An Imperative, First-Order Calculus with Object Extension

Viviana Bono
Dipartimento di Informatica
dell'Università di Torino
C.so Svizzera 185,
10149 Torino, Italy
bono@di.unito.it

Kathleen Fisher
AT&T Labs—Research,
180 Park Ave.,
Florham Park, NJ 07932
+1 973 360-8675
kfisher@research.att.com

December 1, 1997

Abstract

This paper presents an imperative object calculus, designed to support class-based programming via a combination of extensible objects and encapsulation. This calculus simplifies the language presented in [Fis96] in that, like C++ and Java, it chooses to support an imperative semantics instead of method specialization. We show how Java-style classes and "mixins" may be coded in this calculus, prove a soundness theorem (via a subject reduction property), and give a sound and complete typing algorithm.

1 Introduction

In the past few years, many theoretical studies have addressed the problem of modeling object-oriented languages via primitive operations [AC96b, FHM94, Fis96, BBL96, PT94, RS98]. Most of these approaches have centered on object-based calculi, where objects, not classes, are seen as the primitive building blocks. However, statically-typed, class-based languages like Simula [BDMN73], Eiffel [Mey92], C++ [ES90, Str94], and Java [AG96] are the most common form of object-oriented language. Hence it is important to evaluate object-based calculi with respect to how well they support class-based programming. The main contribution of this paper is the formal study of a calculus of encapsulated extensible objects, which we use to model class hierarchies. To motivate our study, we begin with a comparison between our approach to modeling classes and the well-known *record-of-premethods* approach. This comparison, which summarizes [FM98], reveals why extensible calculi are relevant to class-based programming. Interested readers are referred to that paper for more details.

1.1 Premethods vs. Extensible Objects

In the context of object calculi, it seems natural to define inheritance using *premethods*, functions that are written with the intent of becoming object methods, but which are not yet installed in any object. Premethods are functions that explicitly depend on the "object itself," typically assumed to be the first parameter to the function. Following this idea, Abadi and Cardelli have proposed encoding classes in a pure object system using records of premethods [AC96c]; these ideas are also used in [RR96]. In this approach, a class is an object that contains a record of premethods and a constructor function used to package these premethods into objects.

The primary advantage of the record-of-premethods encoding is that it does not require a complicated form of object. All that is needed is a way of forming an object from a list of component definitions. However, this approach has some serious drawbacks. We discuss these drawbacks using a list of criteria developed in [FM98] to characterize the role of classes in class-based languages.

Does the class construct provide a coherent, extensible collection? The combination of a record of premethods and a constructor function may be thought of as a coherent, extensible collection. Because premethods are simply fields in a record, nothing requires that they be coherent until a constructor function is supplied. Since the constructor function installs the premethods into an object, however, the fact that a given constructor is typable implies that the premethods it uses are coherent. Notice, however, that nothing requires a given constructor to mention all of the premethods in a given premethod record.

Does the class construct guarantee initialization? In more elaborate record-of-premethod models, the code to initialize private instance variables is guaranteed to run if any of the associated premethods is installed into an object. However, constructor functions cannot be usefully reused in derived classes. A consequence is that if a class designer puts initialization code into a class constructor, that code will not be executed for derived classes. There are several program-development scenarios where this lack would be a serious problem. For example, class designers may wish to perform some kind of bookkeeping whenever objects are instantiated from a class or its descendants. Such bookkeeping is common in database applications, for example. To achieve it, programmers need a place to put code that will execute whenever an object is instantiated. With the record-of-premethods approach, however, there is no appropriate place: no base class constructor function will be called for derived classes, and a premethod function may be called without creating an object.

Does the class construct provide an explicit type hierarchy? In many existing class-based languages, it is possible to restrict the subtypes of an object type to classes that inherit all or part of the class implementation. This restriction may be useful for optimizing operations on objects, allowing access to argument objects in binary methods, and guaranteeing semantic consistency beyond type considerations (some discussion of these points appears in [KLM94]). A special case of this capability is the ability to define *final* classes, as recognized in work on Rapide [KLM94] and incorporated (presumably independently) as a language feature in Java.

This ability is lacking in the record-of-premethods approach since any object whose type is a structural subtype of another type τ can be used as an object of type τ.

Does the class mechanism automatically propagate base class changes? Because derived class constructors must explicitly name the methods that they wish to inherit, the record-of-premethods approach does not automatically propagate base class method changes. In particular, if a derived class `D` is defined from a base class `B` in Java or related languages, then adding a method to `B` will result in an additional method of `D`, and similarly for every other class derived from `B` (and there may be many). With the record-of-premethods approach, derived class constructors must be explicitly rewritten each time base classes change. Since object-oriented programs are typically quite large and maintenance may be distributed across many people, the person who maintains a base class may fail to inform those maintaining its derived classes of its change, causing unpredictable errors. There is no mechanism in this approach to detect such errors automatically.

While we readily admit that simplicity is a virtue, the above discussion reveals that several important and desirable features of class-based programming are lost in the record-of-premethods model. Extensible objects provide a rich alternative. As discussed in [FM98, Fis96], a principled way to think about class-based object-oriented languages is as the combination of two orthogonal components: (i), an object system that supports inheritance and message sending and (ii), an encapsulation mechanism that provides hiding. We call this model of classes the "Classes = Extensible Objects + Encapsulation" approach. Referring to the class-evaluation checklist we used to evaluate the pre-methods model, we can see that this approach successfully addresses each of the points listed there: it provides an extensible coherent collection, guarantees initialization, supports an explicit type hierarchy, and automatically propagates base class changes.

Extensible, Coherent Collection. Extensible objects obviate the need for premethods, since collections of methods that are already installed in objects may be extended. Because of this fact, we may impose static constraints on the ways in which one method may be combined with others. For example, if an object contains two mutually recursive methods, then we cannot replace one with another of a different type. In contrast, in the record-of-premethods approach, it is possible to form a record of premethods without a "covering" constructor that checks to be sure that all of the premethods are coherent.

Guaranteed Initialization. A second advantage of extensible objects is that class constructors and initialization code can be inherited, *i.e.*, reused in derived classes. For example, to create `ColorPoint` objects, we may invoke the `Point` class constructor and add color methods to the resulting extensible object. This process guarantees that the `Point` class has the opportunity to initialize properly any inherited components. It also guarantees that the designers of the `Point` class have the opportunity to update any bookkeeping information they may be keeping about instantiations of `Point` objects.

Explicit Type Hierarchy. A further advantage is the rich subtyping structure of this approach. In particular, it provides "implementation" types that subtype along the inheritance hierarchy, "interface" types that subtype via structural subtyping rules, and a hybrid subtyping relation that allows implementation types to be subtypes of interface types. With this subtyping structure, programmers can use implementation types where the extra information is useful and interface types where more generality is required.

Automatic Propagation of Changes. Another advantage arises with private (or protected) methods. In the extensible-object formulation, methods always remain within an object, even when it is extended. These hidden methods exist in all future extensions, but they can only be accessed by methods that were defined before the method became hidden. Furthermore, these private methods need not be explicitly manipulated by derived class constructors to insure that they are treated properly.

These advantages may be seen in the first example in Section 4, which shows how a `Point`-`ColorPoint` hierarchy can be formally modeled in our calculus.

1.2 Our Extensible Calculus

Fisher's thesis [Fis96] formally presents a functional object calculus that includes extensible objects and encapsulation. This calculus realizes the benefits described in the previous section. In addition, it includes "mytype" type variables to support method specialization. Unfortunately, these variables greatly complicate analysis. Consequently, to understand the "Classes = Extensible Objects + Encapsulation" idea in its simplest form, Fisher and Mitchell use a simpler language without a "mytype" mechanism in [FM98]. However, they do not study the formal properties of that simpler language. The formal study of such a language is the subject of this paper.

Our calculus, like the one in Fisher's thesis, includes extensible objects and encapsulation; however, it differs in several key aspects. In particular, to compensate for the lack of a "mytype" mechanism, we have adopted an imperative semantics. Hence instead of cascading messages:

$$((\mathtt{ob} \Leftarrow \mathtt{m_1}(\mathtt{arg_1}) \ldots) \Leftarrow \mathtt{m_n}(\mathtt{arg_n})$$

a programmer can send a sequence of messages to the same object:

$$\mathtt{ob} \Leftarrow \mathtt{m_1}(\mathtt{arg_1}); \quad \cdots; \quad \mathtt{ob} \Leftarrow \mathtt{m_n}(\mathtt{arg_n});$$

Moreover, in [Fis96], the type system insures the soundness of the object extension operator (a method `m` can only be added to an object that does not already have a method `m`) by tracking method *absences.* Tracking absences requires recording the list of all methods which might be added to the object in the future. This practice has the unfortunate consequence of requiring parent classes to guess what methods their descendants will want to add. In contrast, in the calculus described in this paper, we adjust the type system to track potential method *presences* instead, relieving super objects (*i.e.*, parent classes) of the need for clairvoyance.

To facilitate the type soundness proof, we have added bounded universal types to our language. In the process, we have gained the ability to write "mixins," functions that take an object as an argument and return an extension of that object as a result. For example, a `color` mixin might add a `color` method to any object that does not already contain `color`.

Finally, our calculus differs from that in [Fis96] in that there is no known typing algorithm for the language with “mytype”.

The rest of this paper is structured as follows. In Sections 2 and 3 we formally present our calculus and its type system. In Section 4, we show how this calculus may be used to encode Java-style classes and to model mixins. We introduce our imperative operational semantics in Section 5, and in Section 6 we sketch the soundness proof for the calculus with respect to the operational semantics (via a subject reduction property, presented in the style of [AC96a]). In Section 7 we present a typing algorithm for our system. We conclude with a discussion of related work and some thoughts on future directions. A preliminary version of this paper (without type soundness and without the typing algorithm) will appear in the FOOL'98 workshop proceedings.

2 The Syntax

The calculus extends an untyped lambda calculus with (i) object primitives to build extensible and updatable objects, (ii) universal types to write polymorphic method bodies, and (iii) existential types to provide encapsulation. Together, these language features may be used to model class hierarchies. Consequently, classes do not appear as primitive expressions in our calculus.

Intuitively, an object is a collection of *methods* and *fields*, each of which has a name and a body. We syntactically distinguish between methods and fields because they have different semantic and typing properties. Briefly, a method is a function which take as an (implicit) argument the object itself, whereas a field can be seen as part of its host object's state. We discuss the distinctions between methods and fields in more detail in Sections 3.4 and 5.

An expression of the untyped calculus can be any of the following:

$$
\begin{array}{rcl}
e & ::= & x \;\mid\; c \;\mid\; \lambda x.\, e \;\mid\; e_1 e_2 \;\mid\; \mathbf{let}\ x \ =\ e_1\ \mathbf{in}\ e_2 \;\mid \\
 & & \langle\rangle \;\mid \\
 & & e \Leftarrow m \;\mid\; \langle e_1 \leftarrow m = e_2 \rangle \;\mid\; \langle e_1 \leftarrow\!\!+\; m = e_2 \rangle \;\mid \\
 & & e.f \;\mid\; \langle e_1.f := e_2 \rangle \;\mid\; \langle e_1 \leftarrow\!\!+\!: f = e_2 \rangle \;\mid \\
 & & \Lambda().\, e \;\mid\; e * () \;\mid \\
 & & \{\!| r <:_w R \ :: L = R_1,\ e |\!\} \;\mid \\
 & & \mathbf{Abstype}\ r <:_w R \ :: L\ \mathbf{with}\ x : \tau\ \mathbf{is}\ e_1\ \mathbf{in}\ e_2
\end{array}
$$

where x is a term variable, c is a constant, $\lambda x.\, e$ is function abstraction, $e_1 e_2$ is function application, and $\mathbf{let}\ x \ =\ e_1\ \mathbf{in}\ e_2$ is let abstraction. In the object-related forms, we use m to denote method names and f for field names. These object forms may be interpreted as follows:

- $\langle\rangle$ is the empty object.
- $e \Leftarrow m$ and $e.f$ are respectively method and field invocation on object e.
- $\langle e_1 \leftarrow m = e_2\rangle$ and $\langle e_1.f := e_2\rangle$ respectively override an existing method or field in e_1 with a new body e_2.
- $\langle e_1 \leftarrow\!\!+\; m = e_2\rangle$ and $\langle e_1 \leftarrow\!\!+\!\!: f = e_2\rangle$ extend e_1 with a new method or a new field, respectively. In either case, expression e_2 provides the body of the new component.

