An Imperative Object Calculus

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract. We develop an imperative calculus of objects. Its main type constructor is the one for object types, which incorporate variance annotations and Self types. A subtyping relation between object types supports object subsumption. The type system for objects relies on unusual but beneficial assumptions about the possible subtypes of an object type. With the addition of polymorphism, the calculus can express classes and inheritance.

1 Introduction

Object calculi are formalisms at the same level of abstraction as λ-calculi, but based exclusively on objects rather than functions. Unlike λ-calculi, object calculi are designed specifically for clarifying features of object-oriented languages. There is a wide spectrum of relevant object calculi, just as there is a wide spectrum of λ-calculi. One can investigate untyped, simply-typed, and polymorphic calculi, as well as functional and imperative calculi.

In object calculi, as in λ-calculi, a small untyped kernel is enriched with derived constructions and with increasingly sophisticated type systems, until language features can be realistically modeled. The compactness of the initial kernel gives conceptual unity to the calculi, and enables formal analysis.

In this paper we introduce a tiny but expressive imperative calculus. It provides a minimal setting in which to study the imperative operational semantics and the delicate typing rules of practical object-oriented languages. The calculus comprises objects, method invocation, method update, object cloning, and local definitions. In a quest for minimality, we take objects to be just collections of methods. Fields are important too, but they can be seen as a derived concept; for example a field can be viewed as a method that does not use its self parameter. In our calculus, methods are themselves mutable, so we can dispense with fields.

The main type constructor of our calculus is the one for object types; an object type is a list of method names and method result types. A subtyping relation between object types supports object subsumption, which allows an object to be used where an object with fewer methods is expected. Variance annotations enable flexible subtyping and protection from side effects.

The object type constructor incorporates a notion of Self types. Intuitively, Self is the partially unknown type of the self parameter of each method. Several object-oriented languages have included it in their type systems [23, 27], sometimes with unsound rules [16]. Therefore, it seems important to have a precise understanding of Self. Unfortunately, it has proven hard to reduce Self to more primitive and well-understood

notions (see [3, 7, 24] for recent progress). We aim to provide a satisfactory treatment of Self by taking it as primitive and axiomatizing its desired properties.

The treatment of Self types relies on assumptions about the possible subtypes of object types. These assumptions are operationally sound, but would not hold in natural semantic models. We show the necessity of these assumptions in finding satisfactory typings for programs involving Self types.

We consider also bounded type quantifiers for polymorphism. Taken together, objects with imperative features, object types, and polymorphism form a realistic kernel for a programming language. Using these primitives, we account for classes, subclasses, inheritance, method and field specialization in subclasses, parametric method update, and protection from external updates.

We prove the consistency of our rules using a subject-reduction approach. Our technique is an extension of Harper's [19], using closures and stacks instead of formal substitutions. This approach yields a manageable proof for a realistic implementation strategy.

A few other object formalisms have been defined and studied. Many of these rely on purely functional models, with an emphasis on types [1, 7, 11, 13, 20, 24, 25, 26, 32]. Others deal with imperative features in the context of concurrency; see for example [34]. The works most closely related to ours are that of Eifrig *et al.* on LOOP [18] and that of Bruce *et al.* on PolyTOIL [9]. LOOP and PolyTOIL are typed, imperative, object-oriented languages with procedures, objects, and classes. PolyTOIL takes procedures, objects, and classes as primitive, with fairly complex rules. LOOP is translated into a somewhat simpler calculus. Our calculus is centered on objects; procedures and classes can be defined from them. Despite these differences, we all share the goal of modeling imperative object-oriented languages by precise semantic structures and sound type systems.

This paper is self-contained, but continues our work of [2, 3, 4, 5]. The most apparent novelties are the imperative features, the variance annotations, the treatment of Self types, and the representation of classes and inheritance. The new typing features led us to prefer syntactic proof techniques over denotational methods.

In section 2 we give the untyped term syntax of our calculus and its operational semantics. In section 3 we present object types. In section 4 we add bounded universal quantifiers. In section 5 we discuss soundness. In section 6 we consider the typing of some critical examples, and provide a representation of classes and method inheritance.

2 An Untyped Imperative Calculus

We begin with the syntax of an untyped imperative calculus. The initial syntax is minimal, but in sections 2.2, 2.3, and 6.2 we show how to express convenient constructs such as fields, procedures, and classes. We omit how to encode basic data types and control structures, which can be treated much as in [4]. In section 2.5 we give an operational semantics.

2.1 Syntax and Informal Semantics

The evaluation of terms is based on an imperative operational semantics with a store, and generally proceeds deterministically from left to right. The letter ς (sigma) is a binder; it delays evaluation of the term to its right.

Syntax of terms

$a,b ::=$	term
x	variable
$[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$	object (l_i distinct)
$a.l$	method invocation
$a.l\Leftarrow(y,z=c)\varsigma(x)b$	method update
$clone(a)$	cloning

An object is a collection of components $l_i=\varsigma(x_i)b_i$, for distinct labels l_i and associated methods $\varsigma(x_i)b_i$; the order of these components does not matter, even for our deterministic operational semantics. Each binder ς binds the self parameter of a method; $\varsigma(x)b$ is a method with self variable x and body b.

A method invocation $a.l$ results in the evaluation of a, followed by the evaluation of the body of the method named l, with the value of a bound to the self variable of the method.

A cloning operation $clone(a)$ produces a new object with the same labels as a, with each component sharing the methods of the corresponding component of a.

The method update construct, $a.l\Leftarrow(y,z=c)\varsigma(x)b$, is best understood by first looking at the special case $a.l\Leftarrow\varsigma(x)b$. This simple method update construct evaluates a, replaces the method named l with the new method $\varsigma(x)b$, and returns the modified object. The general form of method update adds the ability to evaluate a term at the time of the update, and to use the result of the evaluation later when the updated method is invoked. The construct $a.l\Leftarrow(y,z=c)\varsigma(x)b$ first evaluates a and binds its value to y, then evaluates c and binds its value to z, and finally updates the l method of y and returns the modified object. The variable y may occur in c and b, and the variables z and x may occur in b. After the update, when the method l is invoked, y still points to the value of a, and x points to the current self. In general, the current self and the value of a need not coincide; for example the current self may be a clone of a, as in $clone(a.l\Leftarrow(y,z=c)\varsigma(x)b).l$.

In an untyped calculus, the construct $a.l\Leftarrow(y,z=c)\varsigma(x)b$ can be expressed in terms of *let* and simple method update, as $let\ y = a\ in\ let\ z = c\ in\ y.l\Leftarrow\varsigma(x)b$. However, the construct $a.l\Leftarrow(y,z=c)\varsigma(x)b$ yields better typings, as shown in section 6.1.

Conversely, simple method update, *let*, and sequencing are definable from method update:

$$a.l\Leftarrow\varsigma(x)b \triangleq a.l\Leftarrow(y,z=y)\varsigma(x)b \quad \text{where } y,z \notin FV(b)$$

$$let\ x = a\ in\ b \triangleq ([val = \varsigma(y)y.val].val\Leftarrow(z,x=a)\varsigma(w)b).val \quad \text{where } y,z,w \notin FV(b) \text{ and } z \notin FV(a)$$

$$a\ ;\ b \triangleq let\ x = a\ in\ b \quad \text{where } x \notin FV(b)$$

2.2 Fields

In our calculus, every component of an object contains a method. However, we can encode fields with eagerly evaluated contents. We write $[l_i{=}b_i\ ^{i\in 1..n},\ l_j{=}\varsigma(x_j)b_j\ ^{j\in 1..m}]$ for an object where $l_i{=}b_i$ are fields and $l_j{=}\varsigma(x_j)b_j$ are methods. We also write $a.l{:=}b$ for field update, and $a.l$, as before, for field selection. We abbreviate:

Field notation

$[l_i{=}b_i\ ^{i\in 1..n},\ l_j{=}\varsigma(x_j)b_j\ ^{j\in 1..m}] \triangleq$
$\quad let\ y_1{=}b_1\ in\ ...\ let\ y_n{=}b_n\ in\ [l_i{=}\varsigma(y_0)y_i\ ^{i\in 1..n},\ l_j{=}\varsigma(x_j)b_j\ ^{j\in 1..m}]$
$\quad\quad$ for $y_i \notin FV(b_i\ ^{i\in 1..n},\ b_j\ ^{j\in 1..m})$, with y_i distinct for $i\in 0..n$

$a.l{:=}b \triangleq$
$\quad a.l{\Leftarrow}(y,z{=}b)\varsigma(x)z$
$\quad\quad$ for $y \notin FV(b)$, with x,y,z distinct

The semantics of an object with fields may depend on the order of its components, because of side-effects in computing contents of fields. The encoding specifies an evaluation order.

When fields and methods are identified, as in our calculus, it is trivial to convert one into the other, conceptually turning passive data into active computation and vice versa. Thus, we use somewhat interchangeably the names selection and invocation.

The hiding of fields from public view has been widely advocated as a means of concealing representation choices, and thereby allowing flexibility in implementation. Identifying fields with methods confers much of the same flexibility, by turning all fields into methods that access a hidden representation.

The unification of fields with methods has also the advantage of simplicity. Both objects and object operations assume a uniform structure. In contrast, the separation of fields from methods induces a corresponding separation of object operations, and leads to the implicit or explicit splitting of objects into two components. Unifying fields with methods gives more compact and therefore more elegant calculi.

