Modular Typechecking for Hierarchically Extensible Datatypes and Functions

Todd Millstein Craig Chambers

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

Technical Report UW-CSE-01-07-02
July 2001, revised October 2001

Abstract

This technical report provides the formal details of EML, an ML-like language containing hierarchically, extensible datatypes and functions while retaining modular typechecking. Section 1 presents the syntax of EML. Section 2 presents its dynamic semantics, and section 3 presents its static semantics. Finally, section 4 describes formal properties of the language.

$$
\begin{array}{rcl}
P & ::= & [M]; E \\
M & ::= & I = Se \mid I = Sig \mid \texttt{functor}\ I(I' : Sig) = S \\
Se & ::= & S \mid S : Sig \mid I(Pn) \\
S & ::= & \texttt{struct}\ [D]\ \texttt{end} \mid \texttt{block}\ [Ood]\ \texttt{end} \\
Sig & ::= & I \mid \texttt{sig}\ [Spec]\ \texttt{end} \mid \texttt{sigblock}\ [Oospec]\ \texttt{end} \\
D & ::= & Tv\ L \triangleright I = E \mid L \triangleright I = Se \\
Ood & ::= & \texttt{interface}\ Tv\ L \triangleright I\ \texttt{extends}\ [Nt] \\
 & \mid & \texttt{<abstract>}\ \texttt{class}\ Tv\ L \triangleright I\ \texttt{<extends}\ Ct >\ \texttt{implements}\ [Nt]\ \texttt{of}\ T \\
 & \mid & \texttt{fun}\ Tv\ L \triangleright I : T \mid \texttt{extend fun}\ F\ Pat = E \\
Spec & ::= & Tv\ L \triangleright I : T \mid L \triangleright I : Sig \\
Oospec & ::= & \texttt{<sealed>}\ \texttt{interface}\ Tv\ L \triangleright I\ \texttt{extends}\ [Nt] \\
 & \mid & \texttt{<sealed>}\ \texttt{<abstract>}\ \texttt{class}\ Tv\ L \triangleright I\ \texttt{<extends}\ Ct >\ \texttt{implements}\ [Nt]\ \texttt{of}\ T \\
 & \mid & \texttt{<sealed>}\ \texttt{fun}\ Tv\ L \triangleright I : T \mid \texttt{extend fun}\ F\ Pat \mid \texttt{val}\ Tv\ L \triangleright I : T
\end{array}
$$

Figure 1: EML programs, modules, and declarations.

1 Syntax

The syntax of EML programs, modules, and declarations is shown in figure 1. Brackets ([]) denote a comma-separated sequence of zero or more items; zero-length sequences are denoted $\emptyset$. Angle brackets (<>) denote an optional piece of syntax; multiple optional pieces of syntax in a given production are independent of one another.

An EML program consists of a sequence of modules followed by a single expression. Like previous formalisms, module components are given both an identifier, for internal access, and a label, for external access. EML includes an ML-like module system consisting of structures, signatures, and functors. Signatures and functors may only be created at top-level, as in ML. Signature ascription and functor application may not occur within a functor body; this restriction simplifies the semantics. Finally, ascription may only be applied to a structure or block literal; this restriction disallows class and function aliasing.

A `struct` and `sig` are similar to the corresponding constructs in ML. In addition, EML has `block`s and `sigblock`s, which contain only OO declarations and specifications and may not be nested. There are four kinds of OO declarations. An `interface` declaration creates a new interface with the specified list of superinterfaces. The `class` declaration creates a fresh class. Each class declares at most one superclass, a list of interfaces that the class implements, and a representation type, which should be a record type. A class may optionally be declared `abstract`, which disallows creating instances of the class; otherwise the class is *concrete*. The `fun` declaration creates a new extensible function of the given type. The `extend fun` declaration imperatively adds a new function case to an existing extensible function; the case consists of a pattern guard and an expression. Classes, interfaces, and functions may be parameterized by a sequence of type variables.

The OO specifications are analogous. The `sealed` keyword can be used in a specification to prohibit later extension of an interface, class, or function. The `val` specification allows a function to be sealed, additionally hiding any associated function cases.

The core of EML is shown in figure 2. EML types include interfaces, classes, function types, and record types. An interface is either a path name or an ifc, which is the internal representation for an interface; ifcs cannot be referred to directly in a source program. An optional sequence of type parameters may be provided. Classes are represented similarly. The function type includes a "mark," which is a sequence of labels used to enforce the modular typechecking restrictions presented in section 3. The components of a record in EML are ordered, unlike traditional records; this simplifies the use of records to encode the fields of a class.

$$
\begin{array}{rcl}
T & ::= & Nt \mid Ct \mid T_1 \stackrel{Mark}{\longrightarrow} T_2 \mid \{L_1 : T_1, \ldots, L_n : T_n\} \\
Nt & ::= & Ts\ N \\
Ct & ::= & Ts\ C \\
N & ::= & Pn \mid ifc \\
C & ::= & Pn \mid cls \\
Mark & ::= & [L] \\
Tv & ::= & ([I]) \\
Ts & ::= & ([T]) \\
E & ::= & Pn \mid Ct\ E \mid F \mid E_1\ E_2 \mid \{L_1 = E_1, \ldots, L_n = E_n\} \\
F & ::= & Pn \mid fun \\
Pat & ::= & _ \mid I\ \texttt{as}\ Pat \mid Tv\ Ct\ Pat \mid \{L_1 = Pat_1, \ldots, L_n = Pat_n\} \\
Pn & ::= & I \mid Pn.L \\
I & ::= & \text{identifier} \\
L & ::= & \text{label}
\end{array}
$$

Figure 2: Core EML.

The expression ($Ct\ E$) instantiates a concrete class Ct, providing the new instance with representation E. An extensible function is represented internally by a fun, which is similar to a location in a language with references. Dynamically, a mapping is maintained from each fun to its bag[1] of cases. This level of indirection allows new cases to be added to an extensible function without modifying the function's value. As with ordinary ML functions, extensible functions are first-class values.

The pattern _ is the wildcard. The pattern (I `as` Pat) binds the identifier I to the given value and matches the value to Pat. The class pattern $Tv\ Ct\ Pat$ matches an instance of Ct or a subclass whose representation matches Pat; the extra type variables are necessary in case the class has more type variables than the class in the associated position of the function type.

We assume that all *binding* occurrences of identifiers in an EML program use distinct identifier names. This includes identifiers bound in modules, declarations, specifications, and patterns. This assumption is made for simplicity in the core language, freeing the semantics from having to worry about name clashes. A translation from some source language could perform the necessary name mangling.

The set of EML values is described by the following grammar:

$v ::= cls\ v \mid fun \mid \{L_1 = v_1, \ldots, L_n = v_n\}$

2 Dynamic Semantics

2.1 Preliminaries

The domains used in the dynamic semantics are shown in figure 3. We use $\rightarrow$ to denote a finite function, $\times$ to denote a cartesian product, | to denote a tagged union (with each case given a label), {} to denote a set, and {| |} to denote a bag. TopEnv maps identifiers to functors, which consist of the parameter name and the body. Env (resp., ModEnv) is a mapping from identifiers (resp., labels) to the dynamic representations of values, classes, and modules (both `structs` and `blocks`); tags are used to distinguish the three cases. An OOInfo consists of the inheritance relation, a bag of function cases, and a set recording the classes that are abstract. Each function case consists of the associated function, the pattern guard, the body expression, and the lexical environment.

[1] A bag is used in the semantics instead of a set, to catch situations when the same function case is defined multiple times.

$$
\begin{array}{rcl}
t & \in & \text{TopEnv} = I \to (I \times S) \\
e & \in & \text{Env} = I \to \text{DynRep} \\
m & \in & \text{ModEnv} = L \to \text{DynRep} \\
n & \in & \text{DynRep} = \texttt{v of } v \mid \texttt{cl of } cls \mid \texttt{mod of ModEnv} \\
s & \in & \text{OOInfo} = \text{Inh} \times \text{Cases} \times \text{Abstract} \\
i & \in & \text{Inh} = cls \to cls \\
f & \in & \text{Cases} = \{|\ \text{Case}\ |\} \\
a & \in & \text{Abstract} = \{cls\} \\
l & \in & \text{Case} = fun \times Pat \times E \times Env
\end{array}
$$

Figure 3: Domains for the dynamic semantics.

We sometimes treat a finite function as a set of pairs, i.e., a binary relation. For a binary relation $r = \{(a_1, b_1), \ldots, (a_n, b_n)\}$, $dom(r) = \{a_1, \ldots, a_n\}$. We use $\cup$ to denote the union of two sets or two bags (retaining duplicates in the case of bags). We use $-$ to denote the difference between two sets or two bags. We extend $\cup$ and $-$ to tuples of sets and bags in the obvious way.

We define the function *label* to return the newly-declared label of a declaration, specification, OO declaration, or OO specification. There is no label for an `extend fun` declaration or specification; *label* returns a fresh label name in these cases.

In the inference rules, <> encloses an optional piece; the pieces surrounded by <> in a given rule must be either all present or all absent. The syntax <<>> is used in a similar way; pieces surrounded by <<>> in a given rule must be either all present or all absent, independent of the presence or absence of pieces surrounded by <>. Some rules use yet a third such independent set of optional pieces, surrounded by <<<>>>.

Type variables and type parameters on classes are ignored by the dynamic semantics (and some of the static semantics). A missing type variable or type parameter sequence is implicitly empty.

2.2 Programs

$\boxed{\vdash P \Rightarrow v}$

$$
\frac{\begin{array}{c}\{\},\{\},(\{\},\{||\},\{\}) \vdash M_1 \Rightarrow e_1, t_1, s_1 \\ \vdots \\ e_1 \cup \ldots \cup e_{n-1}, t_1 \cup \ldots \cup t_{n-1}, s_1 \cup \ldots \cup s_{n-1} \vdash M_n \Rightarrow e_n, t_n, s_n \\ e_1 \cup \ldots \cup e_n, s_1 \cup \ldots \cup s_n \vdash E \Rightarrow v\end{array}}{\vdash M_1, \ldots, M_n; E \Rightarrow v} \quad (1)
$$

2.3 Module Expressions

$\boxed{e, t, s \vdash M \Rightarrow e', t', s'}$

$$
\frac{e, t, s \vdash Se \Rightarrow m, s'}{e, t, s \vdash I = Se \Rightarrow \{(I, \texttt{mod}(m))\}, \{\}, s'} \quad (2)
$$

$$
\frac{}{e, t, s \vdash I = Sig \Rightarrow \{\}, \{\}, (\{\}, \{||\}, \{\})} \quad (3)
$$

Rule 3: Signatures have no dynamic effect.

$$\frac{}{e,t,s \vdash \texttt{functor}\ I(I' : Sig') = S \Rightarrow \{\}, \{(I,(I',S))\}, (\{\},\{||\},\{\})} \quad (4)$$

2.4 Structure and Block Expressions

$\boxed{e,t,s \vdash Se \Rightarrow m,s'}$

$$\frac{e,t,s \vdash S \Rightarrow m,s'}{e,t,s \vdash S : Sig \Rightarrow m,s'} \quad (5)$$

Rule 5: Signature ascription has no dynamic effect.

$$\frac{t(I) = (I',S) \quad e \vdash Pn \Rightarrow m' \qquad e \cup \{(I',\texttt{mod}(m'))\}, t, s \vdash S \Rightarrow m,s'}{e,t,s \vdash I(Pn) \Rightarrow m,s'} \quad (6)$$

$\boxed{e,t,s \vdash S \Rightarrow m,s'}$

$$\frac{\begin{array}{c} e,t,s \vdash D_1 \Rightarrow e_1,m_1,s_1 \\ \cdots \\ e \cup e_1 \cup \ldots \cup e_{n-1}, t, s \cup s_1 \cup \ldots \cup s_{n-1} \vdash D_n \Rightarrow e_n,m_n,s_n \\ label(D_1),\ldots,label(D_n) \text{ distinct} \end{array}}{e,t,s \vdash \texttt{struct}\ D_1 \ldots D_n\ \texttt{end} \Rightarrow m_1 \cup \ldots \cup m_n, s_1 \cup \ldots \cup s_n} \quad (7)$$

$$\frac{\begin{array}{c} e \vdash Ood_1 \Rightarrow e_1,m_1,s_1 \\ \cdots \\ e \cup e_1 \cup \ldots \cup e_{n-1} \vdash Ood_n \Rightarrow e_n,m_n,s_n \\ label(Ood_1),\ldots,label(Ood_n) \text{ distinct} \end{array}}{e,t,s \vdash \texttt{block}\ Ood_1 \ldots Ood_n\ \texttt{end} \Rightarrow m_1 \cup \ldots \cup m_n, s_1 \cup \ldots \cup s_n} \quad (8)$$

2.5 Structure and Block Path Names

$\boxed{e \vdash Pn \Rightarrow m}$

$$\frac{e(I) = \texttt{mod}(m)}{e \vdash I \Rightarrow m} \quad (9)$$

$$\frac{e \vdash Pn \Rightarrow m' \quad m'(L) = \texttt{mod}(m)}{e \vdash Pn.L \Rightarrow m} \quad (10)$$

2.6 Declarations

$\boxed{e,t,s \vdash D \Rightarrow e',m',s'}$

$$\frac{e,s \vdash E \Rightarrow v \quad n = \texttt{v}(v)}{e,t,s \vdash Tv\ L \triangleright I = E \Rightarrow \{(I,n)\}, \{(L,n)\}, (\{\},\{||\},\{\})} \quad (11)$$

$$\frac{e,t,s \vdash Se \Rightarrow m,s' \quad n = \texttt{mod}(m)}{e,t,s \vdash L \triangleright I = Se \Rightarrow \{(I,n)\}, \{(L,n)\}, s'} \quad (12)$$

2.7 OO Declarations

$\boxed{e \vdash Ood \Rightarrow e', m', s'}$

$$\frac{}{e \vdash \texttt{interface}\ Tv\ L \triangleright I\ \texttt{extends}\ [Nt] \Rightarrow \{\}, \{\}, (\{\}, \{||\}, \{\})} \quad (13)$$

Rule 13: Interfaces have no dynamic effect.