We defer a discussion about the remaining syntactic forms to Section 3, as they are closely related to types. We also delay the presentation of the operational semantics of our calculus so that we may first present our type system and describe in a reasonably complete fashion how these constructs may be used to model classes. It is hoped that this discussion may develop the reader's intuition for our calculus.

3 The Type System

Our type system is centered around object types. Informally, an object type is a list of *label:type* pairs, each representing a method or field name with its type. Our calculus supports two sorts of object, as in [FM95a]: *prototypes*, typed with **pro**-types, and "proper" objects, typed with **obj**-types. A prototype can be extended with new methods and fields and its components can be overridden, but it supports only trivial subtyping. A prototype can be "sealed" to reach the status of proper object, to which width and depth subtyping applies, but then it is no longer extensible or overridable. This restriction is to avoid the well-known unsoundness that arises when combining object operations with subsumption (see for example [FM95b]). Object types are used in conjunction with existentially quantified types to type encapsulated extensible objects, which we use to model class hierarchies.

The type syntax is as follows:

Types:

$$\tau \quad ::= \quad unit \mid \iota \mid \tau_1 \rightarrow \tau_2 \mid \mathbf{pro}.R \mid \mathbf{obj}.R \mid$$
$$\forall(r <:_w R :: L)\tau \mid \exists(r <:_w R :: L)\tau$$

Rows:

$$R \quad ::= \quad r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid l : \tau\rangle\!\rangle$$

Kinds:

$$\gamma \quad ::= \quad T \mid L$$

All ::= A countable set of method and field labels.

L ::= $\mathrm{L} \subseteq All$

Types include the *unit* type, constant types, function types, object types of the two sorts, and universally and existentially quantified types. Rows include row variables r, the empty row $\langle\!\langle\rangle\!\rangle$, and rows formed by extending other rows with a *label:type* pair, $\langle\!\langle R \,|\, l : \tau \rangle\!\rangle$. Well-formed types have kind T, while rows have kinds that are subsets of All, the set of all labels that can occur in a row. Intuitively, if row R has kind L, then at least all the labels listed in R appear in L. For notational convenience, we will write closed rows (*i.e.*, rows without row variables) using the notations $\langle\!\langle m_i : \tau_i,\, f_j : \tau'_j \rangle\!\rangle^{i \in I,\, j \in J}$ and $\langle\!\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\!\rangle$.

Rows, introduced in [Wan78], are the tool we use to incorporate polymorphism into our system. We need polymorphism to type extensible objects: because each method takes its host object as an argument, each method body must be polymorphic to work properly in extensions of its original host. Technical considerations require that method body types be closed to prove our subject reduction theorem. Consequently, we have introduced universal quantifiers on row variables to provide closed, polymorphic types. Fortuitously, this type construct makes our system more expressive, for it allows us to type *mixins*, functions that take objects as parameters and return extensions of those objects as results. Finally, existential quantifiers on row variables produce abstract "object" types. Intuitively, such types reveal an object's interface (*i.e.*, the list of methods and fields to which the object will respond) and abstract away implementation details in a fashion similar to abstract data types.

Because of their roles, we find it useful to have partial information about the row variables that are quantified in $\forall$ and $\exists$ types. To that end, each row variable is declared with a width-supertyping bound R and a kind L: $(r <:_w R :: L)$. Intuitively, the width-bound stands for the minimum amount of information that any instance of the row variable r may contain. The kind L records an upper bound on the labels present in any instance of r. The labels in L provide the positive information to track method *presences* that we discussed in Section 1: a new label can be added to a row only if it is not already present in L.

3.1 Type-Related Expressions

With this background, we may now discuss universal and existential quantifier-related expressions. The universal quantifier forms are $\Lambda().\, e$ for row abstraction and $e * ()$ for row application. These forms can be considered as placeholders for the universal quantifier introduction and elimination typing rules. An abstract datatype declaration

$$\textbf{Abstype}\ r <:_w R :: L\ \textbf{with}\ x : \tau\ \textbf{is}\ e_1\ \textbf{in}\ e_2$$

declares a data structure named x whose (public) interface is represented by row R and kind L. Intuitively, R describes the components guaranteed to be in x, while L describes the component names that *may* be used in x. Data structure x can only be extended with components whose names are not listed in L. Expression e_1 is the implementation of x, and e_2 represents the code in which this data structure may be used. An implementation $(\{\!|r <:_w R :: L = R',\ e|\!\})$ of an abstract type is characterized by R' and e. Row R' is the actual set of components used to implement the data structure. The components listed in both R and

R' may be regarded as public, while those appearing only in R' are private. Expression e is the (hidden) implementation code. We use these two constructs to encapsulate the implementations of objects.

3.2 Judgment Forms

Contexts in our calculus are defined as follows:

$$\rho \quad ::= \quad \epsilon \ \mid \ \rho, (r <:_w R :: L)$$

$$\Gamma \quad ::= \quad \emptyset \ \mid \ \Gamma, x : \tau$$

We refer to them respectively as *row* and *term* contexts. Functions $dom(\rho)$ and $dom(\Gamma)$ are defined in the standard way. $FV(\Gamma)$ denotes the set of row variables appearing free in any type τ such that $x : \tau \in \Gamma$, for some x. Judgments are as follows:

$\rho \vdash *$	well-formed row context
$\rho \vdash \tau : T$	well-formed type τ
$\rho \vdash R :: L$	row R has kind L
$\rho \vdash \tau_1 <: \tau_2$	type τ_1 subtype of τ_2
$\rho \vdash R_1 <:_B R_2$	row R_1 is a B-subtype of R_2
$\rho \, ; \Gamma \vdash *$	well-formed contexts
$\rho \, ; \Gamma \vdash e : \tau$	term e has type τ

where the B in the fifth judgment form is a *subtyping annotation* that denotes which subtyping relation has been applied between the two rows R_1 and R_2 (width, or width and depth). We will use the meta-judgment $H \vdash A$ to range over all of the above judgments. In addition, we use the meta-variable U to range over types τ and rows R. The meta-judgment $\rho \vdash U : -\gamma$ represents judgments of the form $\rho \vdash \tau : T$ and $\rho \vdash R :: L$. Similarly, meta-judgment $\rho \vdash U_1 <:_{(B)} U_2$ represents the judgments $\rho \vdash \tau_1 <: \tau_2$ and $\rho \vdash R_1 <:_B R_2$. Finally, we will use the syntax $\mathbf{probj.}R$ as shorthand for either $\mathbf{pro.}R$ or $\mathbf{obj.}R$.

3.3 Subtyping

Our subtyping system includes the standard rules for reflexivity, transitivity, and the contravariant rule for function subtyping. It also makes type *unit* the top type with respect to the subtyping order. In addition, we have a rule for subtyping between object types:

$$(<: obj) \qquad \frac{\rho \vdash R_1 <:_B R_2}{\rho \vdash \mathbf{probj.}R_1 <: \mathbf{obj.}R_2}$$

This rule is used both to "seal" objects, *i.e.*, promote them from a **pro**-state to an **obj**-state, and to infer subtyping relations between **obj**-types. Subtyping between objects occurs at the level of their component lists, *i.e.*, the rows. For example, we have the following rules to compute width and depth subtyping, respectively:

$$(<:w)\qquad \frac{\rho \vdash R_1 <:_B R_2 \qquad \rho \vdash \langle\!\langle R_1 \,|\, l : \tau \rangle\!\rangle :: L_1}{\rho \vdash \langle\!\langle R_1 \,|\, l : \tau \rangle\!\rangle <:_B R_2}$$

$$(<:d)\qquad \frac{\rho \vdash R_1 <:_B R_2 \qquad \rho \vdash \tau_1 <: \tau_2 \qquad \rho \vdash \langle\!\langle R_i \,|\, m : \tau_i \rangle\!\rangle :: L_i \qquad i \in \{1,2\}}{\rho \vdash \langle\!\langle R_1 \,|\, m : \tau_1 \rangle\!\rangle <:_{B+d} \langle\!\langle R_2 \,|\, m : \tau_2 \rangle\!\rangle}$$

Note that the depth subtyping rule applies only to methods. Because fields are updatable, we prevent depth subtyping from applying to fields to preserve type soundness. As this point illustrates, width and depth subtyping have different properties. It is therefore crucial that we control where depth subtyping can occur. For example, when we substitute a row R for an existentially or universally bound row variable r, we need to know that R is a width-only subtype of r's bound. To that end, we annotate the "$<:$" symbol with B:

$$B \ ::= \ w \ \mid \ w,d \ \mid \ B_1 + B_2$$

Here, w stands for "width-only" and w,d stands for "depth-allowed". Annotation w is less than w,d, and $+$ denotes the least upper bound of B_1 and B_2 with respect to this ordering. Our subtyping rules on rows are designed to track where depth subtyping occurs using these annotations.

The complete set of subtyping rules is shown in Appendix A.

3.4 Typing Rules

In this section, we discuss the key typing rules of the system. For reference, all typing rules appear in Appendix A, together with the standard rules for inferring the well-formedness of contexts, rows, and types.

We examine the (*meth ext*) rule in detail to illustrate how we use row-polymorphism:

$$(\mathit{meth\ ext})\qquad \frac{\rho\,;\Gamma \vdash e_1 : \mathbf{pro.}R \qquad \rho \vdash R :: L \qquad m \notin L \qquad \rho\,;\Gamma \vdash e_2 : \forall (r <:_w \langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: \mathit{All})\mathbf{pro.}r \rightarrow \tau}{\rho\,;\Gamma \vdash \langle e_1 \longleftrightarrow\!\!+\ m = e_2 \rangle : \mathbf{pro.}\langle\!\langle R \,|\, m : \tau \rangle\!\rangle}$$

Judgment $\rho\,;\Gamma \vdash e_1 : \mathbf{pro.}R$ asserts that expression e_1 is a prototype with methods listed in row R. The next judgment insures that row R has at most the labels in L; the side-condition assures us that m is not one of these labels. The final judgment guarantees that method body e_2 is appropriately polymorphic: it will map any possible extension of its current host object, called *self* and typed by $\mathbf{pro.}r$, to an expression of type τ, the type of method m. Type $\mathbf{pro}.r$ is an appropriate choice to represent the type of *self* since r may be instantiated with any row that has *at least* the methods in $\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$ and may have *any* other methods, reflected in its maximal kind *All*. Internal extensions (*i.e.*, extensions on *self*) are unsound, so they must be forbidden. The kind *All* in the (*meth ext*) typing rule asserts that *self* may already contain "all" possible methods, so it cannot be extended with any more.

The rule for method override is similar:

$$(\textit{meth over}) \qquad \frac{\begin{array}{c}\rho\,;\,\Gamma \vdash e_1 : \mathbf{pro}.R \\ \rho \vdash R <:_w \langle\!\langle m : \tau \rangle\!\rangle \\ \rho\,;\,\Gamma \vdash e_2 : \forall(r <:_w R :: \mathit{All})\mathbf{pro}.r \to \tau\end{array}}{\rho\,;\,\Gamma \vdash \langle e_1 \leftarrow m{=}e_2 \rangle : \mathbf{pro}.R}$$

The only difference between this rule and the previous one is that here the overridden method must be present in the object. This condition is guaranteed by the judgment $\rho \vdash R <:_w \langle\!\langle m : \tau \rangle\!\rangle$. Note the use of width-only subtyping.

The corresponding rules for fields are (*field ext*) and (*field over*). The differences between these rules and the corresponding ones for methods highlight one of the differences between methods and fields: method bodies are polymorphic, but field bodies are not. This difference arises from the fact that each method body takes its host object (*self*) as an argument so it can operate on that host. In contrast, fields cannot access their host and hence need not be polymorphic. Another difference is that field override on proper objects is sound because depth subtyping is not permitted on fields (see the (*field over*) rule, where the first premise allows fields to be overridden in both **pro**- and **obj**-typed objects).

