-position}}$$

[nlposa-s] $$\frac{(\forall 1\leq i \leq n. k\vdash t_i \textit{ has-no-local-in-negative-position}) \quad k\vdash t_0 \textit{ has-no-local-in-positive-position}}{k\vdash \textbf{arrow}([t_1,...,t_n],t_0) \textit{ has-no-local-in-positive-position}}$$

[nlnegb-s] $$k\vdash c \textit{ has-no-local-in-negative-position}$$

[nlnega-s] $$\frac{(\forall\ 1\leq i \leq n. k\vdash t_i \textit{ has-no-local-in-positive-position}) \quad k\vdash t_0 \textit{ has-no-local-in-negative-position}}{k\vdash \textbf{arrow}([t_1,...,t_n],t_0) \textit{ has-no-local-in-negative-position}}$$

**Figure 3-15: Judgements for determining if an object is non-local**

argument tuples satisfying one of the two re-check conditions has a most-specific method to invoke. A tuple satisfies rule [re1-s] if it is a legal argument tuple and the generic function has an applicable method that was created in the current module. A tuple satisfies rule [re2-s] if it is a legal argument tuple and its first component is an orphan. An orphan is a local, concrete object that descends from a non-local, non-concrete object *t* without also descending from a concrete descendant of *t*.

Finally, the rules defining when an object is non-local appear in figure 3-15. A non-arrow object is non-local if it was not created in the current module. An arrow object is non-local if it does not have a local object in a positive position.

## 3.4 System E

This subsection provides modifications and additions to the base static semantics for System E. Section 3.4.1 presents the modifications necessary to accomodate the augmented syntax of the language (see section 1). Section 3.4.2 presents the implementation of System E's modular typing restrictions.

### 3.4.1 Modifications to the base static semantics

Figure 3-16 defines the necessary modifications and additions to the domains for the static semantics. The *TypeStore* now includes an element of the *Specializers* domain. This domain records, for each generic function, which argument positions are marked and which are unmarked. An argument position gets the *Spec* **s** (for *specializable*) if it is marked, and otherwise it gets the *Spec* **u** (for *unspecialized*). The *Modules* domain includes a component that records the declared extension relation between modules. This will be used to ensure that each module has a most-extending module in the program. Finally, the *LocalTypeStore* domain includes a component that records all non-arrow objects that were declared in (transitive) extendees of the current module.

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

**Figure 3-16: Modifications and additions to the domains for the static semantics**

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

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

$\mathit{add\text{-}ext}((\mathit{isa},\mathit{cnc},\mathit{abs},\mathit{gfms},\mathit{sp},(\mathit{ls},\mathit{ex},\mathit{lgf}))) = (\mathit{isa},\mathit{cnc},\mathit{abs},\mathit{gfms},\mathit{sp},(\{\},\mathit{ls} \cup \mathit{ex},\{\}))$

$\mathit{type}(O) = O \quad \mathit{type}(\#O) = O$

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

**Figure 3-17: Modifications and additions to the definitions and functions**

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

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

**Figure 3-18: Updated judgements for module blocks and modules**

Figure 3-17 makes modifications and additions to the definitions and functions. First the appropriate accessor functions are defined on the modified *TypeStore* and *Module* domains. The *add-imp* function takes a *TypeStore* for a module and returns the "interface" information needed by an importer of the module. The *add-ext* function is similar, but for extenders of the module. In particular, the *add-ext* function appropriately updates the extender's list of objects created in extendees. The *type* and *spec* functions are simple accessor functions on elements of the syntax domain *S*, which represents a possibly marked object.

$$\{I_1,...,I_m\} = \{I \mid 1 \le j \le n \wedge I = O_j\} \quad \{O_1',...,O_o'\} = \{O_1,...,O_n\} - \{I_1,...,I_m\}$$
$$p,k \vdash I_1 : c_1 \; ... \; p,k \vdash I_m : c_m \quad p,k \vdash O_1' : t_1 \succ s_1^{*\prime} ... \; p,k \vdash O_o' : t_o \succ s_o^{*\prime}$$
$$\{(art_1, s_1^*),...,(art_q,s_q^*)\} = \{(art,s^*) \mid 1 \le j \le m \wedge (c_j,art) \in isa(k) \wedge (c_j,s^*) \in sp(k)\}$$
$$\cup \{(art,s^*) \mid 1 \le j \le n \wedge art = t_j \wedge s^* = s_j^{*\prime}\}$$
$$1 \le r \le q \quad k \vdash art_r \le_{isa} art_1 \; ... \; k \vdash art_r \le_{isa} art_q$$
$$k_0 = (isa = \{(\mathbf{object}(i),c_1),...,(\mathbf{object}(i),c_m), (\mathbf{object}(i),art_r)\}, ls = \{\mathbf{object}(i)\},$$
$$sp = \{(\mathbf{object}(i),s_r^*)\})$$