This unification, however, has one debatable consequence. The natural operation on methods is method invocation, and the natural operations on fields are field selection and field update. By unifying fields with methods, we can collapse field selection and method invocation into a single operation. To complete the unification, though, we are forced to generalize field update to method update.

The reliance on method update is one of the most unusual aspects of our formal treatment. This operation is not normally found in programming languages, with some exceptions (for example: Beta [22], Obliq [12]). Method update can be seen as a form of dynamic inheritance [31], which is a feature found in object-based languages [10] but not yet in class-based languages. Like other forms of dynamic inheritance, method update supports the dynamic modification of object behavior allowing objects, in a sense, to change their class dynamically. Thus, method update gives us an edge in modeling object-based constructions, in addition to allowing us to model the more traditional class-based constructions where fields and methods are sharply separated.

A further justification for method update can be found in the desire to tame dynamic inheritance. Dynamic inheritance has potentially unpredictable effects, due to the updating of shared state. These concerns have led to the search for better-behaved, restricted, dynamic inheritance mechanisms [29]. Method update is one of these better-behaved mechanisms; it is statically typeable, and can be used to emulate the mode-switching applications of dynamic inheritance [15]. With method update we avoid some dangerous aspects of dynamic inheritance [17, 29], while maintaining its dynamic specialization aspects [28].

2.3 Procedures

Our object calculus is so minimal that it does not include procedures, but these can be expressed too. To illustrate this point, we consider informally an imperative call-by-value λ-calculus that includes abstraction, application, and assignment to λ-bound variables. For example, assuming arithmetic primitives, $(\lambda(x)\ x{:=}x{+}1)(3)$ is a term yielding 4. We translate this λ-calculus into our object calculus:

Translation of procedures

$\langle\!\langle x \rangle\!\rangle_\rho$	$\triangleq$	$\rho(x)$ if $x \in dom(\rho)$, and x otherwise
$\langle\!\langle x{:=}a \rangle\!\rangle_\rho$	$\triangleq$	$x.arg{:=}\langle\!\langle a \rangle\!\rangle_\rho$
$\langle\!\langle \lambda(x)b \rangle\!\rangle_\rho$	$\triangleq$	$[arg = \varsigma(x)x.arg,\ val = \varsigma(x)\langle\!\langle b \rangle\!\rangle_{\rho\{x \leftarrow x.arg\}}]$
$\langle\!\langle b(a) \rangle\!\rangle_\rho$	$\triangleq$	$(clone(\langle\!\langle b \rangle\!\rangle_\rho).arg{:=}\langle\!\langle a \rangle\!\rangle_\rho).val$

In the translation, an environment ρ maps each variable x either to $x.arg$ if x is λ-bound, or to x if x is a free variable. A λ-abstraction is translated to an object with an *arg* component, for storing the argument, and a *val* method, for executing the body. The *arg* component is initially set to a divergent method, and is filled with an argument upon procedure call. A call activates the *val* method that can then access the argument through self as $x.arg$. An assignment $x{:=}a$ updates $x.arg$, where the argument is stored (assuming that x is λ-bound). A procedure needs to be cloned when it is called; the clone provides a fresh location in which to store the argument of the call, preventing interference with other calls of the same procedure. Such interference would derail recursive invocations.

2.4 A Small Example

We give a trivial example as a notation drill. We use fields, procedures, and booleans in defining a memory cell with *get*, *set*, and *dup* (duplicate) components:

$$\begin{array}{l} let\ m = [get = false,\ \ set = \varsigma(self)\ \lambda(b)\ self.get{:=}b,\ \ dup = \varsigma(self)\ clone(self)] \\ in\ m.set(true);\ m.get \\ \qquad \text{yields } true \end{array}$$

2.5 Operational Semantics

We now give an operational semantics for the basic calculus of section 2.1; the semantics relates terms to results in a global store. Object terms reduce to object results $[l_i{=}\iota_i\ ^{i\in 1..n}]$ consisting of sequences of store locations, one location for each object component. In order to stay close to standard implementation techniques, we avoid using formal substitutions during reduction: we describe a semantics based on stacks and closures. A stack S associates variables with results; a closure $\langle \varsigma(x)b,S\rangle$ is a pair of a method and a stack that is used for the reduction of the method body. A store σ maps locations ι to method closures; we write stores in the form $\iota_i{\mapsto}\langle \varsigma(x_i)b_i,S_i\rangle\ ^{i\in 1..n}$. We let $\sigma.\iota{\leftarrow\!\!\shortmid} m$ denote the result of writing m in the ι location of σ.

The operational semantics is expressed in terms of a relation that relates a store σ, a stack S, a term b, a result v, and another store σ'. This relation is written $\sigma{\bullet}S \vdash b \rightsquigarrow v{\bullet}\sigma'$, and it means that with the store σ and the stack S, the term b reduces to a result v, yielding an updated store σ'; the stack does not change.

Notation

ι			store location (e.g., an integer)
v	::=	$[l_i{=}\iota_i\ ^{i\in 1..n}]$	object result(l_i distinct)
σ	::=	$\iota_i{\mapsto}\langle \varsigma(x_i)b_i,S_i\rangle\ ^{i\in 1..n}$	store(ι_i distinct)
S	::=	$x_i{\mapsto}v_i\ ^{i\in 1..n}$	stack (x_i distinct)

Well-formed store judgment: $\sigma \vdash \diamond$

(Store $\emptyset$)

$$\frac{}{\emptyset \vdash \diamond}$$

(Store ι)

$$\frac{\sigma{\bullet}S \vdash \diamond \qquad \iota \notin dom(\sigma)}{\sigma,\ \iota{\mapsto}\langle \varsigma(x)b,S\rangle \vdash \diamond}$$

Well-formed stack judgment: $\sigma{\bullet}S \vdash \diamond$

(Stack $\emptyset$)

$$\frac{\sigma \vdash \diamond}{\sigma{\bullet}\emptyset \vdash \diamond}$$

(Stack x) (l_i,ι_i distinct)

$$\frac{\sigma{\bullet}S \vdash \diamond \qquad \iota_i \in dom(\sigma) \qquad x \notin dom(S) \qquad \forall i \in 1..n}{\sigma \bullet S,\ x{\mapsto}[l_i{=}\iota_i\ ^{i\in 1..n}] \vdash \diamond}$$

Term reduction judgment: $\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$

(Red x)

$$\frac{\sigma \bullet S', x \mapsto v, S'' \vdash \diamond}{\sigma \bullet S', x \mapsto v, S'' \vdash x \rightsquigarrow v \bullet \sigma}$$

(Red Object) (l_i, ι_i distinct)

$$\frac{\sigma \bullet S \vdash \diamond \qquad \iota_i \notin dom(\sigma) \qquad \forall i \in 1..n}{\sigma \bullet S \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet (\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, S \rangle \ ^{i \in 1..n})}$$

(Red Select)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet \sigma' \qquad \sigma'(\iota_j) = \langle \varsigma(x_j) b_j, S' \rangle \qquad x_j \notin dom(S') \qquad j \in 1..n \qquad \sigma' \bullet S', x_j \mapsto [l_i = \iota_i \ ^{i \in 1..n}] \vdash b_j \rightsquigarrow v \bullet \sigma''}{\sigma \bullet S \vdash a.l_j \rightsquigarrow v \bullet \sigma''}$$

(Red Update)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet \sigma' \qquad \iota_j \in dom(\sigma') \qquad j \in 1..n \qquad \sigma' \bullet S, y \mapsto [l_i = \iota_i \ ^{i \in 1..n}] \vdash c \rightsquigarrow v \bullet \sigma''}{\sigma \bullet S \vdash a.l_j \Leftarrow (y, z = c) \varsigma(x) b \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet \sigma''.\iota_j \leftarrow \langle \varsigma(x) b, (S, y \mapsto [l_i = \iota_i \ ^{i \in 1..n}], z \mapsto v) \rangle}$$

(Red Clone) (ι_i' distinct)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet \sigma' \qquad \iota_i \in dom(\sigma') \qquad \iota_i' \notin dom(\sigma') \qquad \forall i \in 1..n}{\sigma \bullet S \vdash clone(a) \rightsquigarrow [l_i = \iota_i' \ ^{i \in 1..n}] \bullet (\sigma', \iota_i' \mapsto \sigma'(\iota_i) \ ^{i \in 1..n})}$$

A variable reduces to the result it denotes in the current stack. An object reduces to a result consisting of a fresh collection of locations; the store is extended to associate method closures to those locations. A selection operation $a.l_j$ first reduces the object a to a result, then activates the appropriate method closure. An update operation $a.l_j \Leftarrow (y, z = c) \varsigma(x) b$ first reduces the object a to the final result; next, with y bound to that result, it reduces the term c to another result; finally, it updates the appropriate store location with a closure consisting of the new method $\varsigma(x)b$ and a stack binding y and z. A *clone* operation reduces its object to a result; then it allocates a fresh collection of locations that are associated to the existing method closures from the object.