$$\frac{< e \vdash Ct \Rightarrow cls > \quad n = \texttt{cl}(cls') \quad cls' \text{ fresh}}{\begin{array}{c} e \vdash << \texttt{abstract} >>\ \texttt{class}\ Tv\ L \triangleright I\ <\texttt{extends}\ Ct > \texttt{implements}\ [Nt]\ \texttt{of}\ T \\ \Rightarrow \{(I,n)\}, \{(L,n)\}, (\{< (cls', cls) >\}, \{||\}, \{<< cls' >>\}) \end{array}} \quad (14)$$

$$\frac{n = \texttt{v}(fun) \quad fun \text{ fresh}}{e \vdash \texttt{fun}\ Tv\ L \triangleright I : T \Rightarrow \{(I,n)\}, \{(L,n)\}, (\{\}, \{||\}, \{\})} \quad (15)$$

$$\frac{e \vdash F \Rightarrow fun \quad e \vdash Pat \Rightarrow Pat'}{\begin{array}{c} e \vdash \texttt{extend fun}\ F\ Pat = E \\ \Rightarrow \{\}, \{\}, (\{\}, \{|\ (fun, Pat', E, e)\ |\}, \{\}) \end{array}} \quad (16)$$

2.8 Class Types

$\boxed{e \vdash Ct \Rightarrow cls}$

$$\frac{e(I) = \texttt{cl}(cls)}{e \vdash Ts\ I \Rightarrow cls} \quad (17)$$

$$\frac{e \vdash Pn \Rightarrow m \quad m(L) = \texttt{cl}(cls)}{e \vdash Ts\ Pn.L \Rightarrow cls} \quad (18)$$

2.9 Patterns

$\boxed{e \vdash Pat \Rightarrow Pat'}$

These rules evaluate all the path names (representing classes) within the given pattern.

$$\frac{}{e \vdash _ \Rightarrow _} \quad (19)$$

$$\frac{e \vdash Pat \Rightarrow Pat'}{e \vdash (I\ \texttt{as}\ Pat) \Rightarrow (I\ \texttt{as}\ Pat')} \quad (20)$$

$$\frac{e \vdash Pat_1 \Rightarrow Pat'_1 \quad \cdots \quad e \vdash Pat_n \Rightarrow Pat'_n}{e \vdash \{L_1 = Pat_1, \ldots, L_n = Pat_n\} \Rightarrow \{L_1 = Pat'_1, \ldots, L_n = Pat'_n\}} \quad (21)$$

$$\frac{e \vdash Ct \Rightarrow cls \quad e \vdash Pat \Rightarrow Pat'}{e \vdash (Tv\ Ct\ Pat) \Rightarrow (cls\ Pat')} \quad (22)$$

2.10 Expressions

$\boxed{e, s \vdash E \Rightarrow v}$

$$\frac{e(I) = \mathtt{v}(v)}{e, s \vdash I \Rightarrow v} \quad (23)$$

$$\frac{e \vdash Pn \Rightarrow m \quad m(L) = \mathtt{v}(v)}{e, s \vdash Pn.L \Rightarrow v} \quad (24)$$

$$\frac{e \vdash Ct \Rightarrow cls \quad e, s \vdash E \Rightarrow v \quad s = (i, f, a) \quad cls \notin a}{e, s \vdash Ct\ E \Rightarrow cls\ v} \quad (25)$$

Rule 25: Instantiation succeeds only if the specified class is not abstract.

$$\frac{e, s \vdash E_1 \Rightarrow v_1 \quad \cdots \quad e, s \vdash E_n \Rightarrow v_n}{e, s \vdash \{L_1 = E_1, \ldots, L_n = E_n\} \Rightarrow \{L_1 = v_1, \ldots, L_n = v_n\}} \quad (26)$$

$$\frac{\begin{array}{c} e, s \vdash E_1 \Rightarrow fun \quad e, s \vdash E_2 \Rightarrow v_2 \\ s \vdash \text{most-specific-applicable-case-for } (fun, v_2) \Rightarrow (fun, Pat, E, e_1) \\ s = (i, f, a) \quad i, v_2 \vdash Pat \Rightarrow e_2 \quad e_1 \cup e_2, s \vdash E \Rightarrow v \end{array}}{e, s \vdash E_1\ E_2 \Rightarrow v} \quad (27)$$

2.11 Function Application

2.11.1 Top-level Judgements

$\boxed{s \vdash \text{most-specific-applicable-case-for } (fun, v) \Rightarrow l}$

$$\frac{\begin{array}{c} \{|\ l_1, \ldots, l_n\ |\} = \{|\ l \mid l \in f \wedge i \vdash l \text{ applies-to } (fun, v)\ |\} \\ \{|\ l_0\ |\} = \{|\ l \mid l \in \{|\ l_1, \ldots, l_n\ |\} \wedge \forall 1 \leq k \leq n.\ i \vdash l \leq l_k\ |\} \end{array}}{(i, f, a) \vdash \text{most-specific-applicable-case-for } (fun, v) \Rightarrow l_0} \quad (28)$$

Rule 28: The most-specific applicable function case is the unique *applicable* case that is at least as specific as all other applicable cases.

$\boxed{i \vdash l \text{ applies-to } (fun, v)}$

$$\frac{i, v \vdash Pat \Rightarrow e'}{i \vdash (fun, Pat, E, e) \text{ applies-to } (fun, v)} \quad (29)$$

$\boxed{i \vdash l \leq l'}$

$$\frac{i \vdash Pat \leq Pat'}{i \vdash (fun, Pat, E, e) \leq (fun, Pat', E', e')} \quad (30)$$

2.11.2 Pattern Matching

$\boxed{i, v \vdash Pat \Rightarrow e}$

$$\frac{}{i, v \vdash _ \Rightarrow \{\}} \quad (31)$$

$$\frac{i, v \vdash Pat \Rightarrow e}{i, v \vdash (I \texttt{ as } Pat) \Rightarrow e \cup \{(I, \mathtt{v}(v))\}} \quad (32)$$

$$\frac{i, v_1 \vdash Pat_1 \Rightarrow e_1 \quad \cdots \quad i, v_n \vdash Pat_n \Rightarrow e_n \quad n \leq p}{\begin{array}{c} i, \{L_1 = v_1, \ldots, L_p = v_p\} \vdash \{L_1 = Pat_1, \ldots, L_n = Pat_n\} \\ \Rightarrow e_1 \cup \cdots \cup e_n \end{array}} \quad (33)$$

Rule 33: The record value may have more components than the pattern. This allows subclass representations to match superclass representation patterns.

$$\frac{i \vdash cls \leq cls' \quad i, v \vdash Pat \Rightarrow e}{i, (cls\ v) \vdash (cls'\ Pat) \Rightarrow e} \quad (34)$$

2.11.3 Pattern Specificity

$\boxed{i \vdash Pat \leq Pat'}$

$$\frac{}{i \vdash Pat \leq _} \quad (35)$$

$$\frac{i \vdash Pat_1 \leq Pat_2}{i \vdash (I \texttt{ as } Pat_1) \leq Pat_2} \quad (36)$$

$$\frac{i \vdash Pat_1 \leq Pat_2}{i \vdash Pat_1 \leq (I \texttt{ as } Pat_2)} \quad (37)$$

$$\frac{i \vdash cls_1 < cls_2}{i \vdash cls_1\ Pat_1 \leq cls_2\ Pat_2} \quad (38)$$

Rule 38: If cls_1 is a strict subclass of cls_2, the associated representation patterns are irrelevant for specificity.

$$\frac{i \vdash Pat_1 \leq Pat_2}{i \vdash cls\ Pat_1 \leq cls\ Pat_2} \quad (39)$$

$$\frac{i \vdash Pat_1 \leq Pat'_1 \quad \cdots \quad i \vdash Pat_n \leq Pat'_n}{i \vdash \{L_1 = Pat_1, \ldots, L_n = Pat_n\} \leq \{L_1 = Pat'_1, \ldots, L_n = Pat'_n\}} \quad (40)$$

Rule 40: This rule is the essence of multimethod dispatching; all "arguments" are treated symmetrically.

2.11.4 Inheritance

$$\boxed{i \vdash cls \leq cls'}$$

$$\frac{}{i \vdash cls \leq cls} \ (41)$$

$$\frac{i \vdash cls < cls'}{i \vdash cls \leq cls'} \ (42)$$

$$\boxed{i \vdash cls < cls'}$$

$$\frac{(cls, cls') \in i}{i \vdash cls < cls'} \ (43)$$

$$\frac{i \vdash cls < cls'' \quad i \vdash cls'' < cls'}{i \vdash cls < cls'} \ (44)$$

3 Static Semantics

3.1 Preliminaries

This subsection provides auxiliary definitions for the static semantics, augmenting the definitions provided in section 2.1. The domains used in the static semantics is shown in figure 4. We use $\cup$ in the domain definitions to denote an (untagged) union. For a domain D, we use D^k as a shorthand for

$$\underbrace{D \times \cdots \times D}_{k \text{ times}}$$

and D^* as a shorthand for

$$\bigcup_{k \geq 0} D^k$$

It is useful to define $\in$ for tuples: $a \in (a_1, \ldots, a_n)$ if $a = a_i$ for some $1 \leq i \leq n$. The $+$ operator on tuples represents tuple concatenation:

$$(a_1, \ldots, a_n) + (b_1, \ldots, b_m) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$$

The TopTypeEnv maps identifiers to the static representation of signatures, signature blocks, and functors. The TypeEnv (resp., ModTypeEnv) maps identifiers (resp., labels) to the static representations of values, functions, classes, interfaces, structures, and blocks. It's useful for entries in the ModTypeEnv to be maintained in the order in which the associated declarations appear in the structure or block, so the ModTypeEnv is a tuple rather than a finite function. As in the dynamic semantics, we use a tagging scheme to distinguish among the elements of a TypeEnv (ModTypeEnv, TopTypeEnv). However, in the static semantics, the tags are represented as elements of the Tag semantic domain, thereby allowing them to be passed around in the rules.

Each block or signature block has a Context associated with it, consisting of several components:

- Locals is a set of the classes and functions introduced by the block.

$$
\begin{array}{rcl}
\Delta & \in & \text{TopTypeEnv} = I \to ((\text{ModTag} \times \text{ModRep}) \cup (\texttt{fnc} \times \text{Functor})) \\
\Gamma & \in & \text{TypeEnv} = I \to \text{StaticRep} \\
\nu & \in & \text{ModRep} = \text{ModTypeEnv} \times \text{Context} \\
 & & \text{Functor} = I \times \text{ModTag} \times \text{ModRep} \times \text{ModTag} \times \text{ModRep} \\
\eta & \in & \text{StaticRep} = ((\texttt{v} \cup \texttt{cl} \cup \texttt{ifc}) \times \text{PolyType}) \cup (\texttt{f} \times (Tv \times fun)) \cup (\text{ModTag} \times \text{ModRep}) \\
\mu & \in & \text{ModTypeEnv} = (L \times \text{StaticRep})^* \\
c & \in & \text{Context} = \text{Locals} \times \text{SInh} \times \text{Rep} \times \text{FType} \times \text{SCases} \times \text{Abstract} \times \text{Sealed} \times \text{Unknown} \\
 & & \text{Locals} = \{\kappa \cup fun\} \\
\iota & \in & \text{SInh} = \{(Tv \times \kappa) \times (Ts \times \kappa)\} \\
r & \in & \text{Rep} = cls \to \text{PolyType} \\
 & & \text{FType} = fun \to \text{PolyType} \\
 & & \text{SCases} = \{|\ \text{SCase}\ |\} \\
 & & \text{SCase} = fun \times Pat \times \text{TypeEnv} \times \text{Context} \\
a & \in & \text{Abstract} = \{cls\} \\
\theta & \in & \text{Sealed} = \{\kappa \cup fun\} \\
 & & \text{Unknown} = \{\kappa\} \\
\kappa & \in & \text{ClassOrIfc} = cls \cup ifc \\
pt & \in & \text{PolyType} = Tv \times T \\
it & \in & \text{IfcType} = Ts \times ifc \\
ct & \in & \text{ClassType} = Ts \times cls \\
 & & \text{Tag} = \texttt{v} \cup \texttt{f} \cup \texttt{cl} \cup \texttt{ifc} \cup \text{ModTag} \cup \texttt{fnc} \\
\pi & \in & \text{ModTag} = \texttt{str} \cup \texttt{bl} \\
\beta & \in & \text{3Bool} = \texttt{y} \cup \texttt{?}
\end{array}
$$

Figure 4: Domains for the static semantics.

- SInh records the superclasses and superinterfaces of each class and interface in the block.
- Rep records the representation type of each class in the block.
- FType records the type of each function in the block.
- SCases is a bag of function cases, each of which consists of the associated function, the case's pattern, the current lexical type environment, and the current static context. The last two items are used only for the soundness proof.
- Abstract contains the subset of classes in the block that are abstract.
- Sealed contains the subset of interfaces, classes, and functions in the block that are sealed.
- Unknown contains the classes and interfaces that were declared in the parameter signature of a functor; such classes and interfaces have unknown identity and must be dealt with specially.

We use a record-like notation for injecting into and projecting from Contexts, with the domain names as the labels. For example, $(\text{SInh} = \{((Tv, cls), (Ts, cls'))\}, \text{Abstract} = \{cls\})$ represents the Context whose second component is $\{(cls, cls')\}$, sixth component is $\{cls\}$, and other components are empty. Accordingly, $()$ represents the Context whose components are all empty. Similarly, the notation $\text{Abstract}(c)$ accesses the sixth component of the Context c.

When a rule refers to a simple semantic domain element (fun, cls, ifc), the rule is implicitly ensuring that the variable is from the associated domain. For example, a premise of the form $\Gamma, c, Tv \vdash Ct \Rightarrow (Ts, cls)$ is only satisfied if cls is in fact a class representative (rather than, say, an interface).

The domain 3Bool is used to implement a three-valued logic: `y` stands for "yes" and `?` stands for "don't know" (for a "no" answer, the associated judgement will simply not be derivable). We define a meet operator $\sqcap$ on 3Bools: $\beta_1 \sqcap \beta_2$ is `y` if $\beta_1 = \beta_2 =$ `y`, and `?` otherwise.