[gf-s]

$$p,k \vdash \mathbf{interface\ object}_i\ I\ \mathbf{isa}\ O_1, ..., O_n \succ \{(I, \mathbf{object}(i))\}, k_0$$

$$O_1 = type(S_1) \; ... \; O_n = type(S_n)$$
$$s_1 = spec(S_1) \; ... \; s_n = spec(S_n)$$
$$p,k \vdash O_1 : t_1 \; ... \; p,k \vdash O_n : t_n \quad p,k \vdash O : t$$

[art-s]

$$p,k \vdash (S_1,...,S_n) \to O : \mathbf{arrow}([t_1,...,t_n],t) \succ [s_1,...,s_n]$$

**Figure 3-19: Updated judgements for generic functions and arrow objects**

Figure 3-18 shows the updated rules for typechecking module blocks and modules. The rule for module blocks simply takes into account the fact that the *Modules* domain now has one more component than it used to have. The [mod-s] rule is revised in order to use both importees and extendees as context in the evaluation of a module's declarations.

Figure 3-19 shows the modified rule for typechecking generic functions, as well as the rule for typechecking arrow objects with possibly marked argument positions. The generic function rule is identical to the original rule for generic functions, except that the new rule must record which positions of the new generic function are marked and which are unmarked. The rule simply uses whatever markings are on the generic function's most-specific arrow object for this purpose. The complexity in this augmented rule comes from the need to track the markings of each inherited arrow object. The rule for typechecking arrow objects is identical to the old rule, except that the new rule also extracts and returns the argument position markings.

### 3.4.2 Additions to the base static semantics

Now we provide the judgements implementing System E's modular typechecking restrictions. The rule for *program-has-safe-modules* is shown in figure 3-20. This rule implements the check that each module has a unique most-extending module in the program. For this purpose, the $\le_{ext}$ relation is defined as the reflexive, transitive closure of the declared extension relation. The check for most-extending modules is the only global check needed by System E.

The rest of the rules implement restrictions ***E1-E4*** as well as the two kinds of re-implementation-side typechecking required for importers. The restrictions on methods and objects appear in figure 3-21. The *module-has-safe-methods* rule implements restrictions ***E1a*** and ***E1b***. The rule for ***E1a*** ensures that methods do not specialize on unmarked positions of their associated generic function. The rule for ***E1b*** ensures that methods added to imported generic functions are all-local multimethods. The rule also requires such methods to have at least one marked argument position. Because the method must be all-local, this restriction forces at least one specializer to be a local object, so the method will not conflict with unseen methods. The *module-has-safe-objects* rule implements restriction ***E2***, which restricts multiple inheritance across module boundaries.

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

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

$$\text{[exb-s]}\quad \frac{(I_1, I_2) \in \textit{ext}(m)}{m \vdash I_1 \leq_{ext} I_2}$$

$$\text{[exr-s]}\quad m \vdash I \leq_{ext} I$$

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

**Figure 3-20: Global typechecking for System E**

$$\text{[mtsf-s]}\quad \frac{c^* = \{c \mid (c,[t_1,\ldots,t_n]) \in \textit{lgf}(k)\} \quad k \vdash \textit{restriction-E1a}\ c^* \quad k \vdash \textit{restriction-E1b}\ c^*}{k \vdash \textit{module-has-safe-methods}}$$

$$\text{[E1a-s]}\quad \frac{\forall\, (c,[t_1,\ldots,t_n]) \in \textit{lgf}(k).(c \in c^* \wedge (c,[s_1,\ldots,s_n]) \in \textit{sp}(k) \wedge (c, \textbf{arrow}([t_1',\ldots,t_n'],t)) \in \textit{isa}(k)) \Rightarrow \forall\, 1 \leq i \leq n.(s_i = \textbf{u}) \Rightarrow (t_i = t_i')}{k \vdash \textit{restriction-E1a}\ c^*}$$

$$\text{[E1b-s]}\quad \frac{\forall\, (c,[t_1,\ldots,t_n]) \in \textit{lgf}(k).\ c \in c^* \wedge c \notin \textit{ls}(k) \cup \textit{ex}(k) \Rightarrow ((c,[s_1,\ldots,s_n]) \in \textit{sp}(k) \wedge \exists\, 1 \leq i \leq n.\ s_i = \textbf{s} \wedge (\forall 1 \leq i \leq n.\ (s_i = \textbf{u}) \vee t_i \in \textit{ls}(k)))}{k \vdash \textit{restriction-E1b}\ c^*}$$