We illustrate method update, and the creation of loops through the store, by the reduction of the term $[l = \varsigma(x) x.\mathrm{l} := x].l$, that is, $[l = \varsigma(x) x.l \Leftarrow (y, z = x) \varsigma(w) z].l$:

$$\emptyset \bullet \emptyset \vdash [l = \varsigma(x) x.l \Leftarrow (y, z = x) \varsigma(w) z].l \rightsquigarrow [l = 0] \bullet \sigma$$
$$\text{where} \quad \sigma \equiv 0 \mapsto \langle \varsigma(w) z, (x \mapsto [l = 0], y \mapsto [l = 0], z \mapsto [l = 0]) \rangle$$

The store σ contains a loop, because it maps the index 0 to a closure that binds the variable z to a value that contains index 0. Hence, an attempt to read out the result of $[l = \varsigma(x) x.l := x].l$ by "inlining" the store and stack mappings would produce the infinite term $[l = \varsigma(w)[l = \varsigma(w)[l = \varsigma(w) ...]]]$.

3 Typing

In this section we develop a type system for the calculus of section 2. We have the following syntax of types:

Syntax of types

A,B ::=	type
X	type variable
Top	the biggest type
$Obj(X)[l_i \upsilon_i : B_i\{X\}^{\ i\in 1..n}]$	object type ($\upsilon_i \in \{^-, ^o, ^+\}$, l_i distinct)

Let $A \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\ i\in 1..n}]$. The notation $B_i\{X\}$ indicates that X may occur free in B_i. The binder Obj binds a Self type named X, which is known to be a subtype of A. Then A is the type of those objects with methods named $l_i^{\ i\in 1..n}$ having self parameters of type X, and with corresponding result types $B_i\{X\}$. The type X may occur only covariantly in the result types B_i. This covariance requirement is necessary for the soundness of our rules; covariance is defined precisely below.

Each υ_i is a variance annotation; it is one of the symbols $^-$, ^{o}, and $^+$, for contravariance, invariance, and covariance, respectively. Covariant components allow covariant subtyping, but prevent update. Symmetrically, contravariant components allow contravariant subtyping, but prevent invocation. Invariant components can be both invoked and updated; by subtyping, they can be regarded as either covariant or contravariant. Therefore, variance annotations support flexible subtyping and a form of protection.

The rules for our object calculus are given next. The first three groups of rules concern typing environments, types, and the subtyping relation. The final group concerns typing of terms: there is one rule for each construct in the calculus; in addition, a subsumption rule connects term typing with subtyping. In the rules, a premise of the form "$E,E_i \vdash \Im_i \ \ \forall i \in 1..n$" is an abbreviation for n premises "$E,E_1 \vdash \Im_1$ ... $E,E_n \vdash \Im_n$" if $n>0$, and if $n=0$ for "$E \vdash \diamond$", which means that E is well-formed. Instead, "$j \in 1..n$" in the premise indicates that there are n separate rules, one for each j.

Well-formed environment judgment: $E \vdash \diamond$

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)
$$\frac{E \vdash A \qquad x \notin dom(E)}{E,x{:}A \vdash \diamond}$$

(Env $X<:$)
$$\frac{E \vdash A \qquad X \notin dom(E)}{E,X<:A \vdash \diamond}$$

Well-formed type judgment: $E \vdash A$

(Type $X<:$)
$$\frac{E',X<:A,E'' \vdash \diamond}{E',X<:A,E'' \vdash X}$$

(Type Top)
$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Type Object) (l_i distinct, $\upsilon_i \in \{^o, ^-, ^+\}$)
$$\frac{E,X<:Top \vdash B_i\{X^+\} \qquad \forall i \in 1..n}{E \vdash Obj(X)[l_i \upsilon_i : B_i\{X\}^{\ i\in 1..n}]}$$

Formally, $B\{X^+\}$ indicates that X occurs only covariantly in B; that is, either B is a

variable (possibly X), or $B \equiv Top$, or $B \equiv Obj(Y)[l_i \upsilon_i : B_i \, {}^{i \in 1..n}]$ and either $Y \equiv X$ or for each $\upsilon_i \equiv {}^{+}$ we have $B_i\{X^{+}\}$, for each $\upsilon_i \equiv {}^{-}$ we have $B_i\{X^{-}\}$, and for each $\upsilon_i \equiv {}^{o}$ we have $X \notin FV(B_i)$. Similarly, $B\{X^{-}\}$ indicates that X occurs only contravariantly in B; that is, either B is a variable different from X, or $B \equiv Top$, or $B \equiv Obj(Y)[l_i \upsilon_i : B_i \, {}^{i \in 1..n}]$ and either $Y \equiv X$ or for each $\upsilon_i \equiv {}^{+}$ we have $B_i\{X^{-}\}$, for each $\upsilon_i \equiv {}^{-}$ we have $B_i\{X^{+}\}$, and for each $\upsilon_i \equiv {}^{o}$ we have $X \notin FV(B_i)$.

The formation rule for object types (Type Object) requires that all the component types be covariant in Self. According to the definition of covariant occurrences, more than one Self type may be active in a given context, as in the type $Obj(X)[l^{o}: Obj(Y)[m^{+}:X, n^{o}:Y]]$.

By convention, any omitted υ's are taken to be equal to ^{o}. We regard a simple object type $[l_i:B_i \, {}^{i \in 1..n}]$ (as defined in [4]) as an abbreviation for $Obj(X)[l_i{}^{o}:B_i \, {}^{i \in 1..n}]$, where X does not appear in any B_i.

We denote by $B\{\!|A|\!\}$ the result of substituting A for X in $B\{X\}$, where X is clear from context.

Subtyping judgments: $E \vdash A <: B$, $E \vdash \upsilon \, A <: \upsilon' \, B$

(Sub Refl)
$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)
$$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Sub Top)
$$\frac{E \vdash A}{E \vdash A <: Top}$$

(Sub X)
$$\frac{E', X <: A, E'' \vdash \diamond}{E', X <: A, E'' \vdash X <: A}$$

(Sub Object) (where $A \equiv Obj(X)[l_i \upsilon_i : B_i\{X\} \, {}^{i \in 1..n+m}]$, $A' \equiv Obj(X)[l_i \upsilon_i' : B_i'\{X\} \, {}^{i \in 1..n}]$)
$$\frac{E \vdash A \qquad E \vdash A' \qquad E, Y <: A \vdash \upsilon_i \, B_i\{\!|Y|\!\} <: \upsilon_i' \, B_i'\{\!|Y|\!\} \qquad \forall i \in 1..n}{E \vdash A <: A'}$$

(Sub Invariant)
$$\frac{E \vdash B}{E \vdash {}^{o}B <: {}^{o}B}$$

(Sub Covariant)
$$\frac{E \vdash B <: B' \qquad \upsilon \in \{{}^{o}, {}^{+}\}}{E \vdash \upsilon \, B <: {}^{+}B'}$$

(Sub Contravariant)
$$\frac{E \vdash B' <: B \qquad \upsilon \in \{{}^{o}, {}^{-}\}}{E \vdash \upsilon \, B <: {}^{-}B'}$$

The subtyping rule for object types (Sub Object) says, to a first approximation, that a longer object type on the left is a subtype of a shorter one on the right. The antecedents operate under the assumption that Self is a subtype of the longer type.

Because of the variance annotations we use an auxiliary judgment, $E \vdash \upsilon \, B <: \upsilon' \, B'$, for inclusion of components with variance. The rules say:

- (Sub Object) Components that occur both on the left and on the right are handled by the other three rules. For components that occur only on the left, the component types must be well-formed.
- (Sub Invariant) An invariant component on the right requires an identical one on the left.
- (Sub Covariant) A covariant component type on the right can be a supertype of a corresponding component type on the left, either covariant or invariant. Intuitively, an invariant component can be regarded as covariant.
- (Sub Contravariant) A contravariant component type on the right can be a sub-

type of a corresponding component type on the left, either contravariant or invariant. Intuitively, an invariant component can be regarded as contravariant.

The type $Obj(X)[...]$ can be viewed as a recursive type, but with differences in subtyping that are crucial for object-oriented applications. The subtyping rule for object types (Sub Object), with all components invariant, would read:

$$\frac{E,X<:Top \vdash B_i\{X^+\} \qquad \forall i \in 1..n+m}{E \vdash Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n+m}] <: Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

An analogous rule would be unsound with recursive types instead of Self types [6].

Term typing judgment: $E \vdash a : A$

(Val Subsumption)
$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Val x)
$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x{:}A}$$

(Val Object) (where $A \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}]$)
$$\frac{E, x_i{:}A \vdash b_i : B_i[\![A]\!] \qquad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i) b_i^{\,i\in 1..n}] : A}$$

(Val Select) (where $A' \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}]$)
$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad \upsilon_j \in \{^o, ^+\} \qquad j \in 1..n}{E \vdash a.l_j : B_j[\![A]\!]}$$

(Val Update) (where $A' \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}]$)
$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad E, Y<:A, y{:}Y \vdash c : C \qquad E, Y<:A, y{:}Y, z{:}C, x{:}Y \vdash b : B_j[\![Y]\!] \qquad \upsilon_j \in \{^o, ^-\} \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow (y, z=c)\varsigma(x) b : A}$$

(Val Clone) (where $A' \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}]$)
$$\frac{E \vdash a : A \qquad E \vdash A <: A'}{E \vdash clone(a) : A}$$

The typing rules are largely the same ones we would have for a functional calculus; the main novelty is the construct for method update. Because of the eager evaluation associated with imperative semantics, there is a need for a mechanism to express sequential evaluation. We have considered using *let* for this purpose, but we prefer to use our general form of method update; see section 6.1.

To preserve soundness, the rules for selection and update are restricted: selection cannot operate on contravariant components, while update cannot operate on covariant components.