We define a function *refresh* for ModReps. The invocation $refresh(\nu)$, where $\nu = (\mu, c)$, returns the ModRep identical to ν, but with each element of $\text{Locals}(c)$ replaced throughout ν with a fresh element of the same domain (that is, a class is replaced with a fresh class, interface with a fresh interface, function with a fresh function). This function is used to create a new instantiation of a functor body, upon functor invocation.

We define a function *unknown* for Contexts. The invocation $unknown(c)$ returns the Context identical to c, but with $\text{Unknown}(c)$ augmented with $\text{Locals}(c)$. In other words, all of the Context's locals are now considered to be unknown. This is used on a Context that resulted from typechecking a functor parameter signature, so that the functor body can be conservatively typechecked.

We use the meta-variable d to range over elements of all semantic domains. The notation $d[d_1/d'_1, \ldots, d_n/d'_n]$ denotes the element d' formed by simultaneously substituting each d_i for each occurrence of d'_i in d.

3.2 Programs

$$\boxed{\vdash P : T \Rightarrow c}$$

$$\frac{\begin{array}{c}\{\}, \{\}, () \vdash M_1 \Rightarrow \Gamma_1, \Delta_1, c_1 \\ \vdots \\ \Gamma_1 \cup \ldots \cup \Gamma_{n-1}, \Delta_1 \cup \ldots \cup \Delta_{n-1}, c_1 \cup \ldots \cup c_{n-1} \vdash M_n \Rightarrow \Gamma_n, \Delta_n, c_n \\ c = c_1 \cup \ldots \cup c_n \quad \Gamma_1 \cup \ldots \cup \Gamma_n, c, () \vdash E : T\end{array}}{\vdash M_1, \ldots, M_n; E : T \Rightarrow c} \quad (45)$$

3.3 Module Expressions

$$\boxed{\Gamma, \Delta, c \vdash M \Rightarrow \Gamma', \Delta', c'}$$

$$\frac{\Gamma, \Delta, c \vdash Se \Rightarrow \pi, \nu, c'}{\Gamma, \Delta, c \vdash I = Se \Rightarrow \{(I, (\pi, \nu))\}, \{\}, c'} \quad (46)$$

Rule 46: The Context resulting from typechecking the structure or block is added to the global Context.

$$\frac{\Gamma, \Delta, c \vdash_{sig} Sig \Rightarrow \pi, \nu}{\Gamma, \Delta, c \vdash I = Sig \Rightarrow \{\}, \{(I, (\pi, \nu))\}, ()} \quad (47)$$

$$\frac{\begin{array}{c}\Gamma, \Delta, c \vdash_{sig} Sig \Rightarrow \pi_1, \nu_1 \quad \nu_1 = (\mu_1, c_1) \quad c_1' = unknown(c_1) \\ \Gamma \cup \{(I', (\pi_1, (\mu_1, c_1')))\}, \Delta, c \cup c_1' \vdash S \Rightarrow \pi, \nu_2\end{array}}{\Gamma, \Delta, c \vdash \texttt{functor}\ I(I' : Sig) = S \Rightarrow \{\}, \{(I, (\texttt{fnc}, (I', \pi_1, \nu_1, \pi, \nu_2)))\}, ()} \quad (48)$$

Rule 48: The parameter signature is typechecked, producing a ModTypeEnv and a Context. These are then added to the current information used to typecheck the functor body. The *unknown* function is used to signify that classes, interfaces, and functions declared in the parameter signature have unknown identity and must therefore be treated conservatively during typechecking of the functor body.

3.4 Structure and Block Expressions

$$\boxed{\Gamma, \Delta, c \vdash Se \Rightarrow \pi, \nu, c'}$$

$$\frac{\begin{array}{c}\Gamma, \Delta, c \vdash S \Rightarrow \pi, \nu \quad \nu = (\mu', c') \quad \Gamma, \Delta, c \vdash_{sig} Sig \Rightarrow \pi, \nu' \\ \Gamma, c, \pi \vdash \nu \text{ ascribes-to } \nu' \Rightarrow \nu'', sublst \quad \nu'' = (\mu'', c'')\end{array}}{\Gamma, \Delta, c \vdash S : Sig \Rightarrow \pi, \nu'', c' + (\text{Abstract} = \text{Abstract}(c''), \text{Sealed} = \text{Sealed}(c''))} \quad (49)$$

Rule 49: After typechecking the structure (or block) and the associated signature (or signature block), we check if the ascription is legal, producing the ModRep resulting from the ascription. The *sublst* component is used internally by the rules for signature ascription; it is a *substitution list* matching up the classes, interfaces, and functions declared in the structure (block) with those in the signature (signature block). To ease subject reduction, we return the original (unascribed) context c'. In particular, we don't "forget" hidden classes and functions. However, bindings for such classes and functions are not included in the resulting TypeEnv μ'', so they are still not accessible. We augment the context c' with the Abstract and Sealed components of the context resulting from ascription, to allow the ascription to seal classes and functions and to make some concrete classes viewed as abstract.

$$\frac{\begin{array}{c}\Delta(I) = (\texttt{fnc}, (I', \pi', \nu_1, \pi, \nu_2)) \quad \Gamma, \Delta, c \vdash Pn \Rightarrow \pi', \nu' \quad \nu' = (\mu', c') \\ \Gamma, c \cup c', \pi' \vdash \nu' \text{ functor-ascribes-to } \nu_1 \Rightarrow \nu'', ((d_1, d_1'), \ldots, (d_n, d_n')) \quad (\mu_0, c_0) = refresh(\nu_2[d_1/d_1', \ldots, d_n/d_n']) \\ \Gamma, c \cup c_0, c_0, \pi \vdash (\mu_0, c_0) \text{ functor-implementation-side-typecheck}\end{array}}{\Gamma, \Delta, c \vdash I(Pn) \Rightarrow \pi, (\mu_0, c_0), c_0} \quad (50)$$

Rule 50: Ascribing the functor argument ν' to the functor parameter signature ν_1 returns a substitution list which matches up the classes, interfaces, and functions of the actual parameter with those of the formal parameter. This substitution is applied to the functor body ν_2, and the substituted functor body is instantiated via the *refresh* function. Finally, any existing functions that are possibly affected by the functor body

must be typechecked for completeness and unambiguity (which we call *implementation-side* typechecking), to ensure type safety. Functions declared in the functor body itself are not checked; they are safely checked once, when the body is typechecked.

$$\boxed{\Gamma, \Delta, c \vdash S \Rightarrow \pi, \nu, c'}$$

$$\frac{\begin{array}{c}\Gamma, \Delta, c \vdash D_1 \Rightarrow \Gamma_1, \mu_1, c_1 \\ \cdots \\ \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_{n-1}, \Delta, c \cup c_1 \cup \ldots \cup c_{n-1} \vdash D_n \Rightarrow \Gamma_n, \mu_n, c_n \\ label(D_1), \ldots, label(D_n) \text{ distinct} \quad c' = c_1 \cup \ldots \cup c_n\end{array}}{\Gamma, \Delta, c \vdash \texttt{struct}\ D_1 \ldots D_n\ \texttt{end} \Rightarrow \texttt{str}, (\mu_1 + \ldots + \mu_n, c'), c'} \quad (51)$$

$$\frac{\begin{array}{c}\Gamma, c \vdash Ood_1 \Rightarrow \Gamma_1, \mu_1, c_1 \\ \cdots \\ \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_{n-1}, c \cup c_1 \cup \ldots \cup c_{n-1} \vdash Ood_n \Rightarrow \Gamma_n, \mu_n, c_n \\ label(Ood_1), \ldots, label(Ood_n) \text{ distinct} \\ c' = c_1 \cup \ldots \cup c_n \\ \Gamma, c \cup c', c' \vdash c' \text{ implementation-side-typecheck}\end{array}}{\Gamma, \Delta, c \vdash \texttt{block}\ Ood_1 \ldots Ood_n\ \texttt{end} \Rightarrow \texttt{bl}, (\mu_1 + \ldots + \mu_n, c'), c'} \quad (52)$$

Rule 52: The *implementation-side* typechecks ensure the completeness and unambiguity of all functions declared in blocks that the current block statically *depends upon*, according to the dependency relation in section 3.19.4.

3.5 Structure and Block Path Names

$$\boxed{\Gamma \vdash Pn \Rightarrow \mu}$$

$$\frac{\Gamma(I) = (\pi, (\mu, c))}{\Gamma \vdash I \Rightarrow \mu} \quad (53)$$

$$\frac{\Gamma \vdash Pn \Rightarrow \mu' \quad (L, (\pi, (\mu, c))) \in \mu'}{\Gamma \vdash Pn.L \Rightarrow \mu} \quad (54)$$

3.6 Signatures and Signature Blocks

$$\boxed{\Gamma, \Delta, c \vdash_{sig} Sig \Rightarrow \pi, \nu}$$

We use $\vdash_{sig}$ instead of $\vdash$ to cleanly separate the typechecking rules for specifications from the typechecking rules for declarations.

$$\frac{\Delta(I) = (\pi, \nu)}{\Gamma, \Delta, c \vdash_{sig} I \Rightarrow \pi, \nu} \quad (55)$$

$$\frac{\begin{array}{c}\Gamma, \Delta, c \vdash_{sig} Spec_1 \Rightarrow \Gamma_1, \mu_1, c_1 \\ \cdots \\ \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_{n-1}, \Delta, c \cup c_1 \cup \ldots \cup c_{n-1} \vdash_{sig} Spec_n \Rightarrow \Gamma_n, \mu_n, c_n \\ label(Spec_1), \ldots, label(Spec_n) \text{ distinct} \quad c' = c_1 \cup \ldots \cup c_n\end{array}}{\Gamma, \Delta, c \vdash_{sig} \texttt{sig}\ Spec_1 \ldots Spec_n\ \texttt{end} \Rightarrow \texttt{str}, (\mu_1 + \ldots + \mu_n, c')} \quad (56)$$

$$\frac{\begin{array}{c}\Gamma, c \vdash_{sig} \mathit{Oospec}_1 \Rightarrow \Gamma_1, \mu_1, c_1 \\ \cdots \\ \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_{n-1}, c \cup c_1 \cup \ldots \cup c_{n-1} \vdash_{sig} \mathit{Oospec}_n \Rightarrow \Gamma_n, \mu_n, c_n \\ \mathit{label}(\mathit{Oospec}_1), \ldots, \mathit{label}(\mathit{Oospec}_n) \text{ distinct} \\ c' = c_1 \cup \ldots \cup c_n\end{array}}{\Gamma, \Delta, c \vdash_{sig} \texttt{sigblock}\ \mathit{Oospec}_1 \ldots \mathit{Oospec}_n\ \texttt{end} \Rightarrow \texttt{bl}, (\mu_1 + \ldots + \mu_n, c')} \quad (57)$$

3.7 Declarations

$$\boxed{\Gamma, \Delta, c \vdash D \Rightarrow \Gamma', \mu, c'}$$

$$\frac{\Gamma, c, \mathit{Tv} \vdash E : T \quad \eta = (\texttt{v}, (\mathit{Tv}, T))}{\Gamma, \Delta, c \vdash \mathit{Tv}\ L \triangleright I = E \Rightarrow \{(I, \eta)\}, ((L, \eta)), ()} \quad (58)$$

$$\frac{\Gamma, \Delta, c \vdash \mathit{Se} \Rightarrow \pi, \nu \quad \eta = (\pi, \nu) \quad \nu = (\mu, c')}{\Gamma, \Delta, c \vdash L \triangleright I = \mathit{Se} \Rightarrow \{(I, \eta)\}, ((L, \eta)), c'} \quad (59)$$

3.8 OO Declarations

$$\boxed{\Gamma, c \vdash \mathit{Ood} \Rightarrow \Gamma', \mu, c'}$$

$$\frac{\begin{array}{c}\mathit{it} = (\mathit{Tv}, \mathit{ifc}) \quad \eta = (\texttt{ifc}, \mathit{it}) \quad \mathit{ifc} \text{ fresh} \\ \Gamma, c, \mathit{Tv} \vdash \mathit{Nt}_1 \Rightarrow (\mathit{Ts}_1, \mathit{ifc}_1) \quad \cdots \quad \Gamma, c, \mathit{Tv} \vdash \mathit{Nt}_n \Rightarrow (\mathit{Ts}_n, \mathit{ifc}_n) \\ \text{Sealed}(c) = \theta \quad \mathit{ifc}_1 \notin \theta \quad \cdots \quad \mathit{ifc}_n \notin \theta \\ c \vdash \{(\mathit{Ts}_1, \mathit{ifc}_1), \ldots, (\mathit{Ts}_n, \mathit{ifc}_n)\} \text{ consistent}\end{array}}{\begin{array}{l}\Gamma, c \vdash \texttt{interface}\ \mathit{Tv}\ L \triangleright I\ \texttt{extends}\ \mathit{Nt}_1, \ldots, \mathit{Nt}_n \Rightarrow \{(I, \eta)\}, ((L, \eta)), \\ (\text{Locals} = \{\mathit{ifc}\}, \text{SInh} = \{(\mathit{it}, (\mathit{Ts}_1, \mathit{ifc}_1)), \ldots, (\mathit{it}, (\mathit{Ts}_n, \mathit{ifc}_n))\})\end{array}} \quad (60)$$