Invocation of a method is typed by the following rule:

$$(\Leftarrow\ \textit{select}) \qquad \frac{\begin{array}{c}\rho\,;\,\Gamma \vdash e : \mathbf{probj}.R \\ \rho \vdash R <:_w \langle\!\langle m : \tau \rangle\!\rangle\end{array}}{\rho\,;\,\Gamma \vdash e \Leftarrow m : \tau}$$

The second judgment assures the presence of the method being invoked. The (*field select*) rule is similar.

The universal quantifier introduction rule ($\forall$ *intro*) and existential quantifier elimination rule ($\exists$ *elim*) are classical. The universal quantifier elimination rule

$$(\forall\ \textit{elim}) \qquad \frac{\begin{array}{c}\rho\,;\,\Gamma \vdash e : \forall(r <:_w R :: L)\tau \\ \rho \vdash R' <:_w R \\ \rho \vdash R' :: L\end{array}}{\rho\,;\,\Gamma \vdash e * () : [R'/r]\tau}$$

shows that the row variable r can be instantiated with any row that extends its upper bound row and has the proper kind. The existential quantifier introduction rule

$$(\exists\ intro) \qquad \frac{\rho \vdash R' <:_w R \qquad \rho \vdash R' :: L \qquad \rho\ ;\ \Gamma \vdash e : [R'/r]\tau}{\rho\ ;\ \Gamma \vdash \{\!|(r <:_w R :: L) = R',\ e|\!\} : \exists(r <:_w R :: L)\tau}$$

matches the semantics of data type implementations: row R' is the private interface of the data structure. It must be consistent with the public bound R and kind L, as required by the first two hypotheses. The final hypothesis requires expression e to have the appropriate type when r is replaced by R'.

4 Examples

4.1 Class Hierarchy

Using the extensible objects and abstract datatype mechanism from the previous sections, we may encode a simple class hierarchy of `Points` and `ColorPoints` as shown in Figure 1. The two abstract type declarations in Figure 1 define `Point` and `ColorPoint` classes, in that order. The `Point` **Abstype** expression introduces row variable p, which formally serves as the "name" of the `Point` class. Row p is bounded by row P_{pub}, `Point`'s public interface. This row, defined in Figure 1, lists the methods and fields available to clients and descendants of the `Point` class, namely `getX` and `setX`. The kind L_p, defined in Figure 1 to be label set $\{\mathtt{x}, \mathtt{getX}, \mathtt{setX}\}$, indicates that descendant classes may add any methods except ones with these names.

The **with** clause of the declaration specifies that the constructor for the `Point` class, named `newPoint`, will return expressions with type **pro.**p. The constructor returns a prototype so that descendant classes may define their objects via extension from the `Point` class constructor. The row variable p in this return type signals that objects returned by the constructor are defined in the `Point` class. We call such types *partially abstract* after [CW85] because a portion of their structure is hidden.

The **is** clause of the abstraction describes the implementation of the `Point` class. Within this implementation, the row P_{priv} describes the full representation of `Point` class objects by listing all of the components defined in the class and giving their associated types. The expression $\mathtt{Imp}_p$, defined towards the bottom of Figure 1, gives the implementation of the constructor `newPoint`. The type system insures that $\mathtt{Imp}_p$'s type is **pro.**p with p replaced by `Point`'s private interface, P_{priv}. The **Abstype** construct binds row variable p to P_{priv} and variable `newPoint` to $\mathtt{Imp}_p$ within the scope of its **in** clause.

The structure of the `ColorPoint` **Abstype** declaration is similar. There are several aspects of this declaration that are worth mentioning however. In particular, notice that the `ColorPoint` constructor, implemented via the $\mathtt{Imp}_{cp}$ function defined at the bottom of Figure 1, invokes `newPoint` to inherit `Point`'s implementation and initialization code. Notice also that the `ColorPoint` class publicly names the `Point`

$$
\begin{array}{rl}
\textbf{Abstype} & p <:_w P_{pub} :: L_p \\
\textbf{with} & \texttt{newPoint} : int \to \textbf{pro.}\,p \\
\textbf{is} & \{\!|p <:_w P_{pub} :: L_p = P_{priv},\ \texttt{Imp}_p|\!\} \\
\textbf{in} &
\end{array}
$$

$$
\begin{array}{rl}
\textbf{Abstype} & cp <:_w CP_{pub} :: L_{cp} \\
\textbf{with} & \texttt{newCPoint} : int \to color \to \textbf{pro.}\,cp \\
\textbf{is} & \{\!|cp <:_w CP_{pub} :: L_{cp} = CP_{priv},\ \texttt{Imp}_{cp}|\!\} \\
\textbf{in} &
\end{array}
$$

$$
\begin{array}{rl}
\textbf{let} & \texttt{newPoint} : int \to \textbf{obj.}\,p = \texttt{newPoint} \\
\textbf{let} & \texttt{newCPoint} : int \to color \to \textbf{obj.}\,cp = \texttt{newCPoint} \\
\textbf{in} & \langle\texttt{Program}\rangle
\end{array}
$$

where

$$
\begin{array}{lcl}
P_{pub} & \stackrel{def}{=} & \langle\!\langle \texttt{getX} : int,\ \texttt{setX} : int \to unit \rangle\!\rangle \\
P_{priv} & \stackrel{def}{=} & \langle\!\langle \texttt{x} : int,\ \texttt{getX} : int,\ \texttt{setX} : int \to unit \rangle\!\rangle \\
CP_{pub} & \stackrel{def}{=} & \langle\!\langle p \,|\, \texttt{getC} : color,\ \texttt{setC} : color \to unit \rangle\!\rangle \\
CP_{priv} & \stackrel{def}{=} & \langle\!\langle p \,|\, \texttt{c} : color,\ \texttt{getC} : color,\ \texttt{setC} : color \to unit \rangle\!\rangle \\
L_p & \stackrel{def}{=} & \{\texttt{x}, \texttt{getX}, \texttt{setX}\} \\
L_{cp} & \stackrel{def}{=} & \{\texttt{x}, \texttt{getX}, \texttt{setX}, \texttt{c}, \texttt{getC}, \texttt{setC}\}
\end{array}
$$

and

$$
\begin{array}{lcllcl}
\texttt{Imp}_p & \stackrel{def}{=} & \lambda\texttt{xi}.\langle\ \langle\rangle & \longleftarrow\!\!+\!: \ \texttt{x} & = & \texttt{xi}; \\
 & & & \longleftarrow\!\!+\ \texttt{getX} & = & \Lambda().\,\lambda\texttt{self}.(\texttt{self.x}); \\
 & & & \longleftarrow\!\!+\ \texttt{setX} & = & \Lambda().\,\lambda\texttt{self}.\,\lambda\texttt{newX}.\ \texttt{self.x} := \texttt{newX}\rangle;
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{Imp}_{cp} & \stackrel{def}{=} & \lambda\texttt{xi}.\lambda\texttt{ci}. \\
 & & \langle\,\texttt{newPoint xi} \longleftarrow\!\!+\!: \ \texttt{c} = \texttt{ci}; \\
 & & \qquad \longleftarrow\!\!+\ \texttt{getC} = \Lambda().\,\lambda\texttt{self}.(\texttt{self.c}); \\
 & & \qquad \longleftarrow\!\!+\ \texttt{setC} = \Lambda().\,\lambda\texttt{self}.\,\lambda\texttt{newC}. \\
 & & \qquad\qquad \texttt{self.c} := \texttt{newC}\rangle;
\end{array}
$$

Figure 1: Nested abstract datatypes for `Point` and `ColorPoint` classes.

class as its parent by listing p in its public interface CP_{pub}. Without thus naming its parent, the `ColorPoint` object type ($\mathbf{obj.}cp$) would not be a subtype of `Point`'s object type ($\mathbf{obj.}p$) because the only way to be a subtype of a partially abstract type is to extend that type. The row variable p in CP_{pub} signals that `ColorPoint` objects were formed via extension from `Point` prototypes, and hence their partially abstract types may be related via subtyping. This fact is the theoretical counterpart to the observation we made in the introduction: a portion of the subtyping hierarchy is determined by inheritance relationships.

Finally, the two **let** declarations in Figure 1 give constructors for the two classes that return proper objects, instead of prototypes. Technically, this shadowing is type correct because **pro** types are subtypes of **obj** types [FM95a].

4.2 Mixins

Because of the bounded universal quantifiers in our calculus, we may type *mixins*, functions that take objects as parameters and return extensions of those objects as results. For example, we may write a mixin that adds a `move` method to any object that already has *int* `getX` and $int \rightarrow unit$ `setX` methods:

$$
\begin{array}{l}
\textbf{let}\ \texttt{addMove}\ =\ \Lambda().\,\lambda\texttt{ob}.\langle\texttt{ob} \longleftarrow\!\!+ \texttt{move} = \Lambda().\,\lambda\texttt{s}.\,\lambda\texttt{dx}.\,\texttt{s} \Leftarrow \texttt{setX}(\texttt{dx} + \texttt{s} \Leftarrow \texttt{getX})\rangle\ \textbf{in}\\
\quad \textbf{let}\ \texttt{y}\ = \langle\ \langle\rangle \quad \longleftarrow\!\!+: \qquad \texttt{x} \ = \ \texttt{3},\\
\qquad\qquad\qquad\quad \longleftarrow\!\!+ \quad \texttt{getX} \ = \ \Lambda().\,\lambda\texttt{s}.\,\texttt{s.x},\\
\qquad\qquad\qquad\quad \longleftarrow\!\!+ \quad \texttt{setX} \ = \ \Lambda().\,\lambda\texttt{s}.\,\lambda\texttt{nx}.\langle\texttt{s.x} := \texttt{nx}\rangle\rangle\ \textbf{in}\\
\qquad \texttt{addMove} * ()\,\texttt{y}
\end{array}
$$

We may give the various program fragments the following types:

$$
\begin{array}{rcl}
\texttt{addMove} & : & \forall (r <:_w \langle\!\langle \texttt{getX} : int, \texttt{setX} : int \rightarrow unit \rangle\!\rangle :: L)\mathbf{pro.}r \rightarrow \mathbf{pro.}\langle\!\langle r \mid \texttt{move} : int \rightarrow unit \rangle\!\rangle\\
\texttt{y} & : & \mathbf{pro.}\langle\!\langle \texttt{x} : int, \texttt{getX} : int, \texttt{setX} : int \rightarrow unit \rangle\!\rangle\\
\texttt{addMove} * ()\,\texttt{y} & : & \mathbf{pro.}\langle\!\langle \texttt{x} : int, \texttt{getX} : int, \texttt{setX} : int \rightarrow unit, \texttt{move} : int \rightarrow unit \rangle\!\rangle
\end{array}
$$

In this example, L is any kind not containing the label `move`, as we are tracking components that may have already been added. (If we were tracking method absences, then the kind should *include* `move`.)

5 Operational Semantics

The operational semantics, presented in full in Appendix B, is inspired by the semantics for the imperative first-order calculus of [AC96a]. It is based solely on reduction relations, without any equational theory. The semantics relates expressions of the calculus to addresses of a global store in which the results of the evaluation are recorded. The Abadi-Cardelli calculus has only object terms and hence only one sort of result, which consists of sequences of store locations, one for each component of the object. Since our calculus is an extension of lambda calculus, we have values that are not objects (for instance, lambda abstractions and polymorphic functions). For the sake of uniformity, we have chosen to evaluate all terms to store addresses. These addresses then contain the results. In particular, the evaluation of an object term does not return a sequence of store locations but the address in which such a sequence of locations is stored. In the sequel, we will use the terms "location" and "address" synonymously. As in [AC96a], we avoid using formal substitution during reduction, keeping to standard implementation techniques for imperative languages.