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

$$\text{[E2-s]}\quad \frac{\forall\, c \in \textit{ls}(k).(c \in \textit{cnc}(k) \cup \textit{abs}(k) \wedge k \vdash c \leq_{isa} t \wedge k \vdash t\ \textit{is-imported}) \Rightarrow (\forall\, c_1 \in \textit{cnc}(k) \cup \textit{abs}(k).\forall\, c_2 \in \textit{cnc}(k) \cup \textit{abs}(k).\ (k \vdash c \leq_{isa} c_1 \wedge k \vdash c \leq_{isa} c_2 \wedge c_1 \notin \textit{ls}(k)) \Rightarrow (k \vdash c_1 \leq_{isa} c_2 \vee k \vdash c_2 \leq_{isa} c_1))}{k \vdash \textit{restriction-E2}}$$

**Figure 3-21: Restrictions *E1* and *E2***

$$\text{[gfsf-s]}\quad \frac{k \vdash \textit{E-impl-side-typecheck-gfs}\ (ls(k) \cup ex(k)) \qquad k \vdash \textit{restriction-E4a}\ ls(k)}{k \vdash \textit{module-has-safe-gfs}}$$

$$\text{[Egfs-s]}\quad \frac{\begin{array}{c}\forall\, c \in \{c_1', \ldots, c_r'\}.\ \forall\, [t_1, \ldots, t_n]. \\ (c \in cnc(k) \wedge (c, art) \in isa(k) \wedge k \vdash [t_1, \ldots, t_n]\ \textit{E-is-legal-tuple-for}\ (c, art)) \Rightarrow \\ k \vdash c\ \textit{lookup-method}\ [t_1, \ldots, t_n]\end{array}}{k \vdash \textit{E-impl-side-typecheck-gfs}\ \{c_1', \ldots, c_r'\}}$$

$$\text{[Eleg-s]}\quad \frac{\begin{array}{c}(c, [s_1, \ldots, s_n]) \in sp(k) \\ \forall\, 1 \le i \le n. k \vdash t_i \le_{isa} t_i' \\ \forall\, 1 \le q \le n. \forall\, 1 \le r \le n. \\ ((s_q = s_r = \mathbf{s}) \Rightarrow (q = r)) \Rightarrow (t_q \in cnc(k)\ \vee k \vdash t_q\ \textit{is-non-local})\end{array}}{k \vdash [t_1, \ldots, t_n]\ \textit{E-is-legal-tuple-for}\ (c, \mathbf{arrow}([t_1', \ldots, t_n'], t))}$$

$$\text{[E4a-s]}\quad \frac{\begin{array}{c}\forall\, c \in c^*. \forall\, \mathbf{arrow}([t_1, \ldots, t_n], t). ((c, [s_1, \ldots, s_n]) \in sp(k) \wedge c \le_{isa} \mathbf{arrow}([t_1, \ldots, t_n], t)) \Rightarrow \\ \forall 1 \le i \le n. (k \vdash t_i\ \textit{is-imported} \Rightarrow (s_i = \mathbf{u}))\end{array}}{k \vdash \textit{restriction-E4a}\ c^*}$$

**Figure 3-22: Implementation-side typechecking for System E**

The modular typechecking restrictions for generic functions appear in figure 3-22. First the rules for implementation-side typechecking the generic functions in a module are given. These rules ensure that, for each generic function created in the current module, each legal argument tuple $[t_1,\ldots,t_n]$ has a most-specific method to invoke. The *E-is-legal-tuple-for* rule defines which argument tuples are checked for a most-specific method. This rule implements restrictions ***E3a***, ***E3b***, and ***E4b.*** In particular, let $\mathbf{arrow}([t_1',\ldots,t_n'],t)$ be the arrow object of the generic function being checked. For any argument position $i$, all descendants of $t_i'$ are checked, regardless of whether or not these descendants are concrete. However, if the generic function is singly dispatched and $q$ is the single marked position, descendants of $t_q'$ are checked only if they are concrete or non-local. Finally, rule [E4a-s] enforces restriction ***E4a***, which checks that if any descendant of an argument object in a generic function's arrow object is imported, then the associated argument position is unmarked. Note that all arrow objects from which the generic function descends are checked in rule [E4a-s], which by contravariance has the effect of checking all descendants of the argument objects in the generic function's most-specific arrow object.

The rules for the two kinds of re-implementation-side typechecking of imported generic functions are shown in figure 3-23. The [Eigfs-s] rule ensures that, for each imported generic function, all argument tuples satisfying one of the two re-check conditions have a most-specific method to invoke. A tuple satisfies rule [re1-s] if it is a legal argument tuple and the generic function has an applicable method that was created in the current module. A tuple satisfies rule [re2-s] if it is a legal argument tuple, the generic function is singly dispatched, and the tuple has an orphan at the single marked position. We use the same definition of an orphan as in System M (rule [orph-s]).