A remarkable aspect of our type system is that the rules (Val Select), (Val Update), and (Val Clone) are based on structural assumptions about the universe of types (*cf.*

[3]). These assumptions are operationally valid, but would not hold in natural semantic models. For example, the update rule implies that if $x{:}X$ and $X{<:}A'$ where A' is a given object type with an invariant component $l{:}B$, then we may update l in x with a term b of type B yielding an updated object of type X, and not just A'. This rule is based on the assumption that any $X{<:}A'$ is closed under updating of l with elements of B. The closure property holds in a model only if any subtype of A' allows the result of l to be any element of B. Intuitively, this condition may fail because a subtype of A' may be a subset with $l{:}B'$ for B' strictly included in B. Operationally, the closure property holds because any possible instance of X in the course of a computation is a closed object type that, being a subtype of A', has a component l of type exactly B.

Rules based on structural assumptions (structural rules, for short) are critical for Self types; they are required for typing programs satisfactorily. Structural rules allow methods to act parametrically over any $X{<:}A'$, where X is the Self type and A' is a given object type. In section 6 we demonstrate the power of structural rules by examples, and by our representation of classes and inheritance. We can prove that structural rules are sound for our operational semantics (see section 5).

4 Self and Polymorphism

Self types produce an expressive type system, sufficient for interesting examples. However, this type system still lacks facilities for type parameterization. Type parameterization has useful interactions with objects; in particular, it supports method reuse and inheritance, as we show in section 6. To allow for parameterization, we add bounded universal quantifiers [14].

First we extend the syntax of terms. We add two new forms. We write $\lambda()b$ for a type abstraction and $a()$ for a type application. In typed calculi, it is common to find $\lambda(X{<:}A)b$ and $a(A)$ instead. However, we are already committed to an untyped operational semantics, so we strip the types from those terms. Technically, we adopt $\lambda()b$, instead of dropping $\lambda()$ altogether, in order to distinguish the elements of quantified types from those of object types. The distinction greatly simplifies case analysis in proofs.

Additional syntax of terms

a,b	::=	term
	. . .	(as before)
	$\lambda()b$	type abstraction
	$a()$	type application

This choice of syntax affects the operational semantics. In particular, a closure $\langle\lambda()b,S\rangle$, consisting of a type abstraction and a stack, is a result:

Additional results

v	::=	result
	...	(as before)
	$\langle\lambda()b,S\rangle$	type abstraction result

We add two rules to the operational semantics. According to these rules, evaluation stops at type abstractions and is triggered again by type applications. This is a sensible semantics of polymorphism, particularly in presence of side-effects.

Additional term reductions

(Red Fun2)

$$\frac{\sigma\bullet S \vdash \diamond}{\sigma\bullet S \vdash \lambda()b \rightsquigarrow \langle\lambda()b,S\rangle\bullet\sigma}$$

(Red Appl2)

$$\frac{\sigma\bullet S \vdash a \rightsquigarrow \langle\lambda()b,S'\rangle\bullet\sigma' \qquad \sigma'\bullet S' \vdash b \rightsquigarrow v\bullet\sigma''}{\sigma\bullet S \vdash a() \rightsquigarrow v\bullet\sigma''}$$

The typing rules for bounded universal quantifiers are:

Additional typing rules

(Type All<:)

$$\frac{E,X{<:}A \vdash B}{E \vdash \forall(X{<:}A)B}$$

(Sub All)

$$\frac{E \vdash A' <: A \qquad E,X{<:}A' \vdash B <: B'}{E \vdash \forall(X{<:}A)B <: \forall(X{<:}A')B'}$$

(Val Fun2<:)

$$\frac{E,X{<:}A \vdash b : B}{E \vdash \lambda()b : \forall(X{<:}A)B}$$

(Val Appl2<:)

$$\frac{E \vdash b : \forall(X{<:}A)B\{X\} \qquad E \vdash A'{<:}A}{E \vdash b() : B\{\!\{A'\}\!\}}$$

The variance of quantifiers, implied by (Sub All), is the usual one: $\forall(X{<:}A)B$ is contravariant in the bound (A) and covariant in the body (B).

5 Soundness

We show the type soundness of our operational semantics, using an approach similar to subject reduction. Our technique for proving typing soundness is an extension of Harper's [19], but using closures and stacks instead of formal substitutions (see [21, 30, 33] for related techniques). This approach yields a manageable proof for a realistic implementation strategy, and deals easily with typing rules that seem hard to justify denotationally. Our soundness result covers subtyping and polymorphism in the presence of side-effects. The proof is an extension of the one given in [5] for an imperative object calculus with a simpler type structure.

5.1 Basic Notions

The typing of results with respect to stores is delicate. We would not be able to determine the type of a result by examining its substructures recursively, including the ones accessed through the store, because stores may contain loops. Store types, introduced next, allow us to type results independently of particular stores. This is possible because type-sound computations do not store results of different types in the same location. In this section we give an overview of store types and other notions necessary for the proof of subject reduction.

A store type Σ associates a method type to each store location. A method type has the form:

$$M \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i \in 1..n}] \Rightarrow j$$

The type of the self argument for a method of method type M is $Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i \in 1..n}]$, and its result type is $B_j\{\!|Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i \in 1..n}]|\!\}$. The index j on the right of $\Rightarrow$ determines the choice of $B_j\{X\}$.

Using store types we can introduce a judgment for typing results:

$$\Sigma \vDash v : A$$

This means that the result v has type A with respect to the store type Σ. The locations contained in v are assigned types in Σ.

Since in the operational semantics a result is interpreted in a store, we need to connect stores and store types; we use a judgment to express that a store is compatible with a store type:

$$\Sigma \vDash \sigma$$

Checking this judgment reduces to checking that the contents of every store location has the type determined by the store type for that location. Since locations contain closures, we need to determine when a closure has a method type. For this, it is sufficient to check that a stack is compatible with an environment; the environment is then used to type the method. To match a stack with an environment, we need to account for the type variables present in the environment; for this purpose we introduce a type stack. A type stack T is an association of type variables to types, of the form $X_i \mapsto A_i^{\,i \in 1..n}$ where A_i are closed types. We write:

$$\Sigma \vDash S \bullet T : E$$

to mean that the stack S and the type stack T together are compatible with the environment E in Σ. The judgment $\Sigma \vDash S \bullet T : E$ is defined via the result typing judgment, which we have already discussed.

5.2 Typings with Stores

We now present the typing rules for the judgments described informally in the previous section.

Store types

$M ::= Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}] \Rightarrow j$	method type ($j \in 1..n$)
$\Sigma ::= \iota_i \mapsto M_i^{\,i\in 1..n}$	store type (ι_i distinct)
$\Sigma_1(\iota) \triangleq Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}]$	if $\Sigma(\iota) = Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}] \Rightarrow j$
$\Sigma_2(A,\iota) \triangleq B_j\langle\!\langle A \rangle\!\rangle$	if $\Sigma(\iota) = Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}] \Rightarrow j$

Type stacks

$T \equiv X_i \mapsto A_i^{\,i\in 1..n}$	type stack (A_i closed types)
$\diamond\{\leftarrow T\} \triangleq \diamond$	substitution in $\diamond$
$A\{\leftarrow T\} \triangleq A\{X_i \leftarrow A_i^{\,i\in 1..n}\}$	substitution in a type
$(A' <: A)\{\leftarrow T\} \triangleq (A'\{\leftarrow T\}) <: (A\{\leftarrow T\})$	substitution in a subtyping assertion
$(\upsilon A' <: \upsilon A')\{\leftarrow T\} \triangleq \upsilon(A'\{\leftarrow T\}) <: \upsilon(A\{\leftarrow T\})$	substitution in a subtyping assertion (with variance annotations)
$(a:A)\{\leftarrow T\} \triangleq a:(A\{\leftarrow T\})$	substitution in a typing assertion
$\emptyset\{\leftarrow T\} \triangleq \emptyset$	substitution in an environment
$(E,x:A)\{\leftarrow T\} \triangleq E\{\leftarrow T\},x:A\{\leftarrow T\}$	
$(E,X<:A)\{\leftarrow T\} \triangleq E\{\leftarrow T\}$	if $X \in dom(T)$
$(E,X<:A)\{\leftarrow T\} \triangleq E\{\leftarrow T\},X<:A\{\leftarrow T\}$	if $X \notin dom(T)$

Well-formed method type judgment: $\vDash M \in Meth$ (M closed)

(Method Type) (l_i distinct, $\upsilon_i \in \{^{\circ},^{-},^{+}\}$)

$$\frac{\emptyset, X<:Top \vdash B_i\{X^{+}\} \quad j\in 1..n}{\vDash Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i\in 1..n}] \Rightarrow j \in Meth}$$

Well-formed store type judgment: $\Sigma \vDash \diamond$

(Store Type) (ι_i distinct)

$$\frac{\vDash M_i \in Meth \quad \forall i \in 1..n}{\iota_i \mapsto M_i^{\,i\in 1..n} \vDash \diamond}$$

Result typing judgment: $\Sigma \vDash v : A$ (A closed)

(Result Object)

$$\frac{\Sigma \vDash \diamond \quad \Sigma_1(\iota_i) \equiv Obj(X)[l_i \upsilon_i : \Sigma_2(X,\iota_i)^{\,i\in 1..n}] \quad \forall i \in 1..n}{\Sigma \vDash [l_i = \iota_i^{\,i\in 1..n}] : Obj(X)[l_i \upsilon_i : \Sigma_2(X,\iota_i)^{\,i\in 1..n}]}$$