In more detail, the semantics is based on *environments* and *stores*. An environment E associates term variables to addresses. A store S associates addresses to values. Values are the results to which expressions of the calculus evaluate (for instance, a lambda abstraction evaluates to a closure, an object to a sequence of labeled addresses, and so on). The operational semantics is expressed via reduction judgments of the form $\{\!|S;\, E|\!\} \models e \longrightarrow a \bullet S'$, indicating that expression e evaluates to address a starting from store S and environment E. In the process, store S is modified to produce store S'. The value associated with expression e is stored in location a of store S'.

For the most part, the reduction rules are intuitive. Rules (*lambda*) and ($\forall$ *intro*) evaluate term and row variable abstractions, respectively. Both rules store the appropriate closure in a fresh store location. Rule ($\leftarrow\!\ominus$), which appears below, describes how to evaluate method and field extensions and method override. Before discussing the rule, however, we need to introduce some notation. We use the symbol $\leftarrow\!\ominus$ as a meta-notation standing for any of $\longleftarrow\!\!+$, $\longleftarrow$, or $\longleftarrow\!\!+\!:$. Symbol η denotes a finite map from addresses to values; we use η to denote object values. The notation $\eta[l = a]$ denotes the finite map just like η except that $\eta[l = a]$ maps l to a. With this notation, we may now consider the rule:

$$(\leftarrow\!\ominus) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S_1;\, E|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \eta \qquad\qquad a_3 \notin dom(S_2)\end{array}}{\{\!|S;\, E|\!\} \models \langle e_1 \leftarrow\!\ominus\; l = e_2\rangle \longrightarrow a_3 \bullet (S_2, a_3 \mapsto \eta[l = a_2])}$$

Intuitively, we require e_1 to evaluate to an address containing an object (condition $S_2(a_1) = \eta$). Expression e_2 need only evaluate to an address. The type system will insure that $l \notin dom(\eta)$ if $\leftarrow\!\ominus$ is either $\longleftarrow\!\!+$ or $\longleftarrow\!\!+\!:$ and that $l \in dom(\eta)$ if $\leftarrow\!\ominus$ is $\longleftarrow$. It will also insure that $S_2(a_1)$ is an object value, not a closure. The result of the evaluation is then a modification of a "clone" of the targeted object, allocated in a fresh address (a_3). This cloning is necessary for type soundness of the extension operations.

It is worth remarking that the field override operation (for which we have a different rule ($:=$)) changes the value of the field of the object to which it is applied, instead of creating and modifying a clone. Hence field override is inherently imperative.

The rule for method invocation is slightly more complicated:

$$\frac{\begin{array}{c}\{\!|S;\, E|\!\} \models e \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = \eta \qquad\qquad \eta(m) = a_1' \\ S_1(a_1') = \langle \Lambda().\, e_1,\, E_1 \rangle \\ \{\!|S_1;\, E_1|\!\} \models e_1 \longrightarrow a_2 \bullet S_2 \\ S_2(a_2) = \langle \lambda x.\, e_2,\, E_2 \rangle \qquad x \notin dom(E_2) \\ \{\!|S_2;\, E_2, x \mapsto a_1|\!\} \models e_2 \longrightarrow a_3 \bullet S_3\end{array}}{\{\!|S;\, E|\!\} \models e \Leftarrow m \longrightarrow a_3 \bullet S_3} \quad (\Leftarrow)$$

First, we evaluate the receiver object to its address a_1 to obtain η, the list of methods and fields associated with object e. We then look up the row closure associated with method m in η and evaluate its body e_1 in the appropriate environment to find the term closure that stores the method body $\lambda x.\, e_2$. We then evaluate e_2 with x bound to the address of the receiver object in the appropriate environment. This binding is where the first term parameter to a method body is bound to its host object, modeling *self*. The field selection evaluation rule (*field selection*) is simpler, since field bodies are not polymorphic.

It is interesting to look at the rule:

$$\frac{\begin{array}{c}\{\!|S;\, E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = a_1' \qquad x \notin dom(E) \\ \{\!|S_1;\, (E, x \mapsto a_1')|\!\} \models e_2 \longrightarrow a_2 \bullet S_2\end{array}}{\{\!|S;\, E|\!\} \models \textbf{Abstype}\ (r <:_w R :: L)\ \textbf{with}\ x : \tau\ \textbf{is}\ e_1\ \textbf{in}\ e_2 \longrightarrow a_2 \bullet S_2} \quad (\exists\ elim)$$

Here, a_1' represents the store location where the (hidden) implementation of the data declaration is stored. Its symmetric rule ($\exists$ *intro*) is quite straightforward.

6 Soundness Proof

To show that the operational semantics is consistent with the type system, we must give types to results, *i.e.*, give a type to each store location. As noted in Chapter 11 of [AC96b], it is impossible to determine the type of a result simply by recursively examining its subcomponents because stores may contain loops. To address this problem, we follow Abadi and Cardelli in introducing *store typings* to allow us to type results independently of a particular store. This approach is reasonable because type-safe computations store results of different types in different locations. Intuitively, all store locations are typed "globally" with respect to the whole store. Store types and the machinery to prove subject reduction are complicated in our case by the presence of types with free row variables. Since it only makes sense to give store locations closed types, we need:

- a universal quantifier to close method body types;

- a *row environment substitution*, denoted by $[E_\rho \rightarrow]$, to enable comparisons between static types that contain free row variables and run-time store types (Appendix C.2) that do not. This technique was inspired by the substitution operation in Chapter 17 of [AC96b] for their imperative calculus with *Self* type.

Using these tools, we may show that our system enjoys a subject reduction property. Informally, we may state this property as follows:

> If we may statically give expression e type τ and e evaluates in store S with store type Σ to a result stored in address a of the modified store S', then there exists a type τ' and a store type Σ' such that τ' subtypes (a closed version of) type τ and Σ' extends Σ, correctly types S', and gives address a type τ' (*i.e.*, $\Sigma(a) = \tau'$).

The formal statement of subject reduction is Theorem D.6 in Appendix D.

Using the subject reduction property, we can prove that our system is type sound, in the sense that if a closed expression is typable, its evaluation will not reach a *stuck* state. More formally, given a store S, an environment E, and an expression e, there are three possibilities:

- $\{\!\!\{S; E\}\!\!\} \models e \longrightarrow a \bullet S'$ for some location a and store S';

- the proof tree generated by the reduction rules is infinite, *i.e.*, the program runs for ever;

- the proof tree is finite, but it does not produce a conclusion for S, E, and e, *i.e.*, the program gets stuck. This can happen, for example, if a method is invoked on an object that does not contain such method.

We have formally defined the concept of *partial* evaluation by means of the *depth* of a deduction in the operational semantics. If the depth is finite, then the evaluation converges to a location or it gets stuck. For lack of space, we state the soundness theorem without further formal details. These details may be found in [BF97].

Theorem 6.1 (Type Soundness) *If judgment $\varepsilon\,; \emptyset \vdash e : \tau$ is derivable and the evaluation of e does not diverge, then $\{\!\!\{\emptyset; \emptyset\}\!\!\} \models e \longrightarrow a \bullet S$ for some location a and some store S well typed with respect to a store type Σ such that $\Sigma(a) <: \tau$.*

7 The Typing Algorithm

In this section, we sketch a typing algorithm for our system. Formal details can be found in the full version of our paper [BF97]. Our system results from the composition of seven interleaved subsystems: well-formedness of row and term contexts, kinding of rows, well-formedness of types, row subtyping, type subtyping, and typing expressions. Consequently, we must give a sound and complete algorithm for each of these subsystems. Our work was inspired by Compagnoni [Com94] and Hofmann and Pierce [HP96]; although their type systems are quite different from ours, they demonstrated a useful structure for developing typing algorithms.

The first step is to define an *algorithmic* version of our type system S, *i.e.*, a type system S' in which the last rule applied is unequivocally determined by the syntax of the expression in question. For example, in the subtyping subsystem, producing a syntax-directed system requires eliminating transitivity and restricting reflexivity to atomic types. In the row-kinding subsystem, we must develop a system that calculates *minimal* kinds. For expressions, we must produce *minimal* types, *etc.* In addition to modifying the typing rules to produce equivalent algorithmic versions, we must add type annotations to some of our expressions in S'. For example, in S' λ-abstraction is expressed as $\lambda x{:}\tau.e$. An inspection of the *Type* algorithm below reveals where the other annotations must be added. These annotations are needed to insure that the minimal typing property holds for expressions. Although we omit the details here, we have proven that S and S' are appropriately related.

The second step is represented by the algorithms given in Definitions 7.1 to 7.4, one for each subsystem. In the algorithms, we use the following notation:

- $WFR(\rho; R)\downarrow$ stands for $WFR(\rho; R) = L$, for some L. ($WFR(\rho; R)$ is defined in Definition 7.1).
- $label(r, \rho)$ is L whenever $(r <:_w R \;::L) \in \rho$.
- $bound(r, \rho)$ is R whenever $(r <:_w R \;::L) \in \rho$.
- $B \leq B'$ means that the subtyping annotation B is less than or equal to the the subtyping annotation B', according to the order given in Section 3.3.
- $\Gamma(x)$ stands for the type of x in Γ.
- $WB_l(\rho, R)$ returns the type of l in R.

Definition 7.1 (Well-Formedness Algorithms) *Well-formedness algorithm for row contexts (*$WFRC(\rho)$*:* `bool`*) and types (*$WFT(\rho;\tau): \texttt{bool}$*), and minimal kind algorithm for rows (*$WFR(\rho; R) = L$*, for some kind* L*, or* `error`*):*

$$
\begin{array}{lcl}
WFRC(\epsilon) & = & \texttt{true} \\
WFRC(\rho, (r <:_w R::L)) & = & WFRC(\rho) \wedge WFR(\rho, R) \subseteq L \wedge r \notin dom(\rho) \\
\\
WFT(\rho; unit) & = & WFRC(\rho) \\
WFT(\rho; \tau_1 \to \tau_2) & = & WFT(\rho; \tau_1) \wedge WFT(\rho; \tau_2) \\
WFT(\rho; \textbf{probj}.R) & = & WFR(\rho, R)\downarrow \\
WFT(\rho; \forall(r <:_w R::L)\tau) & = & WFT(\rho, (r <:_w R::L); \tau) \\
WFT(\rho; \exists(r <:_w R::L)\tau) & = & WFT(\rho, (r <:_w R::L); \tau)
\end{array}
$$

$$
\begin{array}{lcl}
WFR(\rho; r) & = & \texttt{if } WFRC(\rho) \wedge r \in dom(\rho) \texttt{ then } label(r,\rho) \texttt{ else error} \\
WFR(\rho; \langle\!\langle\rangle\!\rangle) & = & \texttt{if } WFRC(\rho) \texttt{ then } \emptyset \texttt{ else error} \\
WFR(\rho; \langle\!\langle R \mid l : \tau \rangle\!\rangle) & = & \texttt{if } WFT(\rho;\tau) \wedge l \notin WFR(\rho;R) \texttt{ then } \{l\} \cup WFR(\rho;R) \\
 & & \texttt{ else error};
\end{array}
$$