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

$$\text{[Eigfs-s]}\quad \frac{\forall\, c \in c^*.\, \forall\, [t_1,\ldots,t_n].\ ((c,art) \in isa(k) \wedge (k \vdash [t_1,\ldots,t_n]\ \textit{is-local-recheck1-for}\ (c,art) \vee k \vdash [t_1,\ldots,t_n]\ \textit{is-local-recheck2-for}\ (c,art))) \Rightarrow k \vdash c\ \textit{lookup-method}\ [t_1,\ldots,t_n]}{k \vdash \textit{E-impl-side-typecheck-imported-gfs } c^*}$$

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

$$\text{[re2-s]}\quad \frac{k \vdash [t_1,\ldots,t_n]\ \textit{E-is-legal-tuple-for}\ (c, \mathbf{arrow}([t_1,\ldots,t_n],t)) \quad (c,[s_1,\ldots,s_n]) \in sp(k) \qquad 1 \leq i \leq n \qquad s_i = \mathbf{s} \qquad \forall 1 \leq j \leq n.\ s_j = \mathbf{s} \Rightarrow i=j \quad k \vdash t_i\ \textit{is-orphan}}{k \vdash [t_1,\ldots,t_n]\ \textit{is-local-recheck2-for}\ (c, \mathbf{arrow}([t_1,\ldots,t_n],t))}$$

**Figure 3-23: Re-implementation-side typechecking for System E**

$$\text{[isimp-s]}\quad \frac{k \vdash t\ \textit{is-non-local} \quad k \vdash t\ \textit{has-imported-in-positive-position}}{k \vdash t\ \textit{is-imported}}$$

$$\text{[imposb-s]}\quad \frac{c \notin ls(k) \cup ex(k)}{k \vdash c\ \textit{has-imported-in-positive-position}}$$

$$\text{[imposa-s]}\quad \frac{(\exists\ 1 \leq i \leq n.k \vdash t_i\ \textit{has-imported-in-negative-position}) \vee k \vdash t_0\ \textit{has-imported-in-positive-position}}{k \vdash \mathbf{arrow}([t_1,\ldots,t_n],t_0)\ \textit{has-imported-in-positive-position}}$$

$$\text{[imnega-s]}\quad \frac{(\exists\ 1 \leq i \leq n.k \vdash t_i\ \textit{has-imported-in-positive-position}) \vee k \vdash t_0\ \textit{has-imported-in-negative-position}}{k \vdash \mathbf{arrow}([t_1,\ldots,t_n],t_0)\ \textit{has-imported-in-negative-position}}$$

**Figure 3-24: Judgements for determining if an object is imported**

We use the rules in System M for defining when an object is non-local (figure 3-15). Figure 3-24 shows the rules defining when an object is imported. An object is imported if it is non-local and has an imported object in a positive position.

$$[\text{mtsf-s}]\quad \frac{\begin{array}{c} c_1^* = \{c \mid (c,[t_1,\ldots,t_n]) \in lgf(k) \wedge k \vdash c \text{ is-M-gf}\} \\ c_2^* = \{c \mid (c,[t_1,\ldots,t_n]) \in lgf(k) \wedge k \vdash c \text{ is-E-gf}\} \\ k \vdash \text{restriction-M1 } c_1^* \\ k \vdash \text{restriction-E1a } c_2^* \quad k \vdash \text{restriction-E1b } c_2^* \end{array}}{k \vdash \text{module-has-safe-methods}}$$

$$[\text{obsf-s}]\quad \frac{k \vdash \text{restriction-M2} \qquad k \vdash \text{restriction-E2}}{k \vdash \text{module-has-safe-objects}}$$

$$[\text{gfsf-s}]\quad \frac{\begin{array}{c} c_1^* = \{c \mid c \in ls(k) \wedge k \vdash c \text{ is-M-gf}\} \qquad k \vdash \text{M-impl-side-typecheck-gfs } c_1^* \\ c_2^* = \{c \mid c \in ls(k) \cup ex(k) \wedge k \vdash c \text{ is-E-gf}\} \qquad k \vdash \text{E-impl-side-typecheck-gfs } c_2^* \\ c_3^* = \{c \mid c \in ls(k) \wedge k \vdash c \text{ is-E-gf}\} \qquad k \vdash \text{restriction-E4a } c_3^* \end{array}}{k \vdash \text{module-has-safe-gfs}}$$

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

**Figure 3-25: Judgements for System ME**

## 3.5 System ME

This subsection presents the typing restrictions for System ME. The full set of judgements for System ME is the union of the rules presented in this subsection, the base static semantics in section 3.1, the rules for System E in section 3.4, and the rules for System M in section 3.3 that are not overridden by rules of the same name in System E.