$$\frac{\begin{array}{c}\mathit{ct} = (\mathit{Tv}, \mathit{cls}') \quad \eta = (\texttt{cl}, \mathit{ct}) \quad \mathit{cls}' \text{ fresh} \\ < \Gamma, c, \mathit{Tv} \vdash \mathit{Ct} \Rightarrow (\mathit{Ts}, \mathit{cls}) > \quad \text{Sealed}(c) = \theta \quad < \mathit{cls} \notin \theta > \quad < \mathit{Ts} = (T_1'', \ldots, T_r'') > \\ \Gamma, c, \mathit{Tv} \vdash \mathit{Nt}_1 \Rightarrow (\mathit{Ts}_1, \mathit{ifc}_1) \quad \cdots \quad \Gamma, c, \mathit{Tv} \vdash \mathit{Nt}_p \Rightarrow (\mathit{Ts}_p, \mathit{ifc}_p) \\ \mathit{ifc}_1 \notin \theta \quad \cdots \quad \mathit{ifc}_p \notin \theta \\ c \vdash \{< (\mathit{Ts}, \mathit{cls}), > (\mathit{Ts}_1, \mathit{ifc}_1), \ldots, (\mathit{Ts}_p, \mathit{ifc}_p)\} \text{ consistent} \\ < (\text{Rep}(c))(\mathit{cls}) = ((I_1, \ldots, I_r), T') > \quad < T'[T_1''/I_1, \ldots, T_r''/I_r] = \{L_1' : T_1', \ldots, L_m' : T_m'\} > \\ \Gamma, c, \mathit{Tv} \vdash T \Rightarrow \{L_1 : T_1, \ldots, L_n : T_n\} \\ < L_1', \ldots, L_m', > L_1, \ldots, L_n \text{ distinct} \\ T'' = \{< L_1' : T_1', \ldots, L_m' : T_m', > L_1 : T_1, \ldots, L_n : T_n\}\end{array}}{\begin{array}{l}\Gamma, c \vdash << \texttt{abstract} >>\ \texttt{class}\ \mathit{Tv}\ L \triangleright I < \texttt{extends}\ \mathit{Ct} > \texttt{implements}\ \mathit{Nt}_1, \ldots, \mathit{Nt}_p\ \texttt{of}\ T \Rightarrow \{(I, \eta)\}, ((L, \eta)), \\ (\text{Locals} = \{\mathit{cls}'\}, \text{SInh} = \{< (\mathit{ct}, (\mathit{Ts}, \mathit{cls})) >, (\mathit{ct}, (\mathit{Ts}_1, \mathit{ifc}_1)), \ldots, (\mathit{ct}, (\mathit{Ts}_p, \mathit{ifc}_p))\}, \\ \text{Rep} = \{(\mathit{cls}', (\mathit{Tv}, T''))\}, \text{Abstract} = \{<< \mathit{cls}' >>\})\end{array}} \quad (61)$$

$$\frac{\begin{array}{c}\Gamma, c, \mathit{Tv} \vdash T \Rightarrow T' \quad \eta = (\texttt{f}, (\mathit{Tv}, \mathit{fun})) \quad \mathit{fun} \text{ fresh} \\ T' = T_1 \xrightarrow{\mathit{Mark}} T_2 \quad \vdash \mathit{Mark} \text{ ok-for } T_1\end{array}}{\begin{array}{l}\Gamma, c \vdash \texttt{fun}\ \mathit{Tv}\ L \triangleright I : T \Rightarrow \{(I, \eta)\}, ((L, \eta)), \\ (\text{Locals} = \{\mathit{fun}\}, \text{FType} = \{(\mathit{fun}, (\mathit{Tv}, T'))\}\end{array}} \quad (62)$$

$$\frac{\Gamma \vdash F \Rightarrow fun \quad fun \notin \text{Sealed}(c) \quad (\text{FType}(c))(fun) = (Tv, T_1 \stackrel{Mark}{\longrightarrow} T_2) \quad \Gamma, c, T_1, Tv \vdash Pat \Rightarrow Pat', T_1', \Gamma', Tv' \quad \Gamma \cup \Gamma', c, Tv + Tv' \vdash E : T_2}{\Gamma, c \vdash \texttt{extend fun}\ F\ Pat = E \Rightarrow \{\}, (), (\text{SCases} = \{|\ (fun, Pat', \Gamma, c)\ |\})} \quad (63)$$

3.9 Specifications

$$\boxed{\Gamma, \Delta, c \vdash Spec \Rightarrow \Gamma', \mu, c'}$$

$$\frac{\Gamma, c, Tv \vdash T \Rightarrow T' \quad \eta = (\texttt{v}, (Tv, T'))}{\Gamma, \Delta, c \vdash_{sig} Tv\ L \triangleright I : T \Rightarrow \{(I, \eta)\}, ((L, \eta)), ()} \quad (64)$$

$$\frac{\Gamma, \Delta, c \vdash_{sig} Sig \Rightarrow \pi, \nu \quad \eta = (\pi, \nu) \quad \nu = (\mu, c')}{\Gamma, \Delta, c \vdash_{sig} L \triangleright I : Sig \Rightarrow \{(I, \eta)\}, ((L, \eta)), c'} \quad (65)$$

3.10 OO Specifications

$$\boxed{\Gamma, c \vdash Oospec \Rightarrow \Gamma', \mu, c'}$$

$$\frac{\begin{array}{c} it = (Tv, ifc) \quad \eta = (\texttt{ifc}, it) \quad ifc \text{ fresh} \\ \Gamma, c, Tv \vdash Nt_1 \Rightarrow (Ts_1, ifc_1) \quad \cdots \quad \Gamma, c, Tv \vdash Nt_n \Rightarrow (Ts_n, ifc_n) \\ \text{Sealed}(c) = \theta \quad ifc_1 \notin \theta \quad \cdots \quad ifc_n \notin \theta \\ c \vdash \{(Ts_1, ifc_1), \ldots, (Ts_n, ifc_n)\} \text{ consistent} \end{array}}{\begin{array}{l} \Gamma, c \vdash_{sig} \texttt{<< sealed >>}\ \texttt{interface}\ Tv\ L \triangleright I\ \texttt{extends}\ Nt_1, \ldots, Nt_n \Rightarrow \{(I, \eta)\}, ((L, \eta)), \\ (\text{Locals} = \{ifc\}, \text{SInh} = \{(it, (Ts_1, ifc_1)), \ldots, (it, (Ts_n, ifc_n))\}, \text{Sealed} = \{\texttt{<< } ifc \texttt{ >>}\}) \end{array}} \quad (66)$$

$$\frac{\begin{array}{c} ct = (Tv, cls') \quad \eta = (\texttt{cl}, ct) \quad cls' \text{ fresh} \\ < \Gamma, c, Tv \vdash Ct \Rightarrow (Ts, cls) > \quad \text{Sealed}(c) = \theta \quad < cls \notin \theta > \quad < Ts = (T_1'', \ldots, T_r'') > \\ \Gamma, c, Tv \vdash Nt_1 \Rightarrow (Ts_1, ifc_1) \quad \cdots \quad \Gamma, c, Tv \vdash Nt_p \Rightarrow (Ts_p, ifc_p) \\ ifc_1 \notin \theta \quad \cdots \quad ifc_p \notin \theta \\ c \vdash \{< (Ts, cls), > (Ts_1, ifc_1), \ldots, (Ts_p, ifc_p)\} \text{ consistent} \\ < (\text{Rep}(c))(cls) = ((I_1, \ldots, I_r), T') > \quad < T'[T_1''/I_1, \ldots, T_r''/I_r] = \{L_1' : T_1', \ldots, L_m' : T_m'\} > \\ \Gamma, c, Tv \vdash T \Rightarrow \{L_1 : T_1, \ldots, L_n : T_n\} \\ < L_1', \ldots, L_m', > L_1, \ldots, L_n \text{ distinct} \\ T'' = \{< L_1' : T_1', \ldots, L_m' : T_m', > L_1 : T_1, \ldots, L_n : T_n\} \end{array}}{\begin{array}{l} \Gamma, c \vdash_{sig} \texttt{<< sealed >>}\ \texttt{<<< abstract >>>}\ \texttt{class}\ Tv\ L \triangleright I\ \texttt{<extends}\ Ct > \texttt{implements}\ Nt_1, \ldots, Nt_p\ \texttt{of}\ T \\ \Rightarrow \{(I, \eta)\}, ((L, \eta)), (\text{Locals} = \{cls'\}, \text{SInh} = \{< (ct, (Ts, cls)) >, (ct, (Ts_1, ifc_1)), \ldots, (ct, (Ts_p, ifc_p))\}, \\ \text{Rep} = \{(cls', (Tv, T''))\}, \text{Abstract} = \{\texttt{<<< } cls' \texttt{ >>>}\}, \text{Sealed} = \{\texttt{<< } cls' \texttt{ >>}\}) \end{array}} \quad (67)$$

$$\frac{\begin{array}{c} \Gamma, c, Tv \vdash T \Rightarrow T' \quad \eta = (\texttt{f}, (Tv, fun)) \quad fun \text{ fresh} \\ T' = T_1 \stackrel{Mark}{\rightarrow} T_2 \quad \vdash Mark \text{ ok-for } T_1 \end{array}}{\begin{array}{l} \Gamma, c \vdash_{sig} \texttt{< sealed >}\ \texttt{fun}\ Tv\ L \triangleright I : T \Rightarrow \{(I, \eta)\}, ((L, \eta)), \\ (\text{Locals} = \{fun\}, \text{FType} = \{(fun, (Tv, T')\}, \text{Sealed} = \{< fun >\}) \end{array}} \quad (68)$$

$$\frac{\Gamma \vdash F \Rightarrow fun \quad fun \notin \text{Sealed}(c) \quad (\text{FType}(c))(fun) = (Tv, T_1 \overset{Mark}{\longrightarrow} T_2) \quad \Gamma, c, T_1, Tv \vdash Pat \Rightarrow Pat', T_1', \Gamma', Tv'}{\Gamma, c \vdash_{sig} \texttt{extend fun}\ F\ Pat \Rightarrow \{\}, (), (\text{SCases} = \{|\ (fun, Pat', \Gamma, c)\ |\})} \quad (69)$$

$$\frac{\Gamma, c, Tv \vdash T \Rightarrow T' \quad \eta = (\texttt{v}, (Tv, T'))}{\Gamma, c \vdash_{sig} \texttt{val}\ Tv\ L \triangleright I : T \Rightarrow \{(I, \eta)\}, ((L, \eta)), ()} \quad (70)$$

3.11 Types

$\boxed{\Gamma, c, Tv \vdash T \Rightarrow T'}$

$$\frac{(\text{Rep}(c))(cls) = ((I_1, \ldots, I_n), T) \quad \Gamma, c, Tv \vdash T_1' \Rightarrow T_1 \quad \cdots \quad \Gamma, c, Tv \vdash T_n' \Rightarrow T_n}{\Gamma, c, Tv \vdash cls \Rightarrow ((T_1, \ldots, T_n), cls)} \quad (71)$$

Rule 71: This rule is used only for the soundness proof.

$$\frac{I \in Tv}{\Gamma, c, Tv \vdash I \Rightarrow I} \quad (72)$$

$$\frac{\Gamma, c, Tv \vdash T_1' \Rightarrow T_1 \quad \cdots \quad \Gamma, c, Tv \vdash T_n' \Rightarrow T_n \quad \Gamma(I) = (\texttt{cl}, ((I_1, \ldots, I_n), cls))}{\Gamma, c, Tv \vdash ((T_1', \ldots, T_n'), I) \Rightarrow ((T_1, \ldots, T_n), cls)} \quad (73)$$

$$\frac{\Gamma, c, Tv \vdash T_1' \Rightarrow T_1 \quad \cdots \quad \Gamma, c, Tv \vdash T_n' \Rightarrow T_n \quad \Gamma(I) = (\texttt{ifc}, ((I_1, \ldots, I_n), ifc))}{\Gamma, c, Tv \vdash ((T_1', \ldots, T_n'), I) \Rightarrow ((T_1, \ldots, T_n), ifc)} \quad (74)$$

$$\frac{\Gamma, c, Tv \vdash T_1' \Rightarrow T_1 \quad \cdots \quad \Gamma, c, Tv \vdash T_n' \Rightarrow T_n \quad \Gamma \vdash Pn \Rightarrow \mu \quad (L, (\texttt{cl}, ((I_1, \ldots, I_n), cls))) \in \mu}{\Gamma, c, Tv \vdash ((T_1', \ldots, T_n'), Pn.L) \Rightarrow ((T_1, \ldots, T_n), cls)} \quad (75)$$

$$\frac{\Gamma, c, Tv \vdash T_1' \Rightarrow T_1 \quad \cdots \quad \Gamma, c, Tv \vdash T_n' \Rightarrow T_n \quad \Gamma \vdash Pn \Rightarrow \mu \quad (L, (\texttt{ifc}, ((I_1, \ldots, I_n), ifc))) \in \mu}{\Gamma, c, Tv \vdash ((T_1', \ldots, T_n'), Pn.L) \Rightarrow ((T_1, \ldots, T_n), ifc)} \quad (76)$$

$$\frac{\Gamma, c, Tv \vdash T_1 \Rightarrow T_1' \quad \cdots \quad \Gamma, c, Tv \vdash T_n \Rightarrow T_n'}{\Gamma, c, Tv \vdash \{L_1 : T_1, \ldots, L_n : T_n\} \Rightarrow \{L_1 : T_1', \ldots, L_n : T_n'\}} \quad (77)$$

$$\frac{\Gamma, c, Tv \vdash T_1 \Rightarrow T_1' \quad \Gamma, c, Tv \vdash T_2 \Rightarrow T_2'}{\Gamma, c, Tv \vdash T_1 \overset{Mark}{\rightarrow} T_2 \Rightarrow T_1' \overset{Mark}{\rightarrow} T_2'} \quad (78)$$

3.12 Subtyping

$\boxed{c \vdash T \leq_{sub} T'}$

Subtyping rules are used to provide type subsumption for expressions.

$$\frac{(((I_1, \ldots, I_n), \kappa_1), ((T_1'', \ldots, T_m''), \kappa_2)) \in \text{SInh}(c) \quad T_1''[T_1/I_1, \ldots, T_n/I_n] = T_1' \quad \cdots \quad T_m''[T_1/I_1, \ldots, T_n/I_n] = T_m'}{c \vdash ((T_1, \ldots, T_n), \kappa_1) \leq_{sub} ((T_1', \ldots, T_m'), \kappa_2)} \quad (79)$$

$$\frac{c \vdash T_1 \leq_{sub} T_1' \quad \cdots \quad c \vdash T_n \leq_{sub} T_n' \quad n \leq p}{c \vdash \{L_1 : T_1, \ldots, L_p : T_p\} \leq_{sub} \{L_1 : T_1', \ldots, L_n : T_n'\}} \quad (80)$$

$$\frac{c \vdash T_1' \leq_{sub} T_1 \quad c \vdash T_2 \leq_{sub} T_2'}{c \vdash T_1 \overset{Mark}{\rightarrow} T_2 \leq_{sub} T_1' \overset{Mark'}{\rightarrow} T_2'} \quad (81)$$

Rule 81: The marks are irrelevant because the function undergoing subsumption cannot have cases added to it, but rather can only be applied.