(Result Fun2<:)

$$\frac{\Sigma \vDash S \bullet \emptyset : E \quad E, X<:A \vdash b : B\{X\}}{\Sigma \vDash \langle \lambda() b, S \rangle : \forall(X<:A)B\{X\}}$$

Stack typing judgment: $\Sigma \vDash S \bullet T : E$ ($dom(S) \cup dom(T) = dom(E)$; $rng(T)$ closed)

(Stack $\emptyset$ Typing)

$$\frac{\Sigma \vDash \diamond}{\Sigma \vDash \emptyset \bullet \emptyset : \emptyset}$$

(Stack x Typing)

$$\frac{\Sigma \vDash S \bullet T : E \qquad \Sigma \vDash v : A\{\leftarrow T\} \qquad x \notin dom(E)}{\Sigma \vDash (S, x \mapsto v) \bullet T : E, x{:}A}$$

(Stack X Typing)

$$\frac{\Sigma \vDash S \bullet T : E \qquad \emptyset \vdash B <: A\{\leftarrow T\} \qquad X \notin dom(E)}{\Sigma \vDash S \bullet (T, X \mapsto B) : E, X <: A}$$

Store typing judgment: $\Sigma \vDash \sigma$

(Store Typing)

$$\frac{\Sigma \vDash S_i \bullet \emptyset : E_i \qquad E_i, x_i{:}\Sigma_1(\iota_i) \vdash b_i : \Sigma_2(\Sigma_1(\iota_i), \iota_i) \qquad \forall i \in 1..n}{\Sigma \vDash \iota_i \mapsto \langle \varsigma(x_i) b_i, S_i \rangle^{\,i \in 1..n}}$$

The rule (Store Typing) deals with each closure with respect to the whole store, accounting for cycles in the store.

Type stacks are not included in closures along with stacks; this preserves the untyped character of the operational semantics. Note, in particular, that (Store Typing) deals with an empty type stack. Substitutions $A\{\leftarrow T\}$ must be applied appropriately, as shown in (Stack x Typing) and (Stack X Typing).

5.3 Proof of Subject Reduction

In preparation for the subject reduction theorem, we state a few lemmas, without proof.

The first lemma analyzes the possible forms of a type C that is a supertype of an object type C'. According to this lemma, either C is *Top* or C is itself an object type, and in the latter case the component types of C are subtypes or supertypes of the corresponding component types of C':

Lemma 5-1

If $C' \equiv Obj(X)[l_i \upsilon_i' : B_i'\{X\}^{\,i \in I}]$ and $E \vdash C' <: C$ then either $C \equiv Top$,
or $C \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i \in J}]$ with $J \subseteq I$, and, for $j \in J$ we have:

(1) if $\upsilon_j \in \{^o, ^+\}$ then $\upsilon_j' \in \{^o, ^+\}$ and $E, X <: C' \vdash B_j'\{X\} <: B_j\{X\}$,

(2) if $\upsilon_j \in \{^o, ^-\}$ then $\upsilon_j' \in \{^o, ^-\}$ and $E, X <: C' \vdash B_j\{X\} <: B_j'\{X\}$.

□

The second lemma is analogous, but concerns types of the form $\forall(X <: A)B$:

Lemma 5-2

If $C' \equiv \forall(X<:A')B'$ and $E \vdash C' <: C$ then either $C \equiv Top$,
or $C \equiv \forall(X<:A)B$ with $E \vdash A <: A'$ and $E, X<:A \vdash B' <: B$.

□

The proofs of both of these lemmas are by simple inductions on subtyping derivations.

The third lemma establishes that the syntactic criterion for covariance is a sufficient condition for covariant behavior:

Lemma 5-3 (Variance)

Assume $B\{X^+\}$.

If $E, X<:A \vdash B\{X\}$ and $E \vdash C_1 <: C_2$ and $E \vdash C_2 <: A$ then $E \vdash B\{\!\{C_1\}\!\} <: B\{\!\{C_2\}\!\}$.

□

The proof of this lemma is fairly lengthy, but not surprising; one must prove by induction a stronger claim, dealing with more general environments and with both covariant and contravariant occurrences.

The fourth lemma collects several standard properties with routine proofs. The first one is a substitution property. The second one says that the provability of judgments other than typing judgments does not depend on typing assumptions; for example, it says that if $\emptyset, x:A \vdash A'<:A''$ then $\emptyset \vdash A'<:A''$. The third and the fourth properties say that a type A can be replaced with a subtype A' in an environment. Throughout, $\mathfrak{I}$ ranges over assertions of the forms $\diamond$, B, $B' <: B$, $\upsilon' B' <: \upsilon B$, and $b:B$.

Lemma 5-4

(1) If $E', X<:A, E''\{X\} \vdash \mathfrak{I}\{X\}$ and $E' \vdash A' <: A$ then $E', E''\{\!\{A'\}\!\} \vdash \mathfrak{I}\{\!\{A'\}\!\}$.
(2) If $E', x:A, E'' \vdash \mathfrak{I}$ then $E', E'' \vdash \mathfrak{I}$ for $\mathfrak{I}$ not of the form $b:B$.
(3) If $E', X<:A, E'' \vdash \mathfrak{I}$ and $E \vdash A'<:A$ then $E', X<:A', E'' \vdash \mathfrak{I}$.
(4) If $E', x:A, E'' \vdash \mathfrak{I}$ and $E \vdash A'<:A$ then $E', x:A', E'' \vdash \mathfrak{I}$.

□

The fifth lemma also deals with substitutions, this time of the form $\{\leftarrow T\}$:

Lemma 5-5

Assume that $\Sigma \vDash S \bullet T : E$, then:

(1) $X \in dom(E)$ if and only if $X \in dom(T)$.
(2) $\Sigma \vDash S \bullet \emptyset : E\{\leftarrow T\}$.
(3) If $E, E' \vdash \mathfrak{I}$ then $(E, E')\{\leftarrow T\} \vdash \mathfrak{I}\{\leftarrow T\}$.

□

By Lemma 5-5(3), if $\Sigma \vDash S \bullet T : E$ and $E, E' \vdash \mathfrak{I}$ then $(E, E')\{\leftarrow T\} \vdash \mathfrak{I}\{\leftarrow T\}$. By Lemma 5-5(1), $\Sigma \vDash S \bullet T : E$ implies that $E\{\leftarrow T\}$ has no type variables; therefore, if $\mathfrak{I}\{\leftarrow T\}$ is not of the form $b:B$ then $\emptyset, E'\{\leftarrow T\} \vdash \mathfrak{I}\{\leftarrow T\}$ by Lemma 5-4(2). The proof of the subject reduction theorem will use Lemmas 5-5(3) and 5-4(2) in this fashion several times.

We say that Σ' is an extension of Σ (and write $\Sigma' \geqslant \Sigma$) if $dom(\Sigma) \subseteq dom(\Sigma')$ and for all $\iota \in dom(\Sigma)$, $\Sigma'(\iota) = \Sigma(\iota)$. The final lemma says that Σ' shares some of the properties of Σ.

Lemma 5-6

If $\Sigma \vDash S\bullet T : E$ and $\Sigma' \vDash \diamond$ with $\Sigma' \geqslant \Sigma$, then $\Sigma' \vDash S\bullet T : E$.

If $\Sigma \vDash a : A$ and $\Sigma' \vDash \diamond$ with $\Sigma' \geqslant \Sigma$, then $\Sigma' \vDash a : A$.

□

The two parts of the lemma are proved by a joint induction on the derivations of $\Sigma \vDash S\bullet T : E$ and $\Sigma \vDash a : A$.

The subject reduction theorem is:

Theorem 5-7 (Subject reduction)

If $E \vdash a : A \ \wedge\ \sigma\bullet S \vdash a \rightsquigarrow v\bullet\sigma' \ \wedge\ \Sigma \vDash \sigma \ \wedge\ dom(\sigma)=dom(\Sigma) \ \wedge\ \Sigma \vDash S\bullet T : E$
then there exist a closed type A' and a store type Σ' such that:
$\Sigma' \geqslant \Sigma \ \wedge\ \Sigma' \vDash \sigma' \ \wedge\ dom(\sigma')=dom(\Sigma') \ \wedge\ \Sigma' \vDash v : A' \ \wedge\ \emptyset \vdash A' <: A\{\leftarrow T\}$.

Proof

By induction on the derivation of $\sigma\bullet S \vdash a \rightsquigarrow v\bullet\sigma'$.

Case (Red *x*)

$$\frac{\sigma \bullet S', x\mapsto[l_i=\iota_i\ ^{i\in 1..n}], S'' \vdash \diamond}{\sigma \bullet S', x\mapsto[l_i=\iota_i\ ^{i\in 1..n}], S'' \vdash x \rightsquigarrow [l_i=\iota_i\ ^{i\in 1..n}]\bullet\sigma}$$

By hypothesis, $E \vdash x : A \ \wedge\ \Sigma \vDash \sigma \ \wedge\ dom(\sigma)=dom(\Sigma) \ \wedge\ \Sigma \vDash S\bullet T : E$ where $S \equiv S', x\mapsto[l_i=\iota_i\ ^{i\in 1..n}], S''$. Since $E \vdash x : A$, we must have $E \equiv E', x:A', E''$ for some A' such that $E' \vdash A'$ and $E \vdash A' <: A$. Then, by Lemmas 5-5(3) and 5-4(2), $\emptyset \vdash A'\{\leftarrow T\} <: A\{\leftarrow T\}$.