The main typing rules for System ME are presented in figure 3-25. The rules simply invoke the appropriate restrictions from Systems M and E. In particular, the generic functions are partitioned into those that use System M's restrictions and those that use System E's restrictions. A System M generic function obeys restrictions ***M1***, ***M3***, and ***M4***, and similarly for a System E generic function. All objects must obey both restrictions ***M2*** and ***E2***, which greatly limit multiple inheritance across module boundaries.

Finally, figure 3-26 shows the rules that define which typing restrictions are used for each generic function. A generic function with no marked positions uses System M's restrictions, and a generic function with at least one marked position uses System E's restrictions.

# 4 Type Soundness

This section sketches our proof that Dubious's static semantics is sound with respect to its dynamic semantics. Section 4.1 overviews our proof method, which is based on prior work on type soundness by Wright and Felleisen [Wright & Felleisen 94]. Section 4.2 describes the key lemma for each of Systems G, M, E, and ME.

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

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

**Figure 3-26: Judgements for partitioning generic functions**

## 4.1 Proof Outline

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

$$\text{[val-s]}\quad p,k \vdash \mathbf{obj}(i) : \mathbf{object}(i)$$

$$\text{[valart-s]}\quad \frac{p,k \vdash o_1 : t_1 \ldots p,k \vdash o_n : t_n \quad p,k \vdash o : t}{p,k \vdash \mathbf{arrow}([o_1,\ldots,o_n],o) : \mathbf{arrow}([t_1,\ldots,t_n],t)}$$

Since neither of these rules makes use of the given *TypeEnv* and *TypeStore*, we often omit one or both of these contexts to the judgements. We consider *Obj* elements to be members of the *E* syntactic domain. In particular, this allows the use of [smpc-s] and [smpu-s] to provide subsumption for the types of dynamic objects.

Next we identify several correspondences between the static and dynamic contexts of a program. In particular, suppose $\vdash M_1,\ldots,M_n \succ k,m$ and $\vdash M_1,\ldots,M_n \succ s,me$. Let $p = mp(m)(I)$ and $e = me(I)$, for some $I \in domain(mp(m))$. We identify that $p$ and $e$ are isomorphic in a certain sense. In particular, for every pair $(I,ot) \in p$, there exists a pair $(I,v) \in e$ such that $p,k \vdash v : ot$. Conversely, for every pair $(I,v) \in e$, there exists a pair $(I,ot) \in p$ such that $p,k \vdash v : ot$. Similarly, we identify correspondences between $isa(k)$ and $isa(s)$ and between $gfms(k)$ and $gfms(s)$. Intuitively, $isa(k)$ and $isa(s)$ are isomorphic because each records all of the direct inheritance relationships declared in $M_1,\ldots,M_n$. Similarly, each of $gfms(k)$ and $gfms(s)$ records relevant information about each method declared in $M_1,\ldots,M_n$.

Now we prove a subject reduction lemma:

**Lemma 1.** (Subject Reduction) Suppose $\vdash M_1,\ldots,M_n \succ k,m$ and $\vdash M_1,\ldots,M_n \succ s,me$. 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:** Given the correspondences identified above, the result follows by induction on the length of the derivation in the dynamic semantics that $e,s \vdash E \Rightarrow o$.

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

**Definition 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 $\vdash M_1,...,M_n \succ s,me$ 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 $e,s \vdash E$.

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 $\vdash M_1,...,M_n \succ k,m$ and $\vdash M_1,...,M_n \succ s,me$. 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 prove that none of the four cases in the definition of faulty expressions holds. The isomorphism between $p$ and $e$ 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 only part of the entire soundness proof that depends on which type system (G, M, E, or ME) is used. For each of these systems, we prove that every legal message send has a most-specific applicable method in $k$. This is the key lemma of the soundness proof, and section 4.2 sketches it for each of the four type systems. Ruling out case three is completed by showing that the isomorphisms between $isa(k)$ and $isa(s)$ and between $gfms(k)$ and $gfms(s)$ are sufficient to prove that method specificity in $k$ is isomorphic to method specificity in $s$. So in particular, if a message send has a most-specific applicable method in $k$, the message send also has a most-specific applicable method in $s$.