$$\frac{}{c \vdash T \leq_{sub} T} \quad (82)$$

$$\frac{c \vdash T \leq_{sub} T'' \quad c \vdash T'' \leq_{sub} T'}{c \vdash T \leq_{sub} T'} \quad (83)$$

3.13 Function Value Access

$\boxed{\Gamma \vdash F \Rightarrow fun}$

These rules are used when we need to access a generic function's value, rather than its type.

$$\frac{\Gamma(I) = (\texttt{f}, (Tv, fun))}{\Gamma \vdash I \Rightarrow fun} \quad (84)$$

$$\frac{\Gamma \vdash Pn \Rightarrow \mu \quad (L, (\texttt{f}, (Tv, fun))) \in \mu}{\Gamma \vdash Pn.L \Rightarrow fun} \quad (85)$$

3.14 Patterns

$\boxed{\Gamma, c, T, Tv \vdash Pat \Rightarrow Pat', T', \Gamma', Tv'}$

Typechecking a pattern returns four things: the "evaluated" version of the pattern, with all class names replaced by their internal representations; the precise type of values accepted by the pattern; a type environment for the bindings defined by the pattern; a tuple of the new type variables introduced by the pattern.

$$\frac{}{\Gamma, c, T, Tv \vdash _ \Rightarrow _, T, \{\}, ()} \quad (86)$$

$$\frac{\Gamma, c, T,\ Tv \vdash Pat \Rightarrow Pat', T', \Gamma',\ Tv'}{\Gamma, c, T,\ Tv \vdash I\ \texttt{as}\ Pat \Rightarrow I\ \texttt{as}\ Pat', T', \Gamma' \cup \{(I, (\texttt{v}, ((), T')))\},\ Tv'} \quad (87)$$

$$\frac{\begin{array}{c}\Gamma, c, T_1,\ Tv \vdash Pat_1 \Rightarrow Pat'_1, T'_1, \Gamma_1,\ Tv_1 \\ \cdots \\ \Gamma, c, T_n,\ Tv \vdash Pat_n \Rightarrow Pat'_n, T'_n, \Gamma_n,\ Tv_n\end{array}}{\begin{array}{c}\Gamma, c, \{L_1 : T_1, \ldots, L_n : T_n\},\ Tv \vdash \{L_1 = Pat_1, \ldots, L_n = Pat_n\} \\ \Rightarrow \{L_1 = Pat'_1, \ldots, L_n = Pat'_n\}, \{L_1 : T'_1, \ldots, L_n : T'_n\}, (\Gamma_1 \cup \ldots \cup \Gamma_n), (Tv_1 + \ldots + Tv_n)\end{array}} \quad (88)$$

$$\frac{\begin{array}{c}\Gamma, c,\ Tv + Tv' \vdash Ct \Rightarrow (Ts, cls') \quad c \vdash (Ts, cls') \leq_{sub} T \quad Ts = (T_1, \ldots, T_n) \\ Tv' = (I'_1, \ldots, I'_m) \quad c, \{I'_1, \ldots, I'_m\} \vdash (Ts, cls') \text{ is-safe-for } T \\ (\text{Rep}(c))(cls') = ((I_1, \ldots, I_n), T') \quad \Gamma, c, T'[T_1/I_1, \ldots, T_n/I_n],\ Tv + Tv' \vdash Pat \Rightarrow Pat', T'', \Gamma',\ Tv''\end{array}}{\Gamma, c, T,\ Tv \vdash\ Tv'\ \ Ct\ Pat \Rightarrow cls'\ Pat', (Ts, cls'), \Gamma',\ Tv' + Tv''} \quad (89)$$

Rule 89: The returned pattern is stripped of its type variables and type parameters. These are not used during implementation-side typechecking.

3.15 Polymorphism Restrictions

This section describes two restrictions on parametric polymorphism, in order to safely allow implementation-side typechecking to ignore polymorphism altogether.

$$\boxed{c \vdash \{(Ts, \kappa)\} \text{ consistent}}$$

This rule ensures that each instantiation of a class or interface only subtypes (possibly transitively) from a single instantiation of any other given class or interface. For example, class C_1 (with no type parameters) may not subtype from both (bool) C_2 and (int) C_2. It is only possible to violate this restriction in the presence of multiple inheritance, which is supported by interfaces.

$$\frac{\forall 1 \leq i \leq n. \forall 1 \leq j \leq n. (c \vdash (Ts_i, \kappa_i) \leq_{sub} (Ts, \kappa) \wedge c \vdash (Ts_j, \kappa_j) \leq_{sub} (Ts', \kappa) \Rightarrow Ts' = Ts)}{c \vdash \{(Ts_1, \kappa_1), \ldots, (Ts_n, \kappa_n)\} \text{ consistent}} \quad (90)$$

$$\boxed{c, \{I\} \vdash T \text{ is-safe-for } T'}$$

These rules check that a class pattern does not attempt to "dispatch" upon any of its type parameters. As a simple example, suppose (a) Foo subclasses from Bar, and suppose that function f has an argument type of Bar. Then (b) (b) Foo _ is a legal class pattern for a function case of f, as its type parameter is fully general. On the other hand, (bool) Foo _ is not a legal class pattern, because it "dispatches" on its type parameter.

In the rules below, the meta-variable S represents a set of identifiers.

$$\frac{}{c, S \vdash T \text{ is-safe-for } T} \quad (91)$$

$$\frac{(((I_1,\ldots,I_n),\kappa),T') \in \mathrm{SInh}(c) \quad T'' = T'[T_1/I_1,\ldots,T_n/I_n] \quad S \vdash (I_1,T_1)\text{ is-safe-for }T' \Rightarrow S_1 \quad \cdots \quad S-(S_1\cup\ldots\cup S_{n-1}) \vdash (I_n,T_n)\text{ is-safe-for }T' \Rightarrow S_n \quad c, S-(S_1\cup\ldots\cup S_n) \vdash T''\text{ is-safe-for }T}{c, S \vdash ((T_1,\ldots,T_n),\kappa)\text{ is-safe-for }T} \quad (92)$$

Rule 92: We check that the type parameters $T_1,\ldots,T_n$ are a general enough instantiation of the direct superclass T'; all "extra" type parameters are fully general. We then check that the instantiated direct superclass is recursively safe for the argument type T. Any extra type variables are thrown out as they are "used" by the type parameters, to ensure that each is used at most once.

$$\boxed{S \vdash (I,T)\text{ is-safe-for }T' \Rightarrow S'}$$

$$\frac{I\text{ is referred to in }T'}{S \vdash (I,T)\text{ is-safe-for }T' \Rightarrow \{\}} \quad (93)$$

Rule 93: If T' refers to I, then this type parameter is not an "extra" one; it is needed in order for κ (from the previous rule) to subclass from T'. Therefore, the associated type parameter T will be checked by the recursive check on T' in the previous rule.

$$\frac{\text{the previous rule does not apply} \quad I' \in S}{S \vdash (I,I')\text{ is-safe-for }T \Rightarrow \{I'\}} \quad (94)$$

Rule 94: Any extra type parameter of a subclass must be a new type variable that has not previously been used as a type parameter. Once this "use" is seen, the type variable is removed from our set S of allowable extra type variables, so that it cannot be used again.

3.16 Expressions

$$\boxed{\Gamma, c, Tv \vdash E : T}$$

$$\frac{\Gamma, c, Tv \vdash T_1 \Rightarrow T_1' \quad \cdots \quad \Gamma, c, Tv \vdash T_1 \Rightarrow T_n' \quad \mathrm{FType}(c)(fun) = ((I_1,\ldots,I_n),T)}{\Gamma, c, Tv \vdash fun : T[T_1'/I_1,\ldots,T_n'/I_n]} \quad (95)$$

Rule 95: This rule and the next one are used only in the soundness proof.

$$\frac{\Gamma, c, Tv \vdash cls \Rightarrow ((T_1,\ldots,T_n),cls) \quad (\mathrm{Rep}(c))(cls) = ((I_1,\ldots,I_n),T) \quad \Gamma, c, Tv \vdash E : T[T_1/I_1,\ldots,T_n/I_n]}{\Gamma, c, Tv \vdash cls\ E : ((T_1,\ldots,T_n),cls)} \quad (96)$$

Rule 96: This rule is identical to the ordinary rule for class instantiation, except that we do not check that the class is not abstract. This allows the subject reduction proof to go through. In particular, it may be the case that the *cls* is not abstract but is viewed as abstract in the current context, through ascription. Therefore, we would not be able to show type preservation on values of the form *cls v* if we required *cls* to be non-abstract in the current context.

$$\frac{\Gamma, c, Tv \vdash T_1 \Rightarrow T_1' \quad \cdots \quad \Gamma, c, Tv \vdash T_1 \Rightarrow T_n' \quad \Gamma(I) = (\mathtt{v},((I_1,\ldots,I_n),T))}{\Gamma, c, Tv \vdash I : T[T_1'/I_1,\ldots,T_n'/I_n]} \quad (97)$$

Rule 97: In this and the next few rules, we non-deterministically choose types $T_1, \ldots, T_n$ with which to instantiate the polymorphic type.

$$\frac{\Gamma, c, Tv \vdash T_1 \Rightarrow T_1' \quad \cdots \quad \Gamma, c, Tv \vdash T_1 \Rightarrow T_n' \quad \Gamma(I) = (\mathtt{f}, ((I_1, \ldots, I_n), fun)) \quad (\text{FType}(c))(fun) = ((I_1, \ldots, I_n), T)}{\Gamma, c, Tv \vdash I : T[T_1'/I_1, \ldots, T_n'/I_n]} \quad (98)$$

$$\frac{\Gamma, c, Tv \vdash T_1 \Rightarrow T_1' \quad \cdots \quad \Gamma, c, Tv \vdash T_1 \Rightarrow T_n' \quad \Gamma \vdash Pn \Rightarrow \mu \quad (L, (\mathtt{v}, ((I_1, \ldots, I_n), T))) \in \mu}{\Gamma, c, Tv \vdash Pn.L : T[T_1'/I_1, \ldots, T_n'/I_n]} \quad (99)$$

$$\frac{\Gamma, c, Tv \vdash T_1 \Rightarrow T_1' \quad \cdots \quad \Gamma, c, Tv \vdash T_1 \Rightarrow T_n' \quad \Gamma \vdash Pn \Rightarrow \mu \quad (L, (\mathtt{f}, ((I_1, \ldots, I_n), fun))) \in \mu \quad \text{FType}(c)(fun) = ((I_1, \ldots, I_n), T)}{\Gamma, c, Tv \vdash Pn.L : T[T_1'/I_1, \ldots, T_n'/I_n]} \quad (100)$$

$$\frac{\Gamma, c, Tv \vdash\ Ct \Rightarrow ((T_1, \ldots, T_n), cls) \quad (\text{Rep}(c))(cls) = ((I_1, \ldots, I_n), T) \quad \Gamma, c, Tv \vdash E : T[T_1/I_1, \ldots, T_n/I_n] \quad cls \notin \text{Abstract}(c)}{\Gamma, c, Tv \vdash\ Ct\ E : ((T_1, \ldots, T_n), cls)} \quad (101)$$

$$\frac{\Gamma, c, Tv \vdash E_1 : T_1 \quad \cdots \quad \Gamma, c, Tv \vdash E_n : T_n}{\Gamma, c, Tv \vdash \{L_1 = E_1, \ldots, L_n = E_n\} : \{L_1 : T_1, \ldots, L_n : T_n\}} \quad (102)$$

$$\frac{\Gamma, c, Tv \vdash E_1 : T_2 \xrightarrow{Mark} T \quad \Gamma, c, Tv \vdash E_2 : T_2}{\Gamma, c, Tv \vdash E_1\ E_2 : T} \quad (103)$$

$$\frac{\Gamma, c, Tv \vdash E : T' \quad c \vdash T' \leq_{sub} T}{\Gamma, c, Tv \vdash E : T} \quad (104)$$

3.17 Signature Ascription

$$\boxed{\Gamma, c, \pi \vdash \nu \text{ ascribes-to } \nu' \Rightarrow \nu'', sublst}$$

Rule 104: ν is the ModRep of the structure or block being ascribed and ν' is the ModRep of the signature or signature block being ascribed to. The judgement returns a ModRep representing the ascribed structure, as well as a substitution list matching up the classes, interfaces, and functions declared in the structure (block) with those in the signature (signature block).

$$\frac{\Gamma, c \vdash \nu_1 \text{ bl-ascribes-to } \nu_2 \Rightarrow (\mu', c'), sublst \quad \nu_1 = (\mu_1, c_1) \quad c' - c_1 = (\{\}, \{\}, \{\}, \{\}, \{||\}, a, \theta, \{\})}{\Gamma, c, \mathtt{bl} \vdash \nu_1 \text{ ascribes-to } \nu_2 \Rightarrow (\mu', c'), sublst} \quad (105)$$

Rule 105: The condition on $c' - c_1$ ensures that the signature block's Context is a subset of the block's Context, except that the signature block may declare more things `abstract` than the block does and may declare some things `sealed`.

$$\frac{\Gamma, c \vdash \nu_1 \text{ str-ascribes-to } \nu_2 \Rightarrow \nu, sublst}{\Gamma, c, \mathtt{str} \vdash \nu_1 \text{ ascribes-to } \nu_2 \Rightarrow \nu, sublst} \quad (106)$$

$$\boxed{\Gamma, c \vdash \nu \text{ bl-ascribes-to } \nu' \Rightarrow \nu'', sublst}$$

$$\frac{\begin{array}{c}\nu_1 = (\mu_1, c_1) \quad \nu_2 = (\mu_2, c_2) \\ c \cup c_1 \vdash \mu_1 \text{ block-ascribes-to } \mu_2 \Rightarrow ((d_1, d_1'), \ldots, (d_n, d_n')) \quad \nu_2[d_1/d_1', \ldots, d_n/d_n'] = (\mu_0, c_0) \\ c_1 - c_0 = c' \quad \Gamma, c, c_1 \vdash c' \text{ implementation-side-typecheck}\end{array}}{\Gamma, c \vdash \nu_1 \text{ bl-ascribes-to } \nu_2 \Rightarrow (\mu_0, c_0), ((d_1, d_1'), \ldots, (d_n, d_n'))} \quad (107)$$

Rule 107: The hidden part of the block's Context, represented by c', must satisfy implementation-side typechecks as if it were declared as a separate block, to ensure the safety of hiding it.