Definition 7.2 (Subtyping Algorithms) *Algorithm for row subtyping ($SR(\rho, R_1, R_2, B) : \texttt{bool}$) and type subtyping ($ST(\tau_1, \tau_2) : \texttt{bool}$):*

$$
\begin{array}{lcl}
SR(\rho, \langle\!\langle\rangle\!\rangle, R, B) & = & \texttt{if } R = \langle\!\langle\rangle\!\rangle \;\wedge\; WFRC(\rho) \texttt{ then true} \\
 & & \texttt{ else false} \\
SR(\rho, r, R, B) & = & \texttt{if } R \neq r \;\wedge\; WFR(\rho, r)\downarrow \texttt{ then} \\
 & & SR(\rho, bound(r,\rho), R, B) \texttt{ else false} \\
SR(\rho, r, r, B) & = & WFR(\rho; r)\downarrow \\
SR(\rho, \langle\!\langle R_1 \mid l : \tau \rangle\!\rangle, \langle\!\langle R_2 \mid l : \tau \rangle\!\rangle, B) & = & SR(R_1, R_2, B) \;\wedge\; WFR(\rho, \langle\!\langle R_1 \mid l : \tau \rangle\!\rangle)\downarrow \;\wedge \\
 & & WFR(\rho, \langle\!\langle R_2 \mid l : \tau \rangle\!\rangle)\downarrow \\
SR(\rho, \langle\!\langle R_1 \mid l : \tau_1 \rangle\!\rangle, \langle\!\langle R_2 \mid l : \tau_2 \rangle\!\rangle, B') & = & \tau_1 \neq \tau_2 \;\wedge\; SR(\rho, R_1, R_2, B) \;\wedge\; ST(\rho, \tau_1, \tau_2) \\
 & & \wedge\; WFR(\rho, \langle\!\langle R_1 \mid l : \tau_1 \rangle\!\rangle)\downarrow \;\wedge\; WFR(\rho, \langle\!\langle R_2 \mid l : \tau_2 \rangle\!\rangle)\downarrow \\
 & & \wedge\; B' = B + d \\
SR(\rho, \langle\!\langle R_1 \mid l : \tau_1 \rangle\!\rangle, R_2, B) & = & \texttt{if } R_2 \neq \langle\!\langle R_1' \mid l : \tau_1' \rangle\!\rangle \textit{ for any } R_1' \textit{ and } \tau_1' \\
 & & \texttt{ then } SR(\rho, R_1, R_2, B) \;\wedge\; WFR(\rho, \langle\!\langle R_1 \mid l : \tau_1 \rangle\!\rangle)\downarrow \\
 & & \texttt{ else false}; \\
ST(\rho, \tau, unit) & = & WFT(\rho, \tau) \\
ST(\rho, \tau_1 \to \tau_2, \tau_1' \to \tau_2') & = & ST(\rho, \tau_1', \tau_1) \;\wedge\; ST(\tau_2, \tau_2') \\
ST(\rho, \mathbf{probj.}R_1, \mathbf{obj.}R_2) & = & SR(\rho, R_1, R_2, B) \textit{ for some } B \\
ST(\rho, \forall(r <:_w R :: L)\tau, \forall(r <:_w R :: L)\tau) & = & WFT(\rho, \forall(r <:_w R :: L)\tau) \\
ST(\rho, \exists(r <:_w R :: L)\tau, \exists(r <:_w R :: L)\tau) & = & WFT(\rho, \exists(r <:_w R :: L)\tau)
\end{array}
$$

Definition 7.3 (Well-formed Term Contexts) *Algorithm for well-formed term contexts: $WFEC(\rho;\Gamma)$: $\texttt{bool}$;*

$$
\begin{array}{lcl}
WFEC(\rho; \varepsilon) & = & WFRC(\rho) \\
WFEC(\rho; \Gamma, x : \tau) & = & WFEC(\rho;\Gamma) \wedge WFT(\rho;\tau) \wedge x \notin dom(\Gamma);
\end{array}
$$

Definition 7.4 (Minimal Typing Algorithm) Minimal *type algorithm for terms: $Type(\rho;\Gamma;e) = \tau$, for minimal type τ with respect to subtyping, or* `error`*;*

$$
\begin{array}{l}
Type(\rho;\Gamma;x) \quad = \\
\quad \texttt{if } WFEC(\rho;\Gamma) \;\wedge\; x \in dom(\Gamma) \texttt{ then } \Gamma(x) \texttt{ else error}
\end{array}
$$

$Type(\rho;\Gamma;\langle\rangle) \quad =$

if $WFEC(\rho;\Gamma)$ then $\mathbf{pro.}\langle\!\langle\rangle\!\rangle$ else error

$Type(\rho;\Gamma;\lambda x{:}\tau.e) \quad =$

if $Type(\rho;\Gamma,x{:}\tau;\,e) = \tau_2$ then $\tau \rightarrow \tau_2$ else error

$Type(\rho;\Gamma;e_1 e_2) \quad =$

if $Type(\rho;\Gamma;e_1) = \tau_1 \rightarrow \tau_2 \ \wedge \ Type(\rho;\Gamma;e_2) = \tau_1' \ \wedge \ ST(\rho;\tau_1',\tau_1)$ then τ_2 else error

$Type(\rho;\Gamma;\mathbf{let}\ x{:}\tau \ = \ e_1 \ \mathbf{in}\ e_2) \quad =$

if $Type(\rho;\Gamma;e_1) = \tau_1' \ \wedge \ ST(\rho,\tau_1',\tau) \ \wedge \ Type(\rho;\Gamma,x:\tau;e_2) = \tau_2'$ then τ_2' else error

$Type(\rho;\Gamma;\langle e_1 \longleftrightarrow\!\!+\ m = e_2\rangle) \quad =$

if $Type(\rho;\Gamma;e_1) = \mathbf{pro.}R \ \wedge \ WFR(\rho,R) = L \ \wedge \ m \notin L \ \wedge$

$\quad Type(\rho;\Gamma;e_2) = \forall(r <:_w \langle\!\langle R \mid m{:}\tau\rangle\!\rangle ::All)\mathbf{pro.}r \rightarrow \tau$

then $\mathbf{pro.}\langle\!\langle R \mid m{:}\tau\rangle\!\rangle$ else error

$Type(\rho;\Gamma;\langle e_1 \longleftrightarrow\!\!+\!:\ f{:}\tau = e_2\rangle) \quad =$

if $Type(\rho;\Gamma;e_1) = \mathbf{pro.}R \wedge WFR(\rho,R) = L \wedge f \notin L \ \wedge$

$\quad Type(\rho;\Gamma;e_2) = \tau' \ \wedge \ ST(\rho,\tau',\tau)$

then $\mathbf{pro.}\langle\!\langle R \mid f{:}\tau\rangle\!\rangle$ else error

$Type(\rho;\Gamma;\langle e_1 \leftarrow m = e_2\rangle) \quad =$

if $Type(\rho;\Gamma;e_1) = \mathbf{pro.}R \ \wedge \ WB_m(\rho,R) = \tau \ \wedge \ Type(\rho;\Gamma;e_2) = \forall(r <:_w R ::All)\mathbf{pro.}r \rightarrow \tau$

then $\mathbf{pro.}R$ else error

$Type(\rho;\Gamma;\langle e_1 := f = e_2\rangle) \quad =$

if $Type(\rho;\Gamma;e_1) = \mathbf{probj.}R \ \wedge \ WB_f(\rho,R) = \tau \ \wedge \ Type(\rho;\Gamma;e_2) = \tau_2 \ \wedge \ ST(\rho,\tau_2,\tau)$

then $\mathbf{probj.}R$ else error

$Type(\rho;\Gamma;e \Leftarrow m) \quad =$

if $Type(\rho;\Gamma;e) = \mathbf{probj.}R \ \wedge \ WB_m(\rho,R) = \tau$ then τ else error

$Type(\rho;\Gamma;e.f) \quad =$

if $Type(\rho;\Gamma;e) = \mathbf{probj.}R \ \wedge \ WB_f(\rho,R) = \tau$ then τ else error

$Type(\rho;\Gamma;\Lambda(r <:_w R \ ::L).\,e{:}\tau) \quad =$

if $Type(\rho,(r <:_w R \ ::L);\Gamma;e) = \tau' \ \wedge \ ST(\rho,(r <:_w R \ ::L);\Gamma;\tau';\tau) \ \wedge \ r \notin FV(\Gamma)$

then $\forall(r <:_w R \ ::L)\tau$ else error

$Type(\rho;\Gamma;e * R') \quad =$

if $Type(\rho;\Gamma;e) = \forall(r <:_w R \ ::L)\tau \ \wedge \ SR(\rho,R',R,w) \ \wedge \ WFR(\rho,R') = L' \ \wedge \ L' \subseteq L$

then $[R'/r]\tau$ else error

$Type(\rho;\Gamma;\{\!|r <:_w R \ ::L = R',\ e{:}\tau|\!\})) \quad =$

`if` $SR(\rho, R', R) \ \wedge \ WFR(\rho, R') = L' \ \wedge \ L' \subseteq L \ \wedge \ Type(\rho;\Gamma;e) = \tau' \wedge ST(\rho;\tau';[R'/r]\tau)$

`then` $\exists(r <:_w R \ ::L)\tau$ `else error`

$Type(\rho;\Gamma;$ **Abstype** $r <:_w R \ ::L$ **with** $x : \tau$ **is** e_1 **in** $e_2) \quad =$

`if` $Type(\rho;\Gamma;e_1) = \exists(r <:_w R \ ::L)\tau \ \wedge \ Type(\rho,(r <:_w R \ ::L);\Gamma, x:\tau;e_2) = \tau_2' \ \wedge \ WFT(\rho, \tau_2')$

`then` τ_2' `else error`

We prove the correctness and completeness of these algorithms in the following theorem.

Theorem 7.1 (Soundness and Completeness)

- *For all row contexts ρ, $WFRC(\rho) =$ `true` if and only if $\rho \vdash *$;*
- *For all row contexts ρ and all types τ, $WFRC(\rho;\tau) =$ `true` if and only if $\rho \vdash \tau : T$;*
- *For all row contexts ρ and all rows R, $WFR(\rho) = L$ if and only if $\rho \vdash R::L$ and for all L' such that $\rho \vdash R::L'$ is derivable, $L \subseteq L'$;*
- *For all row contexts ρ and all term contexts Γ, $WFEC(\rho;\Gamma) =$ `true` if and only if $\rho;\Gamma \vdash *$;*
- *For all row contexts ρ, all rows R_1, R_2, and all subtyping labels B, $SR(\rho, R_1, R_2, B) =$ `true` if and only if $\rho \vdash R_1 <:_{B'} R_2$ where $B' \leq B$;*
- *For all row contexts ρ and all types τ_1, τ_2, $ST(\rho, \tau_1, \tau_2) =$ `true` if and only if $\rho \vdash \tau_1 <: \tau_2$;*
- *For all row contexts ρ, all term contexts Γ, and all terms e, $Type(\rho;\Gamma;e) = \tau$ if and only if $\rho\,;\,\Gamma \vdash e : \tau$ and for all types τ' such that $\rho\,;\,\Gamma \vdash e : \tau'$ is derivable, we have $\rho \vdash \tau <: \tau'$.*

Finally, we prove that each algorithm given in Definitions 7.1 to 7.4 terminates with the correct answer if one exists or with `error` otherwise. To prove this theorem, we define appropriate measures on each sort of expression and then proceed by induction on these measures.

Theorem 7.2 (Termination) *For all ρ, τ, τ_1, τ_2, R, R_1, R_2, Γ, B, and e, the functions $WFRC(\rho)$, $WFT(\rho;\tau)$, $WFR(\rho;R)$, $WFEC(\rho;\Gamma)$, $SR(\rho, R_1, R_2, B)$, $ST(\rho, \tau_1, \tau_2)$, and $Type(\rho;\Gamma;e)$ all converge.*

8 Related Work

In the literature, there are various approaches to modeling classes other than ours and the record-of-premethods approach that we briefly discussed in Section 1.1:

- *Existential Model*: In [PT94], Pierce and Turner interpret classes as object-generating functions. In their interpretation, inheritance is interpreted as modifications to the object-generating functions that model classes. This encoding is somewhat cumbersome, since it requires programmers to explicitly manipulate `get` and `put` functions, which intuitively convert between the hidden state of parent class objects and derived class objects. Also, because this model provides protection at the object-level, as opposed to the class-level, binary methods require extra machinery. One such solution appears in [PT93]. Another solution is proposed in [HP96]; Hofmann and Pierce introduce a refined version of $F_{<:}$ that permits only positive subtyping. With this restriction, `get` and `put` functions are both guaranteed to exist and hence may be handled in a more automatic fashion in class encodings.