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

**Theorem 1.** (Type Soundness) Let $P = M_1,...,M_n$ **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 $\vdash M_1,...,M_n \succ k,m$ and $p,k \vdash E : ot$. First we show that, since no message sends are evaluated during the evaluation of modules, successful typechecking of modules implies successful evaluation of modules. In particular, $\vdash M_1,...,M_n \succ k,m$ in the static semantics implies $\vdash M_1,...,M_n \succ s,me$ in the dynamic semantics, for some store $s$ and module environment $me$ in the appropriate correspondence with $k$ and $mp(m)$. 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, $p,k \vdash o : ot$.

## 4.2 Key Lemma

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

**Lemma 4.** (Key Lemma) Suppose $\vdash M_1,...,M_n \succ k,m$. If $(c, \mathbf{arrow}([t_1,...,t_n],t)) \in isa(k), c \in cnc(k)$, and $\forall\ 1 \le i \le n.\ (c_i \le_{isa} t_i \wedge c_i \in cnc(k))$, then $k \vdash c$ *lookup-method* $[c_1,...,c_n]$.

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

### 4.2.1 System G

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

### 4.2.2 System M

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

1. The generic function accepts no arguments (that is, $n = 0$). Then implementation-side typechecks in module $M$ (rule [gfsf-s]) ensure the existence of a most-specific method implementation visible in $M$. This case is finished by showing that all applicable methods to the argument tuple [] must be visible in $M$, so module $M$'s typechecks are sufficient to ensure safety. In particular, all methods added to generic function $c$ outside of module $M$ must be encapsulated-style methods, so by rule [M1-s] these methods must have at least one argument. Therefore, these methods are not applicable to the argument tuple [].

2. The first argument object, $c_1$, is visible in $M$. First we show that all methods of $c$ applicable to $[c_1,...,c_n]$ must be visible in $M$. In particular, any method not visible in $M$ must be created in some importer $M_m$ of $M$. Then the method must be encapsulated-style, which means that its first argument specializer, $c_m$, is also not visible in $M$. If $[c_1,...,c_n]$ is applicable to the method, then $c_1$ must (transitively) inherit from $c_m$. But since $c_1$ is visible in $M$, this means that $c_m$ must also be visible in $M$, contradicting what we showed above.

   Therefore, we only need to consider methods of $c$ that are visible in module $M$. Our strategy is to build an argument tuple $[t_1',...,t_n']$ that will be considered during implementation-side typechecks of $c$ in module $M$, and then show that $[c_1,...,c_n]$ is applicable to precisely the methods of $c$ that $[t_1',...,t_n']$ is applicable to. If we show this, then we are done, because implementation-side typechecks in module $M$ ensure that $[t_1',...,t_n']$ has a most-specific method, which implies that $[c_1,...,c_n]$ has a most-specific method as well.

   Each $t_i'$ is defined as follows: If $c_i$ is visible in $M$, then $t_i' = c_i$. Otherwise, if $c_i$ has no non-interface ancestors that descend $t_i$ and are visible in $M$, $t_i' = t_i$. Otherwise, by the restrictions on cross-module multiple inheritance, we can show that $c_i$ must have a single, most-specific non-interface ancestor that descends $t_i$ and is visible in $M$, and we set $t_i'$ to be this ancestor. It is straightforward to show that the tuple $[t_1',...,t_n']$ built by this definition satisfies the requirements of the previous paragraph.

3. The first argument object, $c_1$, is not visible in $M$. Further, it was created in some module $M_0$, and $c$ is not visible in $M_0$. This is the case when $c_1$ is an "open object." We prove this case precisely as in case 2 above. First, we can show that no methods created in importers of $M$ can apply to $[c_1,...,c_n]$, or else we would be able to show that $c$ is visible in $M_0$. Then we build

$[t_1',...,t_n']$ exactly as in case 2 and show by a similar argument that $[t_1',...,t_n']$ is considered during implementation-side typechecking of $c$ in $M$ and that $[t_1',...,t_n']$ applies to precisely the same methods as does $[c_1,...,c_n]$.

4. The first argument object, $c_1$, is not visible in $M$. Further, it was created in some module $M_0$, $c$ is visible in $M_0$, and at least one of the two conditions for re-implementation-side typechecking of $c$ is met in $M_0$. That is, either $M_0$ creates a method in $c$ that is applicable to $[c_1,...,c_n]$ or $c_1$ is an orphan. The proof of this case is analogous to the proof in case 2, but with respect to module $M_0$ instead of $M$. In particular, we first prove that only methods of $c$ visible in $M_0$ can apply to $[c_1,...,c_n]$. Then we build $[t_1',...,t_n']$ as above, but with respect to module $M_0$ instead of module $M$. We show that $[t_1',...,t_n']$ is considered by the re-checks of $c$ performed by $M_0$, so it must have a most-specific method in $c$ visible in $M_0$. Therefore, so does $[c_1,...,c_n]$.