$$\boxed{\Gamma, c \vdash \nu \text{ str-ascribes-to } \nu' \Rightarrow \nu'', sublst}$$

$$\frac{\begin{array}{c}\nu_1 = (\mu_1, c_1) \quad \nu_2 = (\mu_2, c_2) \\ \Gamma, c \cup c_1 \vdash \mu_1 \text{ struct-ascribes-to } \mu_2 \Rightarrow \nu, sublst\end{array}}{\Gamma, c \vdash \nu_1 \text{ str-ascribes-to } \nu_2 \Rightarrow \nu, sublst} \quad (108)$$

Rule 108: We do not make use of the Context components of the signature and structure; they are checked indirectly through the checks on μ_1 and μ_2.

$$\boxed{c \vdash \mu_1 \text{ block-ascribes-to } \mu_2 \Rightarrow ((\kappa \cup fun \cup I) \times (\kappa \cup fun \cup I))^*}$$

These rules are used to match up the ModTypeEnv of the signature block with that of the block. They each must map a given label L to the same kind of thing – class, interface, or function – with the same number of type variables. The signature block may optionally omit some labels. The rules return a *substitution list*, which represents the correspondence between a class (interface, function, type variable) declared in the block and an associated class (interface, function, type variable) declared in the signature block.

$$\frac{}{c \vdash () \text{ block-ascribes-to } () \Rightarrow ()} \quad (109)$$

$$\frac{\begin{array}{c}c \vdash \mu_1 \text{ block-ascribes-to } \mu_2 \Rightarrow sublst \\ c \vdash \mu_1 + ((L, (\texttt{cl}, ((I_1, \ldots, I_n), cls)))) \text{ block-ascribes-to } \mu_2 + ((L, (\texttt{cl}, ((I_1', \ldots, I_n'), cls')))) \\ \Rightarrow sublst + ((I_1, I_1'), \ldots, (I_n, I_n'), (cls, cls'))\end{array}}{} \quad (110)$$

$$\frac{\begin{array}{c}c \vdash \mu_1 \text{ block-ascribes-to } \mu_2 \Rightarrow sublst \\ c \vdash \mu_1 + ((L, (\texttt{ifc}, ((I_1, \ldots, I_n), ifc)))) \text{ block-ascribes-to } \mu_2 + ((L, (\texttt{ifc}, ((I_1', \ldots, I_n'), ifc')))) \\ \Rightarrow sublst + ((I_1, I_1'), \ldots, (I_n, I_n'), (ifc, ifc'))\end{array}}{} \quad (111)$$

$$\frac{\begin{array}{c}c \vdash \mu_1 \text{ block-ascribes-to } \mu_2 \Rightarrow sublst \\ c \vdash \mu_1 + ((L, (\texttt{f}, ((I_1, \ldots, I_n), fun)))) \text{ block-ascribes-to } \mu_2 + ((L, (\texttt{f}, ((I_1', \ldots, I_n'), fun')))) \\ \Rightarrow sublst + ((I_1, I_1'), \ldots, (I_n, I_n'), (fun, fun'))\end{array}}{} \quad (112)$$

$$\frac{\begin{array}{c}c \vdash \mu_1 \text{ block-ascribes-to } \mu_2 \Rightarrow sublst \quad sublst = ((d_1, d_1'), \ldots (d_m, d_m')) \\ \text{FType}(c)(fun) = ((I_1, \ldots, I_n), T) \quad c \vdash T \leq_{sub} T'[d_1/d_1', \ldots, d_m/d_m'][I_1/I_1', \ldots, I_n/I_n'] \\ c \vdash \mu_1 + ((L, (\texttt{f}, ((I_1, \ldots, I_n), fun)))) \text{ block-ascribes-to } \mu_2 + ((L, (\texttt{v}, ((I_1', \ldots, I_n'), T')))) \\ \Rightarrow sublst + ((I_1, I_1'), \ldots, (I_n, I_n'))\end{array}}{} \quad (113)$$

Rule 113: The signature block may view a function as an ordinary value. In this case, the function must have a subtype of the value's specified type.

$$\frac{c \vdash \mu_1 \text{ block-ascribes-to } \mu_2 \Rightarrow sublst}{c \vdash \mu_1 + ((L, \eta)) \text{ block-ascribes-to } \mu_2 \Rightarrow sublst} \quad (114)$$

Rule 114: The signature block may hide a component.

$$\boxed{\Gamma, c \vdash \mu_1 \text{ struct-ascribes-to } \mu_2 \Rightarrow \nu, sublst}$$

$$\frac{}{\Gamma, c \vdash () \text{ struct-ascribes-to } () \Rightarrow (\{\}, ()), ()} \quad (115)$$

$$\frac{\Gamma, c \vdash \mu_1 \text{ struct-ascribes-to } \mu_2 \Rightarrow \nu, sublst}{\Gamma, c \vdash \mu_1 + ((L, (\mathtt{v}, pt))) \text{ struct-ascribes-to } \mu_2 \Rightarrow \nu, sublst} \quad (116)$$

Rule 116: The signature may hide a value.

$$\frac{\begin{array}{c}\Gamma, c \vdash \mu_1 \text{ struct-ascribes-to } \mu_2 \Rightarrow (\mu_0, c_0), sublst \quad sublst = ((d_1, d_1'), \ldots, (d_m, d_m')) \\ T'' = T'[d_1/d_1', \ldots, d_m/d_m'][I_1/I_1', \ldots, I_n/I_n'] \quad c \vdash T \leq_{sub} T''\end{array}}{\begin{array}{c}\Gamma, c \vdash \mu_1 + ((L, (\mathtt{v}, ((I_1, \ldots, I_n), T)))) \text{ struct-ascribes-to } \mu_2 + ((L, (\mathtt{v}, ((I_1', \ldots, I_n'), T')))) \\ \Rightarrow (((L, (\mathtt{v}, ((I_1, \ldots, I_n), T'')))) + \mu_0, c_0), sublst + ((I_1, I_1'), \ldots, (I_n, I_n'))\end{array}} \quad (117)$$

Rule 117: The signature may give a value a supertype of its actual type.

$$\frac{\begin{array}{c}\Gamma, c \cup c_0 \vdash \mu_1 \text{ struct-ascribes-to } \mu_2 \Rightarrow (\mu_3, c_3), ((d_1, d_1'), \ldots, (d_n, d_n')) \\ \Gamma, c, \pi \vdash \nu \text{ ascribes-to } \nu'[d_1/d_1', \ldots, d_n/d_n'] \Rightarrow \nu_0, sublst \quad \nu_0 = (\mu_0, c_0)\end{array}}{\begin{array}{c}\Gamma, c \vdash \mu_1 + ((L, (\pi, \nu))) \text{ struct-ascribes-to } \mu_2 + ((L, (\pi, \nu'))) \\ \Rightarrow (\mu_3 + ((L, (\pi, \nu_0))), c_3 \cup c_0), ((d_1, d_1'), \ldots, (d_n, d_n')) + sublst\end{array}} \quad (118)$$

Rule 118: We recursively ascribe structure and block components to the corresponding signature and signature block components. The Context resulting from the component ascription is added to the context for the overall ascription.

3.18 Functor Matching

$$\boxed{\Gamma, c, \pi \vdash \nu \text{ functor-ascribes-to } \nu' \Rightarrow \nu'', sublst}$$

$$\frac{\begin{array}{c}\Gamma, c \vdash \nu_1 \text{ bl-ascribes-to } \nu_2 \Rightarrow (\mu', c'), sublst \quad \nu_1 = (\mu_1, c_1) \quad \text{FType}(c') = \{\} \quad \text{SCases}(c') = \{||\} \\ c' - c_1 = (\{\}, \iota, \{\}, \{\}, \{||\}, a, \theta, \{\}) \quad \iota = \{(pt_1, T_1), \ldots, (pt_n, T_n)\} \\ \iota' = \text{SInh}(c_1) \quad (pt_1, T_1') \in \iota' \quad \cdots \quad (pt_n, T_n') \in \iota' \quad c_1 \vdash T_1' \leq_{sub} T_1 \quad \cdots \quad c_1 \vdash T_n' \leq_{sub} T_n\end{array}}{\Gamma, c, \mathtt{bl} \vdash \nu_1 \text{ functor-ascribes-to } \nu_2 \Rightarrow \nu, sublst} \quad (119)$$

Rule 119: We require the signature block to have no `fun` or `extend fun` specifications. In addition to allowing the signature block to declare more things abstract and sealed, as in ordinary ascription, we also allow the signature block to specify an indirect superclass/superinterface for a class/interface.

$$\frac{\Gamma, c \vdash \mu_1 \text{ fun-struct-ascribes-to } \mu_2 \Rightarrow \nu, sublst}{\Gamma, c, \mathtt{str} \vdash (\mu_1, c_1) \text{ functor-ascribes-to } (\mu_2, c_2) \Rightarrow \nu, sublst} \quad (120)$$

$\boxed{\Gamma, c \vdash \mu_1 \text{ fun-struct-ascribes-to } \mu_2 \Rightarrow \nu, sublst}$

These rules are the same as the struct-ascribes-to rules above, except that the premises use the functor versions of the inference rules.

$$\frac{}{\Gamma, c \vdash () \text{ fun-struct-ascribes-to } () \Rightarrow (\{\}, ()), ()} \quad (121)$$

$$\frac{\Gamma, c \vdash \mu_1 \text{ fun-struct-ascribes-to } \mu_2 \Rightarrow \nu, sublst}{\Gamma, c \vdash \mu_1 + ((L, (\mathtt{v}, pt))) \text{ fun-struct-ascribes-to } \mu_2 \Rightarrow \nu, sublst} \quad (122)$$

Rule 122: The signature may hide a value.

$$\frac{\begin{array}{c}\Gamma, c \vdash \mu_1 \text{ fun-struct-ascribes-to } \mu_2 \Rightarrow (\mu_0, c_0), sublst \quad sublst = ((d_1, d_1'), \ldots, (d_m, d_m')) \\ T'' = T'[d_1/d_1', \ldots, d_m/d_m'][I_1/I_1', \ldots, I_n/I_n'] \quad c \vdash T \leq_{sub} T''\end{array}}{\begin{array}{c}\Gamma, c \vdash \mu_1 + ((L, (\mathtt{v}, ((I_1, \ldots, I_n), T)))) \text{ fun-struct-ascribes-to } \mu_2 + ((L, (\mathtt{v}, ((I_1', \ldots, I_n'), T')))) \\ \Rightarrow (((L, (\mathtt{v}, ((I_1, \ldots, I_n), T'')))) + \mu_0, c_0), sublst + ((I_1, I_1'), \ldots, (I_n, I_n'))\end{array}} \quad (123)$$

Rule 123: The signature may give a value a supertype of its actual type.

$$\frac{\begin{array}{c}\Gamma, c \cup c_0 \vdash \mu_1 \text{ fun-struct-ascribes-to } \mu_2 \Rightarrow (\mu_3, c_3), ((d_1, d_1'), \ldots, (d_n, d_n')) \\ \Gamma, c, \pi \vdash \nu \text{ functor-ascribes-to } \nu'[d_1/d_1', \ldots, d_n/d_n'] \Rightarrow \nu_0, sublst \quad \nu_0 = (\mu_0, c_0)\end{array}}{\begin{array}{c}\Gamma, c \vdash \mu_1 + ((L, (\pi, \nu))) \text{ fun-struct-ascribes-to } \mu_2 + ((L, (\pi, \nu'))) \\ \Rightarrow (\mu_3 + ((L, (\pi, \nu_0))), c_3 \cup c_0), ((d_1, d_1'), \ldots, (d_n, d_n')) + sublst\end{array}} \quad (124)$$

3.19 Implementation-side Typechecking (ITC)

3.19.1 Mark consistency

$\boxed{\vdash \mathit{Mark} \text{ ok-for } T}$

These rules check that the *Mark* of a function identifies an argument position within the function's argument type T that expects a class instance. The associated position is known as the *distinguished position* (DP) and is used in implementing our modular implementation-side typechecking restrictions.

$$\frac{}{\vdash \emptyset \text{ ok-for } (Ts, \kappa)} \quad (125)$$

$$\frac{\vdash \mathit{Mark} \text{ ok-for } T_i \quad 1 \leq i \leq n}{\vdash (L_i, \mathit{Mark}) \text{ ok-for } \{L_1 : T_1, \ldots, L_n : T_n\}} \quad (126)$$

3.19.2 Top-level ITC

$\boxed{\Gamma, c_0, c'' \vdash c \text{ implementation-side-typecheck}}$

Context c is the block being checked, c_0 is the global program Context up to and including the block being checked. c'' is ordinarily c itself, which is then included as part of c_d, the Context representing all depended-upon blocks. However, when ITC is used to check a hidden part of a block, c is the hidden part, while c'' is the entire block. This is done for simplicity, rather than checking explicitly whether the public part is depended upon by the hidden part.

$$\frac{c_0 \vdash c \text{ depends-on } \Gamma \Rightarrow c' \quad c_d = c' \cup c'' \quad c_d \vdash c \text{ funs-exhaustive} \quad c_d \vdash c \text{ funs-unambiguous}}{\Gamma, c_0, c'' \vdash c \text{ implementation-side-typecheck}} \quad (127)$$

3.19.3 Top-level Functor ITC

These rules perform implementation-side typechecking of all blocks in the ModRep resulting from typechecking a functor instantiation. For simplicity, the entire Context resulting from the instantiation (represented by c'') is always included in c_d, the depended-upon blocks, of a block being checked.

$$\boxed{\Gamma, c_0, c'', \pi \vdash \nu \text{ functor-implementation-side-typecheck}}$$

$$\frac{\Gamma, c_0, c'' \vdash c \text{ bl-func-implementation-side-typecheck}}{\Gamma, c_0, c'', \texttt{bl} \vdash (\mu, c) \text{ functor-implementation-side-typecheck}} \quad (128)$$

$$\frac{\Gamma, c_0, c'' \vdash \mu \text{ str-func-implementation-side-typecheck}}{\Gamma, c_0, c'', \texttt{str} \vdash (\mu, c) \text{ functor-implementation-side-typecheck}} \quad (129)$$

$$\boxed{\Gamma, c_0, c'' \vdash c\text{bl-func-implementation-side-typecheck}}$$

$$\frac{c_0 \vdash c \text{ depends-on } \Gamma \Rightarrow c' \quad c_d = c' \cup c'' \quad c_d \vdash c \text{ functor-funs-exhaustive} \quad c_d \vdash c \text{ functor-funs-unambiguous}}{\Gamma, c_0, c'' \vdash c \text{ bl-func-implementation-side-typecheck}} \quad (130)$$

$$\boxed{\Gamma, c_0, c'' \vdash \mu \text{ str-func-implementation-side-typecheck}}$$

$$\frac{\Gamma, c_0, c'' \vdash \mu \text{ str-func-implementation-side-typecheck}}{\Gamma, c_0, c'' \vdash ((L, (\texttt{v}, pt))) + \mu \text{ str-func-implementation-side-typecheck}} \quad (131)$$

$$\frac{\begin{array}{c}\Gamma, c_0, c'', \pi \vdash \nu \text{ functor-implementation-side-typecheck} \\ \Gamma, c_0, c'' \vdash \mu \text{ str-func-implementation-side-typecheck}\end{array}}{\Gamma, c_0, c'' \vdash ((L, (\pi, \nu))) + \mu \text{ str-func-implementation-side-typecheck}} \quad (132)$$

Rule 132: We recursively perform implementation-side typechecks on all blocks within the given ModTypeEnv.