Now, $\Sigma \vDash S\bullet T : E$ must have been derived from $\Sigma \vDash (S', x\mapsto[l_i=\iota_i\ ^{i\in 1..n}])\bullet T' : E', x:A'$, for some prefix T' of T, and therefore $\Sigma \vDash [l_i=\iota_i\ ^{i\in 1..n}] : A'\{\leftarrow T'\}$. Take $A' \equiv A'\{\leftarrow T'\}$ by (Stack x Typing). We have $A'\{\leftarrow T'\} \equiv A'\{\leftarrow T\}$, because the free variables of A' are included in $dom(E')$, and hence in $dom(T')$ by Lemma 5-5(1), and hence $A' \equiv A'\{\leftarrow T\}$. Take $\Sigma' \equiv \Sigma$.

We conclude $\Sigma' \vDash \sigma \ \wedge\ dom(\sigma)=dom(\Sigma') \wedge\ \Sigma' \vDash [l_i=\iota_i\ ^{i\in 1..n}] : A' \wedge\ \emptyset \vdash A' <: A\{\leftarrow T\}$.

Case (Red Object) (l_i, υ_i distinct)

$$\frac{\sigma\bullet S \vdash \diamond \qquad \iota_i \notin dom(\sigma) \qquad \forall i\in 1..n}{\sigma\bullet S \vdash [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}] \rightsquigarrow [l_i=\iota_i\ ^{i\in 1..n}] \bullet (\sigma, \iota_i\mapsto\langle\varsigma(x_i)b_i, S\rangle\ ^{i\in 1..n})}$$

By hypothesis, $E \vdash [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}] : A \ \wedge\ \Sigma \vDash \sigma \ \wedge\ dom(\sigma)=dom(\Sigma) \ \wedge\ \Sigma \vDash S\bullet T : E$. Since $E \vdash [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}] : A$, we must have $E \vdash [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}] : Obj(X)[l_i\upsilon_i:B_i\{X\}\ ^{i\in 1..n}]$ by (Val Object), for some type $Obj(X)[l_i\upsilon_i:B_i\{X\}\ ^{i\in 1..n}]$ such that $E \vdash Obj(X)[l_i\upsilon_i:B_i\{X\}\ ^{i\in 1..n}] <: A$. Therefore, $E, x_i:Obj(X)[l_i\upsilon_i:B_i\{X\}\ ^{i\in 1..n}] \vdash b_i : B_i\{\!|Obj(X)[l_i\upsilon_i:B_i\{X\}\ ^{i\in 1..n}]|\!\}$ for $i\in 1..n$.

Since we have $E \vdash Obj(X)[l_i\upsilon_i:B_i\{X\}\ ^{i\in 1..n}] <: A$, we must also have $E \vdash Obj(X)[l_i\upsilon_i:B_i\{X\}\ ^{i\in 1..n}]$, and therefore $E, X<:Top \vdash B_i\{X^+\}$ for $i\in 1..n$ and $X\notin dom(E)$. Since $X\notin dom(E)$, we have $X\notin dom(T)$ by Lemma 5-5(1). Hence $\emptyset, X<:Top \vdash B_i\{X\}\{\leftarrow T\}$ by Lemmas 5-5(3) and 5-4(2). Take $A' \equiv Obj(X)[l_i\upsilon_i:B_i\{X\}\ ^{i\in 1..n}]\{\leftarrow T\}$. Therefore, since $A' \equiv Obj(X)[l_i\upsilon_i:B_i\{X\}\{\leftarrow T\}\ ^{i\in 1..n}]$ and $\emptyset, X<:Top \vdash B_i\{X\}\{\leftarrow T\}$, (Method Type) yields $\vDash$

$A^+ \Rightarrow j \in \mathit{Meth}$ for $j \in 1..n$.

Take $\Sigma^+ \equiv \Sigma, \iota_j \mapsto (A^+ \Rightarrow j)^{\,j \in 1..n}$; by (Store Type) we have $\Sigma^+ \vDash \diamond$, because the $\iota_j \notin \mathit{dom}(\sigma)$, and hence $\iota_j \notin \mathit{dom}(\Sigma)$, and because $\vDash A^+ \Rightarrow j \in \mathit{Meth}$ for $j \in 1..n$.

(1) Since Σ^+ is an extension of Σ, we also have $\Sigma^+ \vDash S \bullet T : E$ by Lemma 5-6, so $\Sigma^+ \vDash S \bullet \emptyset : E\{\leftarrow T\}$ by Lemma 5-5(2). From $E, x_i{:}\mathit{Obj}(X)[l_i \upsilon_i{:}B_i\{X\}^{\,i \in 1..n}] \vdash b_i : B_i\{\!|\mathit{Obj}(X)[l_i \upsilon_i{:}B_i\{X\}^{\,i \in 1..n}]\!|\}$ we have $E\{\leftarrow T\}, x_i{:}A^+ \vdash b_i : B_i\{\leftarrow T\}\{\!|A^+|\!|\}$ by Lemma 5-5(3), that is, $E\{\leftarrow T\}, x_i{:}\Sigma^+{}_1(\iota_i) \vdash b_i : \Sigma^+{}_2(\Sigma^+{}_1(\iota_i), \iota_i)$.

(2) We have that σ has the shape $\varepsilon_k \mapsto \langle \varsigma(x_k) b_k, S_k \rangle^{\,k \in 1..m}$. Now, $\Sigma \vDash \sigma$ must come from the (Store Typing) rule, with $\Sigma \vDash S_k \bullet \emptyset : E_k$ and $E_k, x_k{:}\Sigma_1(\varepsilon_k) \vdash b_k : \Sigma_2(\Sigma_1(\varepsilon_k), \varepsilon_k)$. By Lemma 5-6, $\Sigma^+ \vDash S_k \bullet \emptyset : E_k$; moreover $E_k, x_k{:}\Sigma^+{}_1(\varepsilon_k) \vdash b_k : \Sigma^+{}_2(\Sigma^+{}_1(\varepsilon_k), \varepsilon_k)$, because $\Sigma^+(\varepsilon_k) = \Sigma(\varepsilon_k)$ for $k \in 1..m$ since $\mathit{dom}(\sigma) = \mathit{dom}(\Sigma) = \{\varepsilon_k^{\,k \in 1..m}\}$ and Σ^+ extends Σ.

By (1) and (2), via the (Store Typing) rule, we have $\Sigma^+ \vDash (\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, S \rangle^{\,i \in 1..n})$. Since $\Sigma^+ \vDash \diamond$ and $\Sigma^+ \equiv \Sigma, \iota_j \mapsto (A^+ \Rightarrow j)^{\,j \in 1..n}$, by the (Result Object) rule, we have $\Sigma^+ \vDash [l_i = \iota_i^{\,i \in 1..n}] : A^+$.

We conclude that $\Sigma^+ \succcurlyeq \Sigma \;\wedge\; \Sigma^+ \vDash (\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, S \rangle^{\,i \in 1..n}) \;\wedge\; \mathit{dom}(\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, S \rangle^{\,i \in 1..n}) = \mathit{dom}(\Sigma^+) \wedge \Sigma^+ \vDash [l_i = \iota_i^{\,i \in 1..n}] : A^+ \;\wedge\; \emptyset \vdash A^+ <: A\{\leftarrow T\}$.

Case (Red Select)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i^{\,i \in 1..n}] \bullet \sigma' \qquad \sigma'(\iota_j) = \langle \varsigma(x_j) b_j, S' \rangle \qquad x_j \notin \mathit{dom}(S') \qquad j \in 1..n \qquad \sigma' \bullet S', x_j \mapsto [l_i = \iota_i^{\,i \in 1..n}] \vdash b_j \rightsquigarrow v \bullet \sigma''}{\sigma \bullet S \vdash a.l_j \rightsquigarrow v \bullet \sigma''}$$

By hypothesis $E \vdash a.l_j : A \;\wedge\; \Sigma \vDash \sigma \;\wedge\; \mathit{dom}(\sigma) = \mathit{dom}(\Sigma) \;\wedge\; \Sigma \vDash S \bullet T : E$.

Since $E \vdash a.l_j : A$, we must have $E \vdash a.l_j : B_j\{\!|C|\!|\}$ by (Val Select), with $E \vdash B_j\{\!|C|\!|\} <: A$, and $E \vdash a : C$ and $E \vdash C <: D$ where D has the form $\mathit{Obj}(X)[l_j \upsilon_j{:}B_j\{X\}, ...]$ and $\upsilon_j \in \{^o, ^+\}$.

By induction hypothesis, since $E \vdash a : C \;\wedge\; \sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i^{\,i \in 1..n}] \bullet \sigma' \wedge \Sigma \vDash \sigma \wedge \mathit{dom}(\sigma) = \mathit{dom}(\Sigma) \wedge \Sigma \vDash S \bullet T : E$, there exist a closed type A' and a store type Σ' such that $\Sigma' \succcurlyeq \Sigma \;\wedge\; \Sigma' \vDash \sigma' \;\wedge\; \mathit{dom}(\sigma') = \mathit{dom}(\Sigma') \;\wedge\; \Sigma' \vDash [l_i = \iota_i^{\,i \in 1..n}] : A' \wedge \emptyset \vdash A' <: C\{\leftarrow T\}$.