5. The first argument object, $c_1$, is not visible in $M$. Further, it was created in some module $M_0$, $c$ is visible in $M_0$, and neither of the two conditions for re-implementation-side typechecking of $c$ in $M_0$ is met. The key to proving this case is induction on the lemma, on the number of concrete ancestors of $c_1$ (not including itself) that descend from $t_1$. (The base case, when $c_1$ has zero such ancestors, is covered by cases 1-4 above.) Since no re-check condition is met, we know that $c_1$ is not an orphan with respect to $t_1$, so it must have at least one such ancestor. We prove this case by finding a concrete ancestor $c_1'$ that descends from $t_1$ such that $[c_1,...,c_n]$ applies to precisely the same methods of $c$ as does $[c_1',...,c_n]$. By the inductive hypothesis, $[c_1',...,c_n]$ has a most-specific method, which means that $[c_1,...,c_n]$ does as well.

### 4.2.3 System E

The lemma is proven for System E in an analogous way to the proof for System M. In System E, we say that an object (method) is visible in module $M$ if the object (method) was created in a (reflexive, transitive) importee or extendee of $M$. Let $c$ be the generic function object in the statement of the lemma. Since $\vdash M_1,...,M_n \succ k,m$, we know that $k,m \vdash$ *program-has-safe-modules* holds, so by [mdsf-s] in System E, we know that each module has a most-extending module. Let $M$ be the most-extending module of $M_c$, the module that created $c$.

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

Because parts of restrictions ***E3*** and ***E4*** depend on whether or not the generic function is multiply dispatched, we prove soundness separately for generic functions that are multiply dispatched and those that are not. First we sketch the proof when the generic function $c$ has zero or one marked position. The cases here are analogous to those in the sketch for System M:

1. $c$ has zero marked argument positions. Implementation-side typechecks in module $M$ (rule [gfsf-s]) ensure the existence of a most-specific method for the tuple []. Since $M$ is the most-extending module of $M_c$, any method not visible in $M$ must have been added by an importer of $M_c$. Therefore, by restriction ***E1b*** (rule [E1b-s]), the generic function must have at least

one marked position, contradicting our initial assumption. Therefore, only methods visible in $M$ can be applicable to [], so the checks by module $M$ are enough to ensure safety.

2. $c_1$ is visible by $M$. Then implementation-side typechecks in module $M$ ensure the existence of a most-specific method for the tuple $[c_1]$. Any method not visible in $M$ was added in an importer $M_m$ of $M_c$. Therefore, by restriction ***E1b***, this method must be an all-local method, so its specializer $c_1'$ is a local object and is therefore not visible in $M$. If the method is applicable to $[c_1]$, then $c_1$ inherits $c_1'$. Since $c_1$ is visible in $M$ and $c_1$ inherits $c_1'$, we can show that $c_1'$ must also be visible in $M$, contradicting what we showed above. Therefore, only methods visible in $M$ can be applicable to $[c_1]$, so $M$'s checks are enough to ensure safety.

3. $c_1$ is not visible by $M$, $c_1$ was created in module $M_0$, and $c$ is not visible in $M_0$. This is the case when $c_1$ is an "open object." As in System M, we prove this by finding an object $t_1'$ visible in $M$ such that checks on $[t_1']$ ensure the safety of $[c_1]$. We use the same definition for $t_1'$ as in System M's proof: If $c_1$ has no non-interface ancestors that descend $t_1$ and are visible in $M$, then $t_1' = t_1$. Otherwise, $t_1'$ is assigned to be the most-specific such ancestor of $c_1$. We can show that such a most-specific ancestor must exist by a combination of the restrictions on open objects of ***E4a*** and the restrictions on multiple inheritance of ***E2***. Since $c_1$ is an open object, we can show that $t_1'$ must be a non-local object to $M$, so by ***E4b*** we know that $[t_1']$ is considered in implementation-side checks of $c$ in $M$ even if $t_1'$ is non-concrete.

4. $c_1$ is not visible in $M$, $c_1$ was created in module $M_0$, and $M_0$ adds a method to $c$ that is applicable to $[c_1]$. Therefore, one of the conditions for re-implementation-side typechecking $c$ in $M_0$ is met, so re-checks ensure that there is a most-specific method implementation for $[c_1]$ out of all the methods visible in $M_0$. This case is finished by proving that the methods created in module $M_0$ that are applicable to $[c_1]$ (we know there is at least one such method) are strictly more-specific than any other applicable methods. Therefore, the re-checks from module $M_0$ are enough to ensure safety.