3.19.4 Dependency

These rules define our notion of modular typechecking: each block need only be implementation-side typechecked in the context of the blocks on which it *depends*. We say that block b_1 depends on block b_2 if any of the following holds:

- $b_1 = b_2$.
- b_2 declares a function that has a case in b_1.
- b_2 declares a (transitive) superclass of a class declared in b_1.

- b_2 declares a class that is mentioned in the argument type of a function declared in b_1.
- b_2 declares a class that is mentioned in the pattern of a case declared in b_1.

$$\boxed{c_0 \vdash c \text{ depends-on } \Gamma \Rightarrow c'}$$

$$\frac{c_0 \vdash c \text{ depends-on } \eta_1 \Rightarrow c_1 \quad \cdots \quad c_0 \vdash c \text{ depends-on } \eta_n \Rightarrow c_n}{c_0 \vdash c \text{ depends-on } \{(I_1, \eta_1), \ldots, (I_n, \eta_n)\} \Rightarrow c_1 \cup \ldots \cup c_n} \quad (133)$$

$$\boxed{c_0 \vdash c \text{ depends-on } \eta \Rightarrow c'}$$

$$\frac{}{c_0 \vdash c \text{ depends-on } (\mathtt{v}, pt) \Rightarrow ()} \quad (134)$$

$$\frac{\nu = (\mu, c'') \quad c_0 \vdash c \text{ depends-on } \mu \Rightarrow c'}{c_0 \vdash c \text{ depends-on } (\mathtt{str}, \nu) \Rightarrow c'} \quad (135)$$

$$\frac{\nu = (\mu, c'') \quad c_0 \vdash c \text{ depends-on } c'' \Rightarrow c'}{c_0 \vdash c \text{ depends-on } (\mathtt{bl}, \nu) \Rightarrow c'} \quad (136)$$

$$\boxed{c_0 \vdash c \text{ depends-on } \mu \Rightarrow c'}$$

$$\frac{c_0 \vdash c \text{ depends-on } \eta_1 \Rightarrow c_1 \quad \cdots \quad c_0 \vdash c \text{ depends-on } \eta_n \Rightarrow c_n}{c_0 \vdash c \text{ depends-on } ((L_1, \eta_1), \ldots, (L_n, \eta_n)) \Rightarrow c_1 \cup \ldots \cup c_n} \quad (137)$$

$$\boxed{c_0 \vdash c \text{ depends-on } c'' \Rightarrow c'}$$

$$\frac{fun \in \text{Locals}(c'') \quad (fun, Pat, \Gamma_0, c_0) \in \text{SCases}(c)}{c_0 \vdash c \text{ depends-on } c'' \Rightarrow c''} \quad (138)$$

Rule 138: c depends on c'' if c'' declares a function for which c defines a new case.

$$\frac{\kappa \in \text{Locals}(c'') \quad \kappa' \in \text{Locals}(c) \quad c_0 \vdash \kappa' \leq \kappa \Rightarrow \mathtt{y}}{c_0 \vdash c \text{ depends-on } c'' \Rightarrow c''} \quad (139)$$

Rule 139: c depends on c'' if c'' declares a class or interface that is a (definite) superclass/superinterface of a class or interface declared in c.

$$\frac{\kappa \in \text{Locals}(c'') \quad \text{FType}(c)(fun) = (\, Tv, T_1 \xrightarrow{Mark} T_2) \quad T_1 \text{ mentions } \kappa}{c_0 \vdash c \text{ depends-on } c'' \Rightarrow c''} \quad (140)$$

Rule 140: c depends on c'' if c'' declares a class or interface that is mentioned in the argument type of a function declared in c.

$$\frac{\kappa \in \text{Locals}(c'') \quad (fun, Pat, \Gamma_1, c_1) \in \text{SCases}(c) \quad Pat \text{ mentions } \kappa}{c_0 \vdash c \text{ depends-on } c'' \Rightarrow c''} \quad (141)$$

Rule 141: c depends on c'' if c'' declares a class that is mentioned in the pattern of a function case declared in c.

$$\frac{\text{no other rule applies}}{c_0 \vdash c \text{ depends-on } c'' \Rightarrow ()} \quad (142)$$

Rule 142: If the above rules do not apply, then c does not depend on c'', so the empty Context is returned.

3.19.5 Top-level Exhaustiveness Checking

$\boxed{c_d \vdash c \text{ funs-exhaustive}}$

These rules perform exhaustiveness checking on a block. If there are "unknown" classes, interfaces, or functions, then the block is in a functor body. In this case, only functions declared in the block itself are checked; external functions are checked per functor instantiation via the functor-funs-exhaustive rule below. If there are no "unknowns," then all functions in depended-upon blocks are checked.

$$\frac{\{fun_1, \ldots, fun_n\} = dom(\text{FType}(c)) \quad \text{Unknown}(c_d) \neq \{\} \qquad c_d, c \vdash fun_1 \text{ exhaustive} \quad \cdots \quad c_d, c \vdash fun_n \text{ exhaustive}}{c_d \vdash c \text{ funs-exhaustive}} \quad (143)$$

$$\frac{\{fun_1, \ldots, fun_n\} = dom(\text{FType}(c_d)) \quad \text{Unknown}(c_d) = \{\} \qquad c_d, c \vdash fun_1 \text{ exhaustive} \quad \cdots \quad c_d, c \vdash fun_n \text{ exhaustive}}{c_d \vdash c \text{ funs-exhaustive}} \quad (144)$$

$\boxed{c_d, c \vdash \text{functor-funs-exhaustive}}$

$$\frac{\{fun_1, \ldots, fun_n\} = \{fun \mid fun \in dom(\text{FType}(c_d)) \wedge fun \notin \text{Locals}(c)\} \qquad c_d, c \vdash fun_1 \text{ exhaustive} \quad \cdots \quad c_d, c \vdash fun_n \text{ exhaustive}}{c_d, c \vdash \text{functor-funs-exhaustive}} \quad (145)$$

Rule 145: When checking a functor instantiation, we only check functions not declared in the functor body. Functions declared in the functor body are checked once, when the functor is declared.

$\boxed{c_d, c \vdash fun \text{ exhaustive}}$

$$\frac{\begin{array}{c}(\text{FType}(c_d))(fun) = (Tv, T_1 \overset{Mark}{\rightarrow} T_2) \\ \{|\, Pat'_1, \ldots, Pat'_m \,|\} = \{|\, Pat' \mid (fun, Pat', \Gamma', c') \in \text{SCases}(c_d) \,|\} \\ d = \max(depth(T_1),\ depth(Pat'_1),\ \ldots,\ depth(Pat'_m)) \\ \{Pat_1, \ldots, Pat_n\} = patgen(c, c_d, fun, Mark, T_1, d) \\ c_d \vdash (fun, Pat_1) \text{ covered} \quad \cdots \quad c_d \vdash (fun, Pat_n) \text{ covered}\end{array}}{c_d, c \vdash fun \text{ exhaustive}} \quad (146)$$

Rule 146: We generate a set of candidate patterns via the *patgen* function. The generated patterns cover all possible legal arguments to the function, and each pattern has a sufficient *depth* to ensure that our check is not overly conservative. Each generated pattern must have at least one applicable function case in the function being checked.

$\boxed{c_d \vdash (fun, Pat) \text{ covered}}$

$$\frac{(fun, Pat', \Gamma', c') \in \text{SCases}(c_d) \quad c_d \vdash Pat \leq Pat'}{c_d \vdash (fun, Pat) \text{ covered}} \quad (147)$$

3.19.6 Pattern Specificity

$$\boxed{c \vdash Pat \leq Pat'}$$

These rules are the static analogue of the dynamic pattern specificity rules (section 2.11.3). In these rules, we generalize "class" patterns to allow the class component to in fact be an interface. This is necessary because we sometimes treat interfaces as concrete classes for the purposes of exhaustiveness checking.

$$\frac{}{c \vdash Pat \leq _} \quad (148)$$

$$\frac{c \vdash Pat_1 \leq Pat_2}{c \vdash (I \texttt{ as } Pat_1) \leq Pat_2} \quad (149)$$

$$\frac{c \vdash Pat_1 \leq Pat_2}{c \vdash Pat_1 \leq (I \texttt{ as } Pat_2)} \quad (150)$$

$$\frac{c \vdash \kappa_1 \leq \kappa_2 \Rightarrow \texttt{y} \quad \kappa_1 \neq \kappa_2 \quad (\kappa_1 \notin \text{Unknown}(c) \text{ or } \kappa_2 \in \text{Unknown}(c))}{c \vdash \kappa_1\ Pat_1 \leq \kappa_2\ Pat_2} \quad (151)$$

Rule 151: The last two premises conservatively ensure that κ_1 and κ_2 cannot denote the same class or interface.

$$\frac{c \vdash Pat_1 \leq Pat_2}{c \vdash \kappa\ Pat_1 \leq \kappa\ Pat_2} \quad (152)$$

$$\frac{c \vdash Pat_1 \leq Pat'_1 \quad \cdots \quad c \vdash Pat_n \leq Pat'_n}{c \vdash \{L_1 = Pat_1, \ldots, L_n = Pat_n\} \leq \{L_1 = Pat'_1, \ldots, L_n = Pat'_n\}} \quad (153)$$

3.19.7 Pattern Generation

First we define the "depth" of a pattern and of a type.

$depth(_) = 0$
$depth(I \texttt{ as } Pat) = depth(Pat)$
$depth(\ Tv\ Ct\ Pat) = 1 + depth(Pat)$
$depth(\{L_1 = Pat_1, \ldots, L_n = Pat_n\}) = 1 + \max(depth(Pat_1), \ldots, depth(Pat_n))$

$depth(I) = 0$
$depth((\ Ts, \kappa)) = 1$
$depth(\{L_1 : T_1, \ldots, L_n : T_n\}) = 1 + \max(depth(T_1), \ldots, depth(T_n))$
$depth(T_1 \overset{Mark}{\rightarrow} T_2) = 0$

Below we describe the algorithm for generating patterns to check. The *patgen* function takes several arguments:

- The Context of the block being checked.
- The Context representing all blocks depended on by the block being checked.
- The fun that is being checked for exhaustiveness.
- Either a *Mark*, which identifies the DP of the function being checked, or the special symbol $\Box$, which signifies that the DP is not on this branch of the recursion. The meta-variable M ranges over these possibilities.
- A type T, representing the type of the patterns to generate.
- An integer representing the depth of the patterns to generate.

The *patgen* function uses a helper function called *explode*, which essentially performs a Cartesian product. The meta-variable s (possibly subscripted) represents a set of patterns.

$$
\begin{array}{l}
patgen(c, c_d, fun, M, T, 0) = \{\ _\ \} \\
patgen(c, c_d, fun, M, T_1 \xrightarrow{Mark} T_2, d) = \{\ _\ \} \\
\quad \text{where } d > 0 \\
patgen(c, c_d, fun, M, I, d) = \{\ _\ \} \\
\quad \text{where } d > 0 \\
patgen(c, c_d, fun, M, ((T_1, \ldots, T_m), cls), d) = \{cls\ Pat_1, \ldots, cls\ Pat_n\} \\
\quad \text{where } d > 0,\ cls \notin \text{Abstract}(c_d),\ (\text{Rep}(c_d))(cls) = ((I_1, \ldots, I_m), T), \\
\qquad \{Pat_1, \ldots, Pat_n\} = patgen\ (c, c_d, fun, \Box, T[T_1/I_1, \ldots, T_m/I_m], d-1) \\
patgen(c, c_d, fun, M, ((T_1, \ldots, T_m), \kappa), d) = \{\kappa\ Pat_1, \ldots, \kappa\ Pat_n\} \\
\quad \text{where } d > 0,\ \kappa \in \text{Abstract}(c_d) \text{ or } \kappa = ifc,\ (M \neq \emptyset) \vee (fun \in \text{Locals}(c) \wedge \kappa \notin \text{Locals}(c)), \\
\qquad (\text{Rep}(c_d))(\kappa) = ((I_1, \ldots, I_m), T), \\
\qquad \{Pat_1, \ldots, Pat_n\} = patgen(c, c_d, fun, \Box, T[T_1/I_1, \ldots, T_m/I_m], d-1) \\
patgen(c, c_d, fun, M, (Ts, \kappa), d) = patgen(c, c_d, fun, M, (Ts_1, \kappa_1), d) \cup \cdots \cup patgen(c, c_d, fun, M, (Ts_n, \kappa_n), d) \\
\quad \text{where no other case applies,} \\
\qquad \{(Ts_1, \kappa_1), \ldots, (Ts_n, \kappa_n)\} = \{(Ts_0, \kappa_0) \mid ((Tv, \kappa_0), (Ts', \kappa)) \in \text{SInh}(c_d) \wedge c_d \vdash (Ts_0, \kappa_0) \leq_{sub} (Ts, \kappa) \\
\qquad\qquad \wedge \exists I_1, \ldots, I_m.\ (I_1, \ldots, I_m \text{ fresh} \\
\qquad\qquad\qquad \wedge \{I_1, \ldots, I_m\} \vdash (Ts_0, \kappa_0) \text{ is-safe-for } (Ts, \kappa))\} \\
patgen(c, c_d, fun, (L_j, Mark), \{L_1 : T_1, \ldots, L_n : T_n\}, d) = \\
\quad explode([patgen(c, c_d, fun, \Box, T_1, d-1), \ldots, patgen(c, c_d, fun, \Box, T_{j-1}, d-1), \\
\qquad patgen(c, c_d, fun, Mark, T_j, d-1), \\
\qquad patgen(c, c_d, fun, \Box, T_{j+1}, d-1), \ldots, patgen(c, c_d, fun, \Box, T_n, d-1)], \\
\qquad [L_1, \ldots, L_n]) \\
\quad \text{where } d > 0,\ 1 \leq j \leq n \\
patgen(c, c_d, fun, \Box, \{L_1 : T_1, \ldots, L_n : T_n\}, d) = \\
\quad explode([patgen(c, c_d, fun, \Box, T_1, d-1), \ldots, patgen(c, c_d, fun, \Box, T_n, d-1), \\
\qquad [L_1, \ldots, L_n]) \\
\quad \text{where } d > 0 \\
\\
explode([], []) = \{\} \\
explode([s_1, \ldots, s_n], [L_1, \ldots, L_n]) = \{\}
\end{array}
$$