Since $\sigma'(\iota_j) = \langle \varsigma(x_j) b_j, S' \rangle$, the judgment $\Sigma' \vDash \sigma'$ must come via (Store Typing) from $\Sigma' \vDash S' \bullet \emptyset : E_j$ and $E_j, x_j{:}\Sigma'_1(\iota_j) \vdash b_j : \Sigma'_2(\Sigma'_1(\iota_j), \iota_j)$ for some E_j. Since $\Sigma' \vDash [l_i = \iota_i^{\,i \in 1..n}] : A'$ must come from (Result Object), we have $A' \equiv \mathit{Obj}(X)[l_i \upsilon_i'{:}\Sigma'_2(X, \iota_j)^{\,i \in 1..n}] \equiv \Sigma'_1(\iota_j)$. Since $\emptyset \vdash A' <: C\{\leftarrow T\} <: D\{\leftarrow T\}$ by Lemma 5-1 we must have $\upsilon_j' \in \{^o, ^+\}$ and $\emptyset, X <: A' \vdash \Sigma'_2(X, \iota_j) <: B_j\{X\}\{\leftarrow T\}$ with $X \notin \mathit{dom}(T)$. Hence $\emptyset \vdash \Sigma'_2(A', \iota_j) <: B_j\{\!|A'|\!|\}\{\leftarrow T\}$ by Lemma 5-4(1). Then, from $E_j, x_j{:}\Sigma'_1(\iota_j) \vdash b_j : \Sigma'_2(\Sigma'_1(\iota_j), \iota_j)$, that is, from $E_j, x_j{:}A' \vdash b_j : \Sigma'_2(A', \iota_j)$, we obtain $E_j, x_j{:}A' \vdash b_j : B_j\{\!|A'|\!|\}\{\leftarrow T\}$ by subsumption. Moreover, from $\Sigma' \vDash S' \bullet \emptyset : E_j$ and $\Sigma' \vDash [l_i = \iota_i^{\,i \in 1..n}] : A'$ we get $\Sigma' \vDash (S', x_j \mapsto [l_i = \iota_i^{\,i \in 1..n}]) \bullet \emptyset : E_j, x_j{:}A'$ by (Stack x Typing).

Let $E' \equiv E_j, x_j{:}A'$. By induction hypothesis, since $E' \vdash b_j : B_j\{\!|A'|\!|\}\{\leftarrow T\} \;\wedge\; \sigma' \bullet S', x_j \mapsto [l_i = \iota_i^{\,i \in 1..n}] \vdash b_j \rightsquigarrow v \bullet \sigma'' \;\wedge\; \Sigma' \vDash \sigma' \;\wedge\; \mathit{dom}(\sigma') = \mathit{dom}(\Sigma') \;\wedge\; \Sigma' \vDash (S', x_j \mapsto [l_i = \iota_i^{\,i \in 1..n}]) \bullet \emptyset : E'$, there exist a closed type A^+ and a store type Σ^+ such that $\Sigma^+ \succcurlyeq \Sigma' \;\wedge\; \Sigma^+ \vDash \sigma'' \;\wedge\; \mathit{dom}(\sigma'') = \mathit{dom}(\Sigma^+) \;\wedge\; \Sigma^+ \vDash v : A^+ \wedge \emptyset \vdash A^+ <: B_j\{\!|A'|\!|\}\{\leftarrow T\}$.

We conclude:

- $\Sigma^\dagger \succcurlyeq \Sigma$ by transitivity from $\Sigma^\dagger \succcurlyeq \Sigma'$ and $\Sigma' \succcurlyeq \Sigma$,
- $\Sigma^\dagger \vDash \sigma''$ with $dom(\sigma'')=dom(\Sigma^\dagger)$,
- $\Sigma^\dagger \vDash v : A^\dagger$,
- $\emptyset \vdash A^\dagger <: A\{\leftarrow T\}$. Since $E,X<:Top \vdash B_j\{X\}$ and $B_j\{X\}$ is covariant in X, we also have $\emptyset,X<:Top \vdash B_j\{X\}\{\leftarrow T\}$ by Lemmas 5-5(3) and 5-4(2), with $B_j\{X\}\{\leftarrow T\}$ covariant in X. Since $\emptyset \vdash A'<:C\{\leftarrow T\}$, we obtain $\emptyset \vdash B_j[\![A']\!]\{\leftarrow T\} <: B_j[\![C\{\leftarrow T\}]\!]\{\leftarrow T\}$, by Lemma 5-3, that is, $\emptyset \vdash B_j[\![A']\!]\{\leftarrow T\} <: B_j[\![C]\!]\{\leftarrow T\}$. Since $E \vdash B_j[\![C]\!] <: A$, we obtain $\emptyset \vdash B_j[\![C]\!]\{\leftarrow T\} <: A\{\leftarrow T\}$ by Lemmas 5-5(3) and 5-4(2). We conclude that $\emptyset \vdash A^\dagger <: A\{\leftarrow T\}$ by transitivity from $\emptyset \vdash A^\dagger <: B_j[\![A']\!]\{\leftarrow T\}$, $\emptyset \vdash B_j[\![A']\!]\{\leftarrow T\} <: B_j[\![C]\!]\{\leftarrow T\}$, and $\emptyset \vdash B_j[\![C]\!]\{\leftarrow T\} <: A\{\leftarrow T\}$.

Case (Red Update)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i=\iota_i{}^{i\in 1..n}] \bullet \sigma' \qquad j\in 1..n \qquad \iota_j \in dom(\sigma') \qquad \sigma' \bullet S, y\mapsto[l_i=\iota_i{}^{i\in 1..n}] \vdash c \rightsquigarrow v \bullet \sigma''}{\sigma \bullet S \vdash a.l_j \Leftarrow (y,z=c)\varsigma(x)b \rightsquigarrow [l_i=\iota_i{}^{i\in 1..n}] \bullet \sigma''.\iota_j \leftarrow\!\!\mapsto \langle \varsigma(x)b, (S, y\mapsto[l_i=\iota_i{}^{i\in 1..n}], z\mapsto v)\rangle}$$

By hypothesis $E \vdash a.l_j \Leftarrow (y,z=c)\varsigma(x)b : A \;\wedge\; \Sigma \vDash \sigma \;\wedge\; dom(\sigma)=dom(\Sigma) \;\wedge\; \Sigma \vDash S \bullet T : E$.

Since $E \vdash a.l_j \Leftarrow (y,z=c)\varsigma(x)b : A$, we must have $E \vdash a.l_j \Leftarrow (y,z=c)\varsigma(x)b : A'$ by (Val Update), for some A' with $E \vdash A'<:A$. Hence we have $E \vdash a : A'$ and $E \vdash A' <: D$ where D has the form $Obj(X)[l_j \upsilon_j:B_j\{X\}, ...]$ and $E, Y<:A', y:Y \vdash c : C$ and $E, Y<:A', y:Y, z:C, x:Y \vdash b : B_j[\![Y]\!]$ with $\upsilon_j \in \{^o,^-\}$. Since $Y \notin dom(E)$, we have $Y \notin dom(T)$ by Lemma 5-5(1).

By induction hypothesis, since $E \vdash a : A' \;\wedge\; \sigma \bullet S \vdash a \rightsquigarrow [l_i=\iota_i{}^{i\in 1..n}] \bullet \sigma' \;\wedge\; \Sigma \vDash \sigma \;\wedge\; dom(\sigma)=dom(\Sigma) \;\wedge\; \Sigma \vDash S \bullet T : E$, then there exist a closed type $A^\dagger$ and a store type Σ' such that $\Sigma' \succcurlyeq \Sigma \;\wedge\; \Sigma' \vDash \sigma' \;\wedge\; dom(\sigma')=dom(\Sigma') \;\wedge\; \Sigma' \vDash [l_i=\iota_i{}^{i\in 1..n}] : A^\dagger \wedge \emptyset \vdash A^\dagger <: A'\{\leftarrow T\}$.

Now, $\Sigma' \vDash [l_i=\iota_i{}^{i\in 1..n}] : A^\dagger$ must have been derived via (Result Object) from $\Sigma' \vDash \diamond$ and $\Sigma'_1(\iota_i) \equiv Obj(X)[l_i \upsilon_i':\Sigma'_2(X,\iota_i)^{\,i\in 1..n}] \equiv A^\dagger$ for all $i\in 1..n$. We have $\emptyset \vdash A'\{\leftarrow T\} <: D\{\leftarrow T\}$ by Lemma 5-5(3) and Lemma 5-4(2), and then $\emptyset \vdash A^\dagger <: D\{\leftarrow T\}$ by transitivity. Hence by Lemma 5-1 we must have $\upsilon_j' \in \{^o,^-\}$ and $\emptyset$, $X<:A^\dagger \vdash B_j\{X\}\{\leftarrow T\} <: \Sigma'_2(X,\iota_j)$, for $X \notin dom(T)$. Therefore, $\emptyset \vdash B_j[\![A^\dagger]\!]\{\leftarrow T\} <: \Sigma'_2(A^\dagger,\iota_j)$ by Lemma 5-4.