- *Direct Models*: Kim Bruce has developed a family of type-safe formal languages that model classes directly instead of via an interpretation as the combination of more basic primitives. In [Bru93], Bruce describes TOOPL, a functional object-oriented language. PolyTOIL, presented in [BSv95], incorporates imperative features and introduces the notion of *matching*, a relationship between object types that holds whenever the first is an extension of the second, regardless of the variance of the "mytype" type variable. In these languages, the type of an object reflects only its public interface; it cannot convey implementation information. Scott Smith and the Hopkins Object Group have designed a type-safe class-based object-oriented language with a rich feature set called I-Loop, [EST95]. Their type system is based on polymorphic recursively constrained types, for which they have a sound type inferencing algorithm. The main advantage of this approach is the extreme flexibility afforded by recursively constrained types. Currently, the main problem is that it returns large, difficult-to-read types. Some form of simplification may be required. Work in this area is in progress.

Our system falls in the category of prototype calculi with subtyping. We briefly mention here three other approaches to extensible objects compatible with some form of subtyping.

- In [BL95] a type system is presented for the functional Lambda Calculus of Objects [FHM94], in which a limited form of width subtyping compatible with object extension is introduced. Only methods that are not invoked by other methods in the object can be "forgotten" by subtyping. The machinery to track information about used methods is somewhat heavy, but it avoids constraints on the variance of the *self*-type, which is instead one of the weak points of [FM95a].

- The type system of [AC96b] is extended in [Liq96] to allow a form of extension in the presence of a limited form of width subtyping. This approach combines a variant of [BL95] with that of [FM95a]: an object may be in one of two states, one allowing extensions and limited width subtyping, the other forbidding extensions but supporting unrestricted width subtyping.

- PolyTOIL combines a form of (match-bounded) universal quantification and extensible structures. It uses the matching relation on types to type check inheritance of extensible classes and uses match-bound quantification to type check polymorphic functions.

9 Conclusion

We have presented a formal study of a calculus to support the "Classes = Extensible Objects + Encapsulation" interpretation of classes. Despite its apparent complexity (although this impression may be caused more by the number of rules than by the complexity of the rules themselves), the calculus is rather intuitive, since:

- it uses well-established concepts such as universal and existential quantifiers,
- it enables straightforward translations of Java-like classes and mixins, and
- it has a clear imperative semantics and a well-developed formal meta-theory, including a soundness theorem and a typing algorithm.

We have not completed our study of the system. Future work includes:

- adopting a module construct and a dot notation as in [CL90, HL94] to replace the present nested structure of class hierarchies, and
- introducing a form of alpha-conversion to allow us to hide the private names that are currently exposed by our positive kinds, as in [RS98].

References

[AC96a] M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1996. Earlier version appeared in TAPSOFT '95 proceedings.

[AC96b] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[AC96c] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, 1996. Earlier version appeared in TACS '94 proceedings, LNCS 789.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[BBL96] V. Bono, M. Bugliesi, and L. Liquori. A lambda calculus of incomplete objects. In *Proc. MFCS '96*, pages 218–229. Springer LNCS 1113, 1996.

[BDMN73] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, Box 1717, S-222 01 Lund, Sweden; Auerbach, Philadelphia, 1973.

[BF97] V. Bono and K. Fisher. A first-order, extensible-object calculus with support for classes. Unpublished manuscript; work-in-progress, 1997.

[BL95] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In L. Pacholsky and J. Tiuryn, editors, *Proc. of Int'l Conf. of Computer Science Logic*, pages 16–30, Berlin, June 1995. Springer LNCS 933.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 285–298, 1993.

[BSv95] K. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proc. 9th European Conference on Object-Oriented Programming*, pages 26–51, Aarhus, Denmark, 1995. Springer LNCS 952.

[CL90] L. Cardelli and X. Leroy. *Abstract types and the dot notation*, pages 479–504. IFIP State of the Art Reports. North Holland, March 1990. Also appeared as SRC Research Report 56.

[Com94] A.B. Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, Katholieke Universiteit Nijmegen, 1994.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C^{++} Reference Manual.* Addison-Wesley, 1990.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, pages 169–184, October 1995.

[FHM94] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[Fis96] K. Fisher. *Type Systems for object-oriented programming languages.* PhD thesis, Stanford University, 1996.

[FM95a] K. Fisher and J.C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th Int'l Conf. Fundamentals of Computation Theory (FCT'95)*, pages 42–61. Springer LNCS 965, 1995.

[FM95b] K. Fisher and J.C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1995. Preliminary version appeared in TACS '94 proceedings.

[FM98] K. Fisher and J.C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 1998. To Appear. Preliminary version appeared in Marktoberdorf '97 proceedings.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, 1994.

[HP96] M. Hofmann and B.C. Pierce. Positive subtyping. *Information and Computation*, 126(1):11–33, 1996. Preliminary version appeared in *Proc. 22nd ACM Symp. on Principles of Programming Languages*, 1995.

[KLM94] D. Katiyar, D. Luckham, and J.C. Mitchell. A type system for prototyping languages. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, 1994.

[Liq96] L. Liquori. An extended theory of primitive objects: First and second order systems. Technical Report CS-23-96, Dipartimento di Informatica, Universitá di Torino, 1996. A portion of this work appears in ECOOP '97 Proceedings, LNCS 1241.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[PT93] B.C. Pierce and D.N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[PT94] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994. Preliminary version appeared in *Proc. 20th ACM Symp. on Principles of Programming Languages*, 1993, under the title "Object-oriented programming without recursive types".

[RR96] J.H. Reppy and J.G. Riecke. Classes in Object ML via modules, 1996. Presented at FOOL3 workshop.

[RS98] J.G. Riecke and C. Stone. Privacy via subsumption. In *Proc. 5th Annual FOOL Workshop*, January 1998. To appear.

[Str94] B. Stroustrup. *The Design and Evolution of* C^{++}. Addison-Wesley, 1994. Chapter 3: The birth of C^{++}.

[Wan78] M. Wand. A new incompleteness result for Hoare's system. *J. ACM*, 25(1):168–175, Jan. 1978.

A Typing rules

A.1 Context Rules

$$(start)\quad \frac{}{\epsilon \vdash *}$$

$$(exp\ var)\quad \frac{\rho\ ;\ \Gamma \vdash * \qquad \rho \vdash \tau : T \qquad x \notin dom(\Gamma)}{\rho\ ;\ \Gamma,\ x : \tau \vdash *}$$

$$(row\ var)\quad \frac{\rho \vdash * \qquad r \notin dom(\rho) \qquad \rho \vdash R :: L_1 \qquad L_1 \subseteq L_0}{\rho\ ,\ (r <:_w R :: L_0) \vdash *}$$

$$(empty\ exp)\quad \frac{\rho \vdash *}{\rho\ ;\ \epsilon \vdash *}$$

A.2 Rules for Type Expressions

(*type unit*) $$\frac{\rho \vdash *}{\rho \vdash unit : T}$$

($\forall$) $$\frac{\rho\,,\,(r <:_w R :: L) \vdash \tau : T}{\rho \vdash \forall(r <:_w R :: L)\tau : T}$$

(*type arrow*) $$\frac{\rho \vdash \tau_1 : T \quad \rho \vdash \tau_2 : T}{\rho \vdash \tau_1 \to \tau_2 : T}$$

($\exists$) $$\frac{\rho\,,\,(r <:_w R :: L) \vdash \tau : T}{\rho \vdash \exists(r <:_w R :: L)\tau : T}$$

(*probj*) $$\frac{\rho \vdash R :: L}{\rho \vdash \mathbf{probj}.R : T}$$

A.3 Rules for Row Expressions

(*row proj*) $$\frac{\rho \vdash * \quad r <:_w R :: L \in \rho}{\rho \vdash r :: L}$$

(*row label*) $$\frac{\rho \vdash R :: L_1 \quad L_1 \subseteq L_2}{\rho \vdash R :: L_2}$$

(*empty row*) $$\frac{\rho \vdash *}{\rho \vdash \langle\!\langle\rangle\!\rangle :: \emptyset}$$

(*row ext*) $$\frac{\rho \vdash R :: L \qquad l \notin L \quad \rho \vdash \tau : T}{\rho \vdash \langle\!\langle R \mid l : \tau \rangle\!\rangle :: L \cup \{l\}}$$

A.4 Subtyping Rules for Types

($<:\to$) $$\frac{\rho \vdash \tau_1' <: \tau_1 \quad \rho \vdash \tau_2 <: \tau_2'}{\rho \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

($<:$ *unit*) $$\frac{\rho \vdash \tau : T}{\rho \vdash \tau <: unit}$$

($<:$ *proj*) $$\frac{\rho\,,\iota_1 <: \iota_2 \vdash *}{\rho\,,\iota_1 <: \iota_2 \vdash \iota_1 <: \iota_2}$$

($<:$ *obj*) $$\frac{\rho \vdash R_1 <:_B R_2}{\rho \vdash \mathbf{probj}.R_1 <: \mathbf{obj}.R_2}$$

A.5 Subtyping Rules for Rows

(*row proj bound*) $$\frac{\rho \vdash * \quad r <:_w R :: L \in \rho}{\rho \vdash r <:_w R}$$

($<: d$) $$\frac{\rho \vdash R_1 <:_B R_2 \qquad \rho \vdash \tau_1 <: \tau_2 \quad \rho \vdash \langle\!\langle R_i \mid m : \tau_i \rangle\!\rangle :: L_i \qquad i \in \{1,2\}}{\rho \vdash \langle\!\langle R_1 \mid m : \tau_1 \rangle\!\rangle <:_{B+d} \langle\!\langle R_2 \mid m : \tau_2 \rangle\!\rangle}$$

($<:$ *cong*) $$\frac{\rho \vdash R_1 <:_B R_2 \quad \rho \vdash \langle\!\langle R_i \mid l : \tau \rangle\!\rangle :: L_i \qquad i \in \{1,2\}}{\rho \vdash \langle\!\langle R_1 \mid l : \tau \rangle\!\rangle <:_B \langle\!\langle R_2 \mid l : \tau \rangle\!\rangle}$$

($<: w$) $$\frac{\rho \vdash R_1 <:_B R_2 \quad \rho \vdash \langle\!\langle R_1 \mid l : \tau \rangle\!\rangle :: L_1}{\rho \vdash \langle\!\langle R_1 \mid l : \tau \rangle\!\rangle <:_B R_2}$$

A.6 Subtyping Rules for Rows and Types

$$(<:\ \mathit{refl})\qquad \frac{\rho \vdash U : -\gamma}{\rho \vdash U <:_{(B)} U}$$

$$(<:\ \mathit{trans})\qquad \frac{\begin{array}{c}\rho \vdash U_1 <:_{(B)} U_2\\ \rho \vdash U_2 <:_{(B')} U_3\end{array}}{\rho \vdash U_1 <:_{(B+B')} U_3}$$