where $\exists 1 \leq j \leq n.s_j = \emptyset$
$explode([\{Pat_1\}, \ldots, \{Pat_n\}], [L_1, \ldots, L_n]) = \{\{L_1 = Pat_1, \ldots, L_n = Pat_n\}\}$
$explode([s_1, \ldots, s_n], [L_1, \ldots, L_n]) =$
$explode([s_1, \ldots, s_{j-1}, \{Pat_1\}, s_{j+1}, \ldots, s_n], [L_1, \ldots, L_n])$
$\cup\ explode([s_1, \ldots, s_{j-1}, \{Pat_2, \ldots, Pat_n\}, s_{j+1}, \ldots, s_n], [L_1, \ldots, L_n])$
where $\forall 1 \leq k \leq n.s_k \neq \emptyset$, $1 \leq j \leq n$, $s_j = \{Pat_1, \ldots, Pat_n\}$, $n \geq 2$

The only pattern at depth 0 is the wildcard. The only pattern for an argument position whose type is a function type or type variable is also the wildcard. For a concrete class at depth d, we recursively compute all patterns for the class's representation at depth $d-1$ and create the corresponding class patterns. The next case treats an abstract class or interface as if it were concrete if either the current position is not the DP, or the function being checked is declared in the current block while the argument type was not. If these conditions are not met, we may leave the abstract class or interface unimplemented; we therefore recursively generate patterns for all of the class or interface's available direct subclasses. A complication in this case is the need to invent type parameters (which are simply fresh type variables) for any extra type variables introduced by these subclasses. The last two cases generate patterns for records by recursively generating the patterns for components and performing a Cartesian product (via the *explode* function). The cases differ on whether or not a mark is propagated to the appropriate component.

3.19.8 Top-level Ambiguity Checking

$\boxed{c_d \vdash c \text{ funs-unambiguous}}$

These rules perform ambiguity checking on a block. If there are "unknown" classes, interfaces, or functions, then the block is in a functor body. In this case, only functions declared in the block itself are checked; external functions are checked per functor instantiation via the functor-funs-unambiguous rule below. If there are no "unknowns," then all functions in depended-upon blocks are checked.

$$\frac{\{fun_1, \ldots, fun_n\} = dom(\text{FType}(c)) \quad \text{Unknown}(c_d) \neq \{\} \quad c_d, c \vdash fun_1 \text{ unambiguous} \quad \cdots \quad c_d, c \vdash fun_n \text{ unambiguous}}{c_d \vdash c \text{ funs-unambiguous}} \quad (154)$$

$$\frac{\{fun_1, \ldots, fun_n\} = dom(\text{FType}(c_d)) \quad \text{Unknown}(c_d) = \{\} \quad c_d, c \vdash fun_1 \text{ unambiguous} \quad \cdots \quad c_d, c \vdash fun_n \text{ unambiguous}}{c_d \vdash c \text{ funs-unambiguous}} \quad (155)$$

$\boxed{c_d \vdash c \text{ functor-funs-unambiguous}}$

$$\frac{\{fun_1, \ldots, fun_n\} = \{fun \mid fun \in dom(\text{FType}(c_d)) \wedge fun \notin \text{Locals}(c)\} \quad c_d, c \vdash fun_1 \text{ unambiguous} \quad \cdots \quad c_d, c \vdash fun_n \text{ unambiguous}}{c_d \vdash c \text{ functor-funs-unambiguous}} \quad (156)$$

Rule 156: When checking a functor instantiation, we only check functions not declared in the functor body. Functions declared in the functor body are checked once, when the functor is declared.

$\boxed{c_d, c \vdash fun \text{ unambiguous}}$

$$\frac{\begin{array}{c}\{|\ Pat_1, \ldots, Pat_n\ |\} = \{|\ Pat \mid (fun, Pat, \Gamma', c') \in \text{SCases}(c_d)\ |\} \\ c_d \vdash (Pat_1, Pat_2) \text{ unambiguous-in } fun \\ \cdots \\ c_d \vdash (Pat_1, Pat_n) \text{ unambiguous-in } fun \\ \cdots \\ c_d \vdash (Pat_{n-1}, Pat_n) \text{ unambiguous-in } fun \\ c \vdash (fun, Pat_1) \text{ obeys } Mark \quad \cdots \quad c \vdash (fun, Pat_n) \text{ obeys } Mark\end{array}}{c_d, c \vdash fun \text{ unambiguous}} \ (157)$$

Rule 157: Each pair of function-case patterns must be unambiguous. In addition, each function-case pattern must satisfy the "marked argument restriction" (see below).

$\boxed{c_d \vdash (Pat_1, Pat_2) \text{ unambiguous-in } fun}$

$$\frac{c_d \vdash Pat_1 \text{ disjoint-from } Pat_2}{c_d \vdash (Pat_1, Pat_2) \text{ unambiguous-in } fun} \ (158)$$

$$\frac{\begin{array}{c}c_d \vdash Pat_1 \cap Pat_2 \Rightarrow Pat \quad (fun, Pat_3, \Gamma_3, c_3) \in \text{SCases}(c_d) \\ c_d \vdash Pat \leq Pat_3 \quad c_d \vdash Pat_3 \leq Pat_1 \quad c_d \vdash Pat_3 \leq Pat_2 \\ c_d \not\vdash Pat_1 \leq Pat_3 \text{ or } c_d \not\vdash Pat_2 \leq Pat_3\end{array}}{c_d \vdash (Pat_1, Pat_2) \text{ unambiguous-in } fun} \ (159)$$

Rule 159: Pat_1 and Pat_2 are not ambiguous if there exists a function case with a pattern Pat_3 that covers the intersection of Pat_1 and Pat_2, is as specific as both of those patterns, and is strictly more specific than one of the two (this last condition is ensured by the last premise in the rule).

3.19.9 Disjointness Checking

$\boxed{c_d \vdash Pat \text{ disjoint-from } Pat'}$

$$\frac{c_d \vdash Pat_1 \text{ disjoint-from } Pat_2}{c_d \vdash (I \texttt{ as } Pat_1) \text{ disjoint-from } Pat_2} \ (160)$$

$$\frac{c_d \not\vdash cls_1 \leq cls_2 \Rightarrow ? \quad c_d \not\vdash cls_2 \leq cls_1 \Rightarrow ?}{c_d \vdash (cls_1\ Pat_1) \text{ disjoint-from } (cls_2\ Pat_2)} \ (161)$$

Rule 161: The patterns are disjoint if there is no possibility for one class to subclass the other.

$$\frac{c_d \vdash Pat_1 \text{ disjoint-from } Pat_2}{c_d \vdash (cls_1\ Pat_1) \text{ disjoint-from } (cls_2\ Pat_2)} \ (162)$$

$$\frac{c_d \vdash Pat_{1,i} \text{ disjoint-from } Pat_{2,i} \quad 1 \leq i \leq n \quad n \leq k}{c_d \vdash \{L_1 = Pat_{1,1}, \ldots, L_n = Pat_{1,n}\} \text{ disjoint-from } \{L_1 = Pat_{2,1}, \ldots, L_k = Pat_{2,k}\}} \ (163)$$

$$\frac{c_d \vdash Pat_2 \text{ disjoint-from } Pat_1}{c_d \vdash Pat_1 \text{ disjoint-from } Pat_2} \ (164)$$

3.19.10 Pattern Intersection

$$\boxed{c_d \vdash Pat \cap Pat' \Rightarrow Pat''}$$

$$\frac{}{c_d \vdash Pat \cap _ \Rightarrow Pat} \quad (165)$$

$$\frac{c_d \vdash Pat_1 \cap Pat_2 \Rightarrow Pat}{c_d \vdash I \texttt{ as } Pat_1 \cap Pat_2 \Rightarrow Pat} \quad (166)$$

$$\frac{c_d \vdash cls_1 \leq cls_2 \Rightarrow \texttt{y} \quad cls_1 \neq cls_2 \quad (cls_1 \notin \text{Unknown}(c) \text{ or } cls_2 \in \text{Unknown}(c))}{c_d \vdash (cls_1\ Pat_1) \cap (cls_2\ Pat_2) \Rightarrow (cls_1\ Pat_1)} \quad (167)$$

Rule 167: The last two premises ensure that cls_1 does not denote the same class as cls_2.

$$\frac{c_d \vdash Pat_1 \cap Pat_2 \Rightarrow Pat}{c_d \vdash (cls\ Pat_1) \cap (cls\ Pat_2) \Rightarrow (cls\ Pat)} \quad (168)$$

$$\frac{c_d \vdash Pat_1 \cap Pat_1' \Rightarrow Pat_1'' \quad \cdots \quad c_d \vdash Pat_n \cap Pat_n' \Rightarrow Pat_n''}{\begin{array}{c} c_d \vdash \{L_1 = Pat_1, \ldots, L_n = Pat_n\} \cap \{L_1 = Pat_1', \ldots, L_n = Pat_n'\} \\ \Rightarrow \{L_1 = Pat_1'', \ldots, L_n = Pat_n''\} \end{array}} \quad (169)$$

$$\frac{c_d \vdash Pat_2 \cap Pat_1 \Rightarrow Pat}{c_d \vdash Pat_1 \cap Pat_2 \Rightarrow Pat} \quad (170)$$

3.19.11 The Marked Argument Restriction

$$\boxed{c \vdash (fun, Pat) \text{ obeys } Mark}$$

These rules ensure that each function case is either declared in the same block as its function or declared in the same block as the class in the distinguished position of the case's pattern.

$$\frac{fun \in \text{Locals}(c)}{c \vdash (fun, Pat) \text{ obeys } Mark} \quad (171)$$

$$\frac{c \vdash (fun, Pat) \text{ obeys } Mark}{c \vdash (fun, (I \texttt{ as } Pat)) \text{ obeys } Mark} \quad (172)$$

$$\frac{cls \in \text{Locals}(c)}{c \vdash (fun, (cls\ Pat)) \text{ obeys } \emptyset} \quad (173)$$

$$\frac{c \vdash (fun, Pat_i) \text{ obeys } Mark \quad 1 \leq i \leq n}{c \vdash (fun, \{L_1 = Pat_1, \ldots, L_n = Pat_n\}) \text{ obeys } (L_i, Mark)} \quad (174)$$

3.19.12 Three-valued Subclassing

$$\boxed{c \vdash \kappa \leq \kappa' \Rightarrow \beta}$$

$$\frac{((Tv, \kappa_1), (Ts, \kappa_2)) \in \text{SInh}(c)}{c \vdash \kappa_1 \leq \kappa_2 \Rightarrow \beta} \quad (175)$$

Rule 175: Type variables and type parameters are irrelevant for the purposes of the subclassing relation.

$$\frac{}{c \vdash \kappa \leq \kappa \Rightarrow \beta} \quad (176)$$

$$\frac{c \vdash \kappa_1 \leq \kappa_3 \Rightarrow \beta_1 \quad c \vdash \kappa_3 \leq \kappa_2 \Rightarrow \beta_2 \quad \beta_3 = \beta_1 \sqcap \beta_2}{c \vdash \kappa_1 \leq \kappa_2 \Rightarrow \beta_3} \quad (177)$$

$$\frac{c \vdash \kappa_1 \leq \kappa_3 \Rightarrow ? \quad c \vdash \kappa_2 \leq \kappa_3 \Rightarrow ? \quad \kappa_1 \in \text{Unknown}(c) \text{ or } \kappa_2 \in \text{Unknown}(c)}{c \vdash \kappa_1 \leq \kappa_2 \Rightarrow ?} \quad (178)$$

Rule 178: This rule is fairly conservative, as it implies that any two types (classes or interfaces) that may have a common supertype are possibly the same type. The multiple inheritance allowed by interfaces forces us to be this conservative. We could do a case analysis on κ_1 and κ_2 to be more precise for classes, which support only single inheritance.

4 Formal Properties

As usual, we divide type soundness for EML into two properties, subject reduction and progress. Subject reduction says that types are preserved through evaluation:

Theorem (Subject Reduction): If $\vdash P : T \Rightarrow c$ and $\vdash P \Rightarrow v$, then $\{\}, c, () \vdash v : T$.
Proof Sketch: We have currently proven this theorem for the subset of EML without signature ascription and functors. We define simple correspondances between Env and ModTypeEnv, TopEnv and TopTypeEnv, OOInfo and Context. Then we show by induction on the length of the typing derivation that these correspondances are preserved through parallel typechecking and evaluation steps, which then implies type preservation. The Context c in the judgement $\{\}, c, () \vdash v : T$ is necessary to provide the types of extensible functions and to allow type subsumption via the subtype relation.

Progress says that well-typed programs do not get "stuck." We are currently in the process of adapting the progress proof from our earlier theoretical work to EML:

Conjecture (Progress): If $\vdash P : T \Rightarrow c$ and $\vdash P$ does not diverge in the dynamic semantics, then $\vdash P \Rightarrow v$.