5. $c_1$ is not visible by $M$, $c_1$ was created in module $M_0$, and $c_1$ is an orphan. Therefore, one of the conditions for re-implementation-side typechecking $c$ in $M_0$ is met, so re-checks ensure that there is a most-specific method implementation for $[c_1]$ out of all the methods visible in $M_0$. Interestingly, there may be more-specific method implementations that are not visible in $M_0$. However, because we can show that $M_0$ must import the module creating $t_1$, $c_1$ is subject to the multiple inheritance restrictions of ***E2***. In particular, this means that all non-local non-interface parents of $c_1$ are totally ordered. Therefore, even if there are more-specific applicable methods for $[c_1]$ than the most-specific method visible in $M_0$, the combination of single dispatching and single inheritance guarantees the existence of a most-specific such method.

6. $c_1$ is not visible by $M$, $c_1$ was created in module $M_0$, and neither of the conditions for re-implementation-side typechecking $c$ in $M_0$ holds. This case is proven analogously to case 5 in System M's proof, by finding a concrete ancestor $c_1'$ of $c_1$ such that $[c_1']$ and $[c]$ apply to precisely the same methods and arguing the safety of $[c_1']$ by induction on the lemma.

Now we sketch the various cases in the proof when $c$ has at least two marked argument positions. We actually prove that $[c_1,...,c_n]$ has a most-specific method when each $c_i$ is a non-interface object, rather than only proving the lemma when each $c_i$ is concrete. This allows us to use a stronger inductive hypothesis in case 5 below.

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

2. At least one $c_i$ is visible in *M*, but some $c_j$ is not visible in *M*. Since there is at least one argument $c_i$ visible in *M*, we can use the same argument in case 2 above to show that only methods visible in *M* can be applicable to $[c_1,...,c_n]$. Then we use the same technique as in case 2 of System M's proof, building the tuple $[t_1',...,t_n']$ which is considered during implementation-side typechecks in *M*, and showing that $[t_1',...,t_n']$ is applicable to precisely the same methods as $[c_1,...,c_n]$. We use the same definition for each $t_i'$ as in that proof.

3. Each $c_i$ is not visible in *M*. Further, each $c_i$ was created in the same module, $M_0$, and $M_0$ adds a method to *c* that is applicable to $[c_1,...,c_n]$. This is just a generalization of case 4 in the proof above to multiple arguments, and the same proof technique applies here.

4. Each $c_i$ is not visible in *M*. Further, case 3 above does not hold, and each $c_i$ has zero non-local non-interface ancestors that inherit from $t_i$. We show that $[c_1,...,c_n]$ applies to precisely the same methods as $[t_1,...,t_n]$. Since each $c_i$ inherits $t_i$, it is clear that every method applicable to $[t_1,...,t_n]$ is applicable to $[c_1,...,c_n]$. Suppose there were some method applicable to $[c_1,...,c_n]$ but not to $[t_1,...,t_n]$. We show that this implies that each $c_i$ was created in the same module, $M_0$, and the applicable method was created in $M_0$ as well. Therefore, case 3 above applies, contradicting our initial assumption.

   So $[c_1,...,c_n]$ applies to precisely the same methods as $[t_1,...,t_n]$. This means that all applicable methods are visible in *M*. By restriction ***E3b***, we know that $[t_1,...,t_n]$ is considered during implementation-side typechecks of *c* in *M*, even if some $t_i$ is non-concrete. This ensures the safety of $[t_1,...,t_n]$, and hence of $[c_1,...,c_n]$.

5. Each $c_i$ is not visible in *M*. Further, cases 3 and 4 above do not hold. As in the final case of the previous proofs, we prove this by induction on the lemma. However, the induction in this case is more complicated than in the previous cases. In particular, induction is performed on the number of ancestor tuples $[c_1',...,c_n']$ of $[c_1,...,c_n]$ according to the $\leq_{isa^*}$ relation, such that each $c_i'$ is a non-interface object that descends from $t_i$. The base case is covered by cases 1-4 above. By induction, each of these ancestor tuples has a most-specific method in *c*. This case is proven by showing that the most-specific method of one of these ancestor tuples must also be the most-specific method of $[c_1,...,c_n]$.

### 4.2.4 System ME

The proof for System ME follows immediately from the proofs for Systems M and E. In particular, System ME partitions its generic functions into those that use System M's rules and those that use System E's rules. Suppose the generic function *c* in the lemma uses System M's rules. By the rules for System ME, this generic function obeys restrictions ***M1***, ***M3***, and ***M4***, and the entire program obeys restriction ***M2***. Therefore, our proof of the lemma for System M is sufficient to prove the lemma for *c*. A similar argument is used if *c* uses System E's rules.

## References

[Millstein & Chambers 99] Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. *The 13th European Conference on Object-Oriented Programming (ECOOP 99)*, Lisbon, Portugal, June 14-18, 1999.

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