A.7 Rules for Assigning Types to Terms

$$(\mathit{exp\ proj})\qquad \frac{\begin{array}{c}\rho\ ;\ \Gamma \vdash *\\ x : \tau \in \Gamma\end{array}}{\rho\ ;\ \Gamma \vdash x : \tau}$$

$$(\mathit{empty\ pro})\qquad \frac{\rho\ ;\ \Gamma \vdash *}{\rho\ ;\ \Gamma \vdash \langle\rangle : \mathbf{pro}.\langle\!\langle\rangle\!\rangle}$$

$$(\mathit{exp\ abs})\qquad \frac{\rho\ ;\ \Gamma,\ x : \tau_1 \vdash e : \tau_2}{\rho\ ;\ \Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$(\mathit{exp\ app})\qquad \frac{\rho\ ;\ \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \rho\ ;\ \Gamma \vdash e_2 : \tau_1}{\rho\ ;\ \Gamma \vdash e_1\, e_2 : \tau_2}$$

$$(\mathit{subsumption})\qquad \frac{\rho\ ;\ \Gamma \vdash e : \tau_1 \quad \rho \vdash \tau_1 <: \tau_2}{\rho\ ;\ \Gamma \vdash e : \tau_2}$$

$$(\mathit{let})\qquad \frac{\begin{array}{c}\rho\ ;\ \Gamma \vdash e_1 : \tau_1\\ \rho\ ;\ \Gamma,\ x : \tau_1 \vdash e_2 : \tau_2\end{array}}{\rho\ ;\ \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$(\mathit{meth\ ext})\qquad \frac{\begin{array}{l}\rho\ ;\ \Gamma \vdash e_1 : \mathbf{pro}.R\\ \rho \vdash R :: L \qquad\qquad m \notin L\\ \rho\ ;\ \Gamma \vdash e_2 : \forall(r <:_w \langle\!\langle R \mid m : \tau\rangle\!\rangle :: \mathit{All})\mathbf{pro}.r \to \tau\end{array}}{\rho\ ;\ \Gamma \vdash \langle e_1 \longleftrightarrow\!\!+\ m = e_2\rangle : \mathbf{pro}.\langle\!\langle R \mid m : \tau\rangle\!\rangle}$$

$$(\mathit{field\ ext})\qquad \frac{\begin{array}{l}\rho\ ;\ \Gamma \vdash e_1 : \mathbf{pro}.R\\ \rho \vdash R :: L \qquad\qquad f \notin L\\ \rho\ ;\ \Gamma \vdash e_2 : \tau\end{array}}{\rho\ ;\ \Gamma \vdash \langle e_1 \longleftrightarrow\!\!+\!:\ f = e_2\rangle : \mathbf{pro}.\langle\!\langle R \mid f : \tau\rangle\!\rangle}$$

$$(\mathit{meth\ over})\qquad \frac{\begin{array}{l}\rho\ ;\ \Gamma \vdash e_1 : \mathbf{pro}.R\\ \rho \vdash R <:_w \langle\!\langle m : \tau\rangle\!\rangle\\ \rho\ ;\ \Gamma \vdash e_2 : \forall(r <:_w R :: \mathit{All})\mathbf{pro}.r \to \tau\end{array}}{\rho\ ;\ \Gamma \vdash \langle e_1 \leftarrow m = e_2\rangle : \mathbf{pro}.R}$$

$$(\mathit{field\ over})\qquad \frac{\begin{array}{l}\rho\ ;\ \Gamma \vdash e_1 : \mathbf{probj}.R\\ \rho \vdash R <:_w \langle\!\langle f : \tau\rangle\!\rangle\\ \rho\ ;\ \Gamma \vdash e_2 : \tau\end{array}}{\rho\ ;\ \Gamma \vdash \langle e_1.f := e_2\rangle : \mathbf{probj}.R}$$

$$(\Leftarrow\ \mathit{select})\qquad \frac{\begin{array}{c}\rho\ ;\ \Gamma \vdash e : \mathbf{probj}.R\\ \rho \vdash R <:_w \langle\!\langle m : \tau\rangle\!\rangle\end{array}}{\rho\ ;\ \Gamma \vdash e \Leftarrow m : \tau}$$

$$(\mathit{field\ select})\qquad \frac{\begin{array}{c}\rho\ ;\ \Gamma \vdash e : \mathbf{probj}.R\\ \rho \vdash R <:_w \langle\!\langle f : \tau\rangle\!\rangle\end{array}}{\rho\ ;\ \Gamma \vdash e.f : \tau}$$

$$(\forall\ \mathit{intro})\qquad \frac{\begin{array}{c}\rho, (r <:_w R :: L)\ ;\ \Gamma \vdash e : \tau\\ r \notin FV(\Gamma)\end{array}}{\rho\ ;\ \Gamma \vdash \Lambda().e : \forall(r <:_w R :: L)\tau}$$

$$(\forall\ \mathit{elim})\qquad \frac{\begin{array}{c}\rho\ ;\ \Gamma \vdash e : \forall(r <:_w R :: L)\tau\\ \rho \vdash R' <:_w R\\ \rho \vdash R' :: L\end{array}}{\rho\ ;\ \Gamma \vdash e * () : [R'/r]\tau}$$

$$(\exists\ \mathit{intro})\qquad \frac{\begin{array}{c}\rho \vdash R' <:_w R\\ \rho \vdash R' :: L\\ \rho\ ;\ \Gamma \vdash e : [R'/r]\tau\end{array}}{\rho\ ;\ \Gamma \vdash \{\!|(r <:_w R :: L) = R',\ e|\!\} : \exists(r <:_w R :: L)\tau}$$

$$(\exists\ elim)\quad \frac{\rho\ ;\Gamma \vdash e_1 : \exists(r <:_w R :: L)\tau \qquad \rho\ , (r <:_w R :: L)\ ; \Gamma,\ x : \tau \vdash e_2 : \tau_2 \qquad \rho \vdash \tau_2 : T}{\rho\ ; \Gamma \vdash \textbf{Abstype}\ (r <:_w R :: L)\ \textbf{with}\ x : \tau\ \textbf{is}\ e_1\ \textbf{in}\ e_2 : \tau_2}$$

B Operational Semantics

Addresses		
	$a\ ::=$	an integer, selected from one of four disjoint sets: S_{OBJ}, S_{RCL}, S_{CL}, or S_{EXIS}
Storable Values		
	$sv\ ::=$	$\eta \mid c \mid rc \mid a$
Object Value		
	$\eta\ ::=$	$[l_i\ =\ a_i]^{i \in I}$
Closure Value		
	$c\ ::=$	$\langle \lambda x.\, e,\, E \rangle$
Row Closure Value		
	$rc\ ::=$	$\langle \Lambda().\, e,\, E \rangle$
Environment		
	$E\ ::=$	$x_i \mapsto a_i,\ i \in 1, \ldots, n$
Store		
	$S\ ::=$	$a_i \mapsto sv_i,\ i \in 1, \ldots, n$
Contexts		
	$\{\!\!\{S;\, E\}\!\!\}\ ::=$	store/environment pair.

B.1 Semantic Judgments

$S \models *$	well-formed store
$\{\!\!\{S;\, E\}\!\!\} \models *$	well-formed store/env pair.
$\{\!\!\{S;\, E\}\!\!\} \models e \longrightarrow a \bullet S'$	expression e evaluates to address a and produces store S'.

B.2 Context Judgments

$$(empty\ store)\quad \frac{}{\emptyset \models *}$$

$$(insert\ closure)\quad \frac{\{\!\!\{S;\, E\}\!\!\} \models * \qquad a \notin dom(S)}{S,\, a \mapsto \langle \lambda x.\, e,\, E \rangle \models *}$$

$$(insert\ \Lambda{-}closure)\quad \frac{\{\!\!\{S;\, E\}\!\!\} \models * \qquad a \notin dom(S)}{S,\, a \mapsto \langle \Lambda().\, e,\, E \rangle \models *}$$

$$(insert\ object)\quad \frac{S \models * \qquad a \notin dom(S) \qquad a_i \in dom(S) \text{ for all } i \in I}{S,\, a \mapsto [l_i = a_i]^{i \in I} \models *}$$

$$(insert\ exist)\quad \frac{S \models * \qquad a \notin dom(S) \qquad a' \in dom(S)}{S,\, a \mapsto a' \models *}$$

$$(empty\ environment)\quad \frac{S \models *}{\{\!\!\{S;\, \emptyset\}\!\!\} \models *}$$

$$(insert\ var)\quad \frac{\{\!\!\{S;\, E\}\!\!\} \models * \qquad x \notin dom(E) \qquad a \in dom(S)}{\{\!\!\{S;\, E, x \mapsto a\}\!\!\} \models *}$$

B.3 Reduction Rules

$$(var) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models * \\ E(x) = a\end{array}}{\{\!|S;\, E|\!\} \models x \longrightarrow a \bullet S}$$

$$(lambda) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models * \\ a \notin dom(S)\end{array}}{\{\!|S;\, E|\!\} \models \lambda x.\, e \longrightarrow a \bullet (S, a \mapsto \langle \lambda x.\, e,\, E\rangle)}$$

$$(app) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S_1;\, E|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \langle \lambda x.\, e,\, E'\rangle \qquad x \notin dom(E') \\ \{\!|S_2;\, E',\, x \mapsto a_2|\!\} \models e \longrightarrow a_3 \bullet S_3\end{array}}{\{\!|S;\, E|\!\} \models e_1\, e_2 \longrightarrow a_3 \bullet S_3}$$

$$(empty\ object) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models * \\ a \notin dom(S)\end{array}}{\{\!|S;\, E|\!\} \models \langle\rangle \longrightarrow a \bullet (S, a \mapsto [])}$$

In the following rule, we use symbol $\leftarrow\!\!\!\circ$ as a meta-notation standing for any of $\longleftarrow\!+$, $\longleftarrow$, or $\longleftarrow\!+\!:$. The notation $\eta[l = a]$ denotes the finite map just like η except that $\eta[l = a]$ maps l to a.

$$(\leftarrow\!\!\!\circ) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S_1;\, E|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \eta \qquad a_3 \notin dom(S_2)\end{array}}{\{\!|S;\, E|\!\} \models \langle e_1 \leftarrow\!\!\!\circ\, l = e_2\rangle \longrightarrow a_3 \bullet (S_2, a_3 \mapsto \eta[l = a_2])}$$

In the following rule, the notation $S_2[a_1 \mapsto \eta[f = a_2]]$ denotes the finite map just like S_2 except that a_1 is mapped now to $\eta[f = a_2]$.

$$(:=) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S_1;\, E|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \eta\end{array}}{\{\!|S;\, E|\!\} \models \langle e_1.f := e_2\rangle \longrightarrow a_1 \bullet S_2[a_1 \mapsto \eta[f = a_2]]}$$

$$(\Leftarrow) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models e \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = \eta \qquad\qquad \eta(m) = a_1' \\ S_1(a_1') = \langle \Lambda().\, e_1,\, E_1\rangle \\ \{\!|S_1;\, E_1|\!\} \models e_1 \longrightarrow a_2 \bullet S_2 \\ S_2(a_2) = \langle \lambda x.\, e_2,\, E_2\rangle \qquad x \notin dom(E_2) \\ \{\!|S_2;\, E_2, x \mapsto a_1|\!\} \models e_2 \longrightarrow a_3 \bullet S_3\end{array}}{\{\!|S;\, E|\!\} \models e \Leftarrow m \longrightarrow a_3 \bullet S_3}$$

$$(field\ selection) \qquad \frac{\begin{array}{c}\{\!|S;\, E|\!\} \models e \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = \eta \qquad \eta(f) = a_2\end{array}}{\{\!|S;\, E|\!\} \models e.f \longrightarrow a_2 \bullet S_1}$$

$$
(let)\qquad \frac{\{\!\!\{S;\, E\}\!\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \quad \{\!\!\{S;\, E,\, x \mapsto a_1\}\!\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \quad x \notin dom(E)}{\{\!\!\{S;\, E\}\!\!\} \models \textbf{let } x \ = \ e_1 \ \textbf{in } e_2 \longrightarrow a_2 \bullet S_2}
$$

$$
(\forall\ intro)\qquad \frac{\{\!\!\{S;\, E\}\!\!\} \models * \quad a \notin dom(S)}{\{\!\!\{S;\, E\}\!\!\} \models \Lambda().\, e \longrightarrow a \bullet (S, a \mapsto \langle \Lambda().\, e,\, E \rangle)}
$$

$$
(\forall\ elim)\qquad \frac{\{\!\!\{S;\, E\}\!\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \quad S_1(a_1) = \langle \Lambda().\, e,\, E' \rangle \quad \{\!\!\{S_1;\, E'\}\!\!\} \models e \longrightarrow a_2 \bullet S_2}{\{\!\!\{S;\, E\}\!\!\} \models e_1 * () \longrightarrow a_2 \bullet S_2}
$$

$$
(\exists\ intro)\qquad \frac{\{\!\!\{S;\, E\}\!\!\} \models e \longrightarrow a_1 \bullet S_1 \quad a_2 \notin dom(S_1)}{\{\!\!\{S;\, E\}\!\!\} \models \{\!\!\{(r <:_w R :: L) = R',\ e\}\!\!\} \longrightarrow a_2 \bullet (S_1, a_2 \mapsto a_1)}
$$

$$
(\exists\ elim)\qquad \frac{\{\!\!\{S;\, E\}\!\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \quad S_1(a_1) = a_1' \qquad x \notin dom(E) \quad \{\!\!\{S_1;\, (E, x \mapsto a_1')\}\!\!\} \models e_2 \longrightarrow a_2 \bullet S_2}{\{\!\!\{S;\, E\}\!\!\} \models \textbf{Abstype } (r <:_w R :: L) \textbf{ with } x : \tau \textbf{ is } e_1 \textbf{ in } e_2 \longrightarrow a_2 \bullet S_2}
$$

C Semantic Typing

Store Types

$$\Sigma \quad ::= \quad a_i \mapsto \tau_i,\ i \in 1,\ldots,n \quad \tau_i \text{ closed.}$$

Furthermore, if τ_i is of the form $\exists(r <:_w R :: L)\tau$, then there is a subsidiary map $\Sigma_\exists$:

$$\Sigma_\exists \quad ::= \quad a_i \mapsto R_i \quad R_i \text{ closed.}$$

Intuitively, R_i is the hidden implementation row associated with existential type $\exists(r <:_w R :: L)\tau$.

Row Environment

$$E_\rho \quad ::= \quad r_i \mapsto R_i,\ i \in 1,\ldots,m \quad R_i \text{ closed}$$

C.1 Semantic Typing Judgments

$\Sigma \models *$	if $\forall a \in dom(\Sigma)$, $\epsilon \vdash \Sigma(a) : T$, and if $\Sigma(a) = \exists(r <:_w R :: L)\tau$, then $\epsilon \vdash \Sigma_\exists(a) <:_w R$ and $\epsilon \vdash \Sigma_\exists(a) :: L$.
$\Sigma' \succeq \Sigma$	if $dom(\Sigma) \subseteq dom(\Sigma')$ and $\forall a \in dom(\Sigma)$, $\Sigma(a) = \Sigma'(a)$, and $\forall a \in dom(\Sigma_\exists)$, $\Sigma_\exists(a) = \Sigma'_\exists(a)$.
$\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma$	Row environment E_ρ and value environment E consistent with row context ρ and expression context Γ.
$\Sigma \models_{ob} S$	object portion of S well-typed wrt store type Σ
$\Sigma \models_{cl} S$	closure portion of S well-typed wrt store type Σ.
$\Sigma \models_{rcl} S$	row closure portion of S well-typed wrt store type Σ.
$\Sigma \models_\exists S$	existential portion of S well-typed wrt store type Σ.
$\Sigma \models S$	S well-typed wrt store type Σ.

C.2 Row Environment Substitution

$$[E_\rho \rightarrow]* \stackrel{def}{=} *$$

$$[E_\rho \rightarrow]\tau \stackrel{def}{=} [R_i/r_i]^{i\in 1\ldots n}\tau \qquad \text{for } E_\rho \equiv r_i \mapsto R_i, i \in 1\ldots n$$

$$[E_\rho \rightarrow]R \stackrel{def}{=} [R_i/r_i]^{i\in 1\ldots n}R \qquad \text{for } E_\rho \equiv r_i \mapsto R_i, i \in 1\ldots n$$

$$[E_\rho \rightarrow]e \stackrel{def}{=} [R_i/r_i]^{i\in 1\ldots n}e \qquad \text{for } E_\rho \equiv r_i \mapsto R_i, i \in 1\ldots n$$

$$[E_\rho \rightarrow](\tau_1 <: \tau_2) \stackrel{def}{=} [E_\rho \rightarrow]\tau_1 <: [E_\rho \rightarrow]\tau_2$$

$$[E_\rho \rightarrow](R_1 <:_B R_2) \stackrel{def}{=} [E_\rho \rightarrow]R_1 <:_B [E_\rho \rightarrow]R_2$$

$$[E_\rho \rightarrow](e : \tau) \stackrel{def}{=} [E_\rho \rightarrow]e : [E_\rho \rightarrow]\tau$$

$[E_\rho \rightarrow]\Gamma$ is defined by:

$$[E_\rho \rightarrow]\epsilon \stackrel{def}{=} \epsilon$$

$$[E_\rho \rightarrow](\Gamma,\ x : \tau) \stackrel{def}{=} [E_\rho \rightarrow]\Gamma,\ x : [E_\rho \rightarrow]\tau$$

$[E_\rho \rightarrow]\rho$ is defined by:

$$[E_\rho \rightarrow]\epsilon \stackrel{def}{=} \epsilon$$

$$[E_\rho \rightarrow](\rho,\ (r <:_w R :: L)) \stackrel{def}{=} [E_\rho \rightarrow]\rho \qquad \text{if } r \in dom(E_\rho)$$

$$[E_\rho \rightarrow](\rho,\ (r <:_w R :: L)) \stackrel{def}{=} [E_\rho \rightarrow]\rho,\ (r <:_w [E_\rho \rightarrow]R :: L) \qquad \text{if } r \notin dom(E_\rho)$$

$[E_\rho \rightarrow](\rho\,;\Gamma) \stackrel{def}{=} [E_\rho \rightarrow]\rho\,; [E_\rho \rightarrow]\Gamma$

C.3 Semantic Consistency Judgments

$$(Empty)\quad \frac{\Sigma \models *}{\Sigma \models \emptyset \bullet \emptyset : \epsilon \bullet \epsilon}$$

$$(Exp\ Var)\quad \frac{\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma \qquad \epsilon \vdash \Sigma(a) <: [E_\rho \to]\tau \qquad x \notin dom(\Gamma)}{\Sigma \models E_\rho \bullet (E, x \mapsto a) : \rho \bullet (\Gamma,\ x : \tau)}$$

$$(Row\ Var)\quad \frac{\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma \qquad \epsilon \vdash R_1 <:_w [E_\rho \to]R_2 \qquad \epsilon \vdash R_1 :: L \qquad r \notin dom(\rho)}{\Sigma \models (E_\rho,\ r \mapsto R_1) \bullet E : (\rho,\ (r <:_w R_2 :: L)) \bullet \Gamma}$$

C.4 Store Typing Judgments

$$(\models_{ob})\quad \frac{\begin{array}{c}\Sigma \models * \\ \Sigma(a_j) = \mathbf{pro.}\langle\!\langle m_i : \tau_i,\ f_k : \tau'_k \rangle\!\rangle^{i\in I,\, k\in K} \\ j \in 1..|S_{OBJ}| \\ \Sigma(a_i) = \forall(r_i <:_w R_i :: All)\mathbf{pro.}r_i \to \tau_i \qquad i \in I \\ \epsilon \vdash \Sigma(a_k) <: \tau'_k \qquad k \in K \\ \epsilon \vdash \langle\!\langle m_i : \tau_i,\ f_k : \tau'_k \rangle\!\rangle^{i\in I,\, k\in K} <:_w R_i\end{array}}{\begin{array}{c}\Sigma \models_{ob} a_j \mapsto [m_i = a_i,\ f_k = a_k]^{i\in I,\, k\in K} \\ j \in 1..|S_{OBJ}|\end{array}}$$

$$(\models_{cl})\quad \frac{\begin{array}{c}\Sigma \models * \\ \Sigma(a_j) = \tau_j \to \tau'_j \qquad j \in 1..|S_{CL}| \\ \Sigma \models \emptyset \bullet E_j : \epsilon \bullet \Gamma_j \\ \epsilon\,;\Gamma_j,\, x_j : \tau_j \vdash e_j : \tau'_j\end{array}}{\begin{array}{c}\Sigma \models_{cl} a_j \mapsto \langle \lambda x_j \,.\, e_j,\ E_j\rangle \\ j \in 1..|S_{CL}|\end{array}}$$

$$(\models_{rcl})\quad \frac{\begin{array}{c}\Sigma \models * \\ \Sigma(a_j) = \forall(r_j <:_w R_j :: L_j)\tau_j \qquad j \in 1..|S_{RCL}| \\ \Sigma \models \emptyset \bullet E_j : \epsilon \bullet \Gamma_j \\ (r_j <:_w R_j :: L_j)\,;\Gamma_j \vdash e_j : \tau_j\end{array}}{\begin{array}{c}\Sigma \models_{rcl} a_j \mapsto \langle \Lambda().\, e_j,\ E_j\rangle \\ j \in 1..|S_{RCL}|\end{array}}$$

$$(\models_{\exists})\quad \frac{\begin{array}{c}\Sigma \models * \\ \Sigma(a_j) = \exists(r_j <:_w R_j :: L_j)\tau_j \qquad j \in 1..|S_{EXIS}| \\ \Sigma_\exists(a_j) = R'_j \\ \epsilon \vdash \Sigma(a'_j) <: [R'_j/r_j]\tau_j\end{array}}{\begin{array}{c}\Sigma \models_\exists a_j \mapsto a'_j \\ j \in 1..|S_{EXIS}|\end{array}}$$

$$(\models)\quad \frac{\Sigma \models_{ob} S \qquad \Sigma \models_{cl} S \qquad \Sigma \models_{rcl} S \qquad \Sigma \models_\exists S}{\Sigma \models S}$$

D Meta-theory

In this appendix, we briefly list some of the key lemmas needed to prove subject reduction.

Lemma D.1 (Bound Transformation) *If the judgments* $\rho, (r <:_w R{::}L), \rho'; \Gamma \vdash A$, $\rho \vdash R_1 <:_w R$, *and* $\rho \vdash R_1{::}L_1$ *are all derivable, where* $L_1 \subseteq L$, *then the judgment* $\rho, (r <:_w R_1 :: L), \rho'; \Gamma \vdash A$ *is derivable as well.*

Lemma D.2 (Row Substitution) *If the judgments* $\rho, (r <:_w R::L), \rho'; \Gamma \vdash A$, $\rho \vdash R_1 <:_w R$, *and* $\rho \vdash R_1 :: L_1$ *are all derivable, then so is the judgment* $\rho, [R_1/r]\rho'; [R_1/r]\Gamma \vdash [R_1/r]A$.

Lemma D.3 (Strengthening) *If the judgments* $\rho; \Gamma, x : \tau, \Gamma' \vdash A$ *and* $\rho \vdash \tau' <: \tau$ *are derivable, then so is* $\rho; \Gamma, x : \tau', \Gamma' \vdash A$

Lemma D.4 (Properties of Consistency Judgments) *If* $\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma$ *is derivable, then*

1. $r \in dom(E_\rho)$ *iff* $r \in dom(\rho)$,
2. $\Sigma \models \emptyset \bullet E : \epsilon \bullet [E_\rho \rightarrow]\Gamma$ *is derivable as well,*
3. *if* $\rho \vdash A$ *is derivable, then so is* $[E_\rho \rightarrow](\rho) \vdash [E_\rho \rightarrow]A$, *and*
4. *if* $\rho \,;\, \Gamma \vdash A$ *is derivable, then so is* $[E_\rho \rightarrow](\rho) \,;\, [E_\rho \rightarrow](\Gamma) \vdash [E_\rho \rightarrow]A$.

Lemma D.5 (Store Type Weakening)

$$\begin{array}{ll} \textit{If} & \Sigma \models E_\rho \bullet E : \rho \bullet \Gamma, \\ & \Sigma' \models *, \ \textit{and} \\ & \Sigma' \succeq \Sigma \ \textit{all hold, then} \\ & \Sigma' \models E_\rho \bullet E : \rho \bullet \Gamma \ \textit{holds as well.} \end{array}$$

Theorem D.6 (Subject Reduction) *If*

$$\begin{array}{l} \rho \,;\, \Gamma \vdash e : \tau, \\ \{\!|S;\, E|\!\} \models e \longrightarrow a \bullet S', \\ \Sigma \models S, \\ dom(\Sigma) = dom(S), \ \textit{and} \\ \Sigma \models E_\rho \bullet E : \rho \bullet \Gamma, \end{array}$$

there exists a closed type τ' *and a store type* Σ' *such that*

$$\begin{array}{l} \Sigma' \succeq \Sigma, \\ \Sigma' \models S', \\ dom(\Sigma') = dom(S'), \\ \Sigma'(a) = \tau', \ \textit{and} \\ \epsilon \vdash \tau' <: [E_\rho \rightarrow]\tau. \end{array}$$

Proof The proof is by induction on the derivation of $\{\!|S;\, E|\!\} \models e \longrightarrow a \bullet S'$.