Let $T' \equiv T, Y\mapsto A^\dagger$. From $\Sigma \vDash S \bullet T : E$ we have $\Sigma' \vDash S \bullet T : E$ by Lemma 5-6. Then from $\emptyset \vdash A^\dagger <: A'\{\leftarrow T\}$ we obtain $\Sigma' \vDash S \bullet T' : E, Y<:A'$ by (Stack X Typing). Then from $\Sigma' \vDash [l_i=\iota_i{}^{i\in 1..n}] : A^\dagger \equiv Y\{\leftarrow T'\}$ we obtain $\Sigma' \vDash (S, y\mapsto[l_i=\iota_i{}^{i\in 1..n}]) \bullet T' : E, Y<:A', y:Y$ by (Stack x Typing).

By induction hypothesis, since $E, Y<:A', y:Y \vdash c : C \;\wedge\; \sigma' \bullet S, y\mapsto[l_i=\iota_i{}^{i\in 1..n}] \vdash c \rightsquigarrow v \bullet \sigma'' \;\wedge\; \Sigma' \vDash \sigma' \;\wedge\; dom(\sigma')=dom(\Sigma') \;\wedge\; \Sigma' \vDash (S, y\mapsto[l_i=\iota_i{}^{i\in 1..n}]) \bullet T' : E, Y<:A', y:Y$, there exist a closed type $C^\dagger$ and a store type $\Sigma^\dagger$ such that $\Sigma^\dagger \succcurlyeq \Sigma' \;\wedge\; \Sigma^\dagger \vDash \sigma'' \;\wedge\; dom(\sigma'')=dom(\Sigma^\dagger) \;\wedge\; \Sigma^\dagger \vDash v : C^\dagger \wedge \emptyset \vdash C^\dagger <: C\{\leftarrow T'\}$.

Take $\sigma^\dagger \equiv \sigma''.\iota_j \leftarrow\!\!\mapsto \langle \varsigma(x)b, (S, y\mapsto[l_i=\iota_i{}^{i\in 1..n}], z\mapsto v)\rangle$.

- $\Sigma^\dagger \geqslant \Sigma$ by transitivity.
- We have $dom(\Sigma^\dagger) = dom(\sigma')$. Moreover $dom(\sigma'') = dom(\sigma')$ by the definition of reduction, and hence $\iota_j \in dom(\sigma'')$. So $\sigma^\dagger$ is well-defined, and $dom(\sigma'') = dom(\sigma^\dagger)$. Therefore, $dom(\Sigma^\dagger) = dom(\sigma^\dagger)$.
- By (1), (2), and (3) below, we obtain $\Sigma^\dagger \vDash \sigma''.\iota_j \leftarrowtail \langle \varsigma(x)b, (S, y \mapsto [l_i = \iota_i\ ^{i \in 1..n}], z \mapsto v)\rangle$ via the (Store Typing) rule.

  (1) From $\Sigma' \vDash S \bullet T : E$ we have $\Sigma' \vDash S \bullet \emptyset : E\{\leftarrow T\}$ by Lemma 5-5(2). From $\Sigma' \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$ we have $\Sigma' \vDash (S, y \mapsto [l_i = \iota_i\ ^{i \in 1..n}]) \bullet \emptyset : (E\{\leftarrow T\}, y{:}A^\dagger)$ by (Stack x Typing). By Lemma 5-6 we have $\Sigma^\dagger \vDash (S, y \mapsto [l_i = \iota_i\ ^{i \in 1..n}]) \bullet \emptyset : (E\{\leftarrow T\}, y{:}A^\dagger)$ From $\Sigma^\dagger \vDash v : C^\dagger$ we have $\Sigma^\dagger \vDash (S, y \mapsto [l_i = \iota_i\ ^{i \in 1..n}], z \mapsto v) \bullet \emptyset : (E\{\leftarrow T\}, y{:}A^\dagger, z{:}C^\dagger)$ by (Stack x Typing).

  (2) From $\Sigma \vDash S \bullet T : E$ and $E, Y <: A', y{:}Y, z{:}C, x{:}Y \vdash b : B_j$ we obtain $E\{\leftarrow T\}, Y <: A'\{\leftarrow T\}, y{:}Y, z{:}C\{\leftarrow T\}, x{:}Y \vdash b : B_j\{\leftarrow T\}$ by Lemma 5-5(3). Since $\emptyset \vdash A^\dagger <: A'\{\leftarrow T\}$, we have $E\{\leftarrow T\}, y{:}A^\dagger, z{:}C\{\leftarrow T'\}, x{:}A^\dagger \vdash b : B_j\{\leftarrow T'\}$ by Lemma 5-4. Since $\emptyset \vdash C^\dagger <: C\{\leftarrow T'\}$ we have $E\{\leftarrow T\}, y{:}A^\dagger, z{:}C^\dagger, x{:}A^\dagger \vdash b : B_j\{\leftarrow T'\}$ by Lemma 5-4(4). Since $\emptyset \vdash B_j\{\!|A^\dagger|\!\}\{\leftarrow T\} \equiv B_j\{\leftarrow T'\} <: \Sigma'_2(A^\dagger, \iota_j)$ we have $E\{\leftarrow T\}, y{:}A^\dagger, z{:}C^\dagger, x{:}A^\dagger \vdash b : \Sigma'_2(A^\dagger, \iota_j)$ by subsumption. That is, $E\{\leftarrow T\}, y{:}A^\dagger, z{:}C^\dagger, x{:}\Sigma^\dagger{}_1(\iota_j) \vdash b : \Sigma^\dagger{}_2(\Sigma^\dagger{}_1(\iota_j), \iota_i)$.

  (3) Since $\Sigma^\dagger \vDash \sigma''$ must come from (Store Typing), σ'' has the shape $\varepsilon_k \mapsto \langle \varsigma(x_k)b_k, S_k\rangle\ ^{k \in 1..m}$, and for all k such that $\varepsilon_k \neq \iota_j$ and for some E_k we have $\Sigma^\dagger \vDash S_k \bullet \emptyset : E_k$, and $E_k, x_k{:}\Sigma^\dagger{}_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger{}_2(\Sigma^\dagger{}_1(\varepsilon_k), \varepsilon_k)$.
- From $\Sigma' \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$ and $\Sigma^\dagger \geqslant \Sigma'$, by Lemma 5-6 we have $\Sigma^\dagger \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$.
- By Lemma 5-5(3) and Lemma 5-4(2) we have $\emptyset \vdash A'\{\leftarrow T\} <: A\{\leftarrow T\}$, so by transitivity $\emptyset \vdash A^\dagger <: A\{\leftarrow T\}$.

  We conclude $\Sigma^\dagger \geqslant \Sigma \wedge \Sigma^\dagger \vDash \sigma^\dagger \wedge dom(\Sigma^\dagger) = dom(\sigma^\dagger) \wedge \Sigma^\dagger \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger \wedge \emptyset \vdash A^\dagger <: A\{\leftarrow T\}$.

Case (Red Clone) (ι_i' distinct)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i \in 1..n}] \bullet \sigma' \qquad \iota_i \in dom(\sigma') \qquad \iota_i' \notin dom(\sigma') \qquad \forall i \in 1..n}{\sigma \bullet S \vdash clone(a) \rightsquigarrow [l_i = \iota_i'\ ^{i \in 1..n}] \bullet (\sigma', \iota_i' \mapsto \sigma'(\iota_i)\ ^{i \in 1..n})}$$

By hypothesis $E \vdash clone(a) : A \wedge \Sigma \vDash \sigma \wedge dom(\sigma) = dom(\Sigma) \wedge \Sigma \vDash S \bullet T : E$.

Since $E \vdash clone(a) : A$, we must have be $E \vdash clone(a) : A'$ by (Val Clone), for some A' with $E \vdash a : A'$ and $E \vdash A' <: D$ (where D is an object type) and such that $E \vdash A' <: A$.

By the induction hypothesis, since $E \vdash a : A' \wedge \sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i \in 1..n}] \bullet \sigma' \wedge \Sigma \vDash \sigma \wedge dom(\sigma) = dom(\Sigma) \wedge \Sigma \vDash S \bullet T : E$, there exist a closed type $A^\dagger$ and a store type Σ' such that $\Sigma' \geqslant \Sigma \wedge \Sigma' \vDash \sigma' \wedge dom(\sigma') = dom(\Sigma') \wedge \Sigma' \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger \wedge \emptyset \vdash A^\dagger <: A'\{\leftarrow T\}$.

Let $\Sigma^\dagger \equiv (\Sigma', \iota_i' \mapsto \Sigma'(\iota_i)\ ^{i \in 1..n})$ and $\sigma^\dagger \equiv (\sigma', \iota_i' \mapsto \sigma'(\iota_i)\ ^{i \in 1..n})$. We have $\Sigma^\dagger \vDash \diamond$ by (Store Type) because $\iota_i' \notin dom(\sigma') = dom(\Sigma')$, ι_i' are all distinct, and $\Sigma' \vDash \diamond$ is a prerequisite of $\Sigma' \vDash \sigma'$

We conclude:

- $\emptyset \vdash A^\dagger <: A'\{\leftarrow T\} <: A\{\leftarrow T\}$, by Lemmas 5-5(3) and 5-4(2).

- $\Sigma^\dagger \succcurlyeq \Sigma$, because $\Sigma' \succcurlyeq \Sigma$ and $\Sigma^\dagger \succcurlyeq \Sigma'$.
- $dom(\sigma^\dagger)=dom(\Sigma^\dagger)$, by construction and $dom(\sigma')=dom(\Sigma')$.
- We show that $\Sigma^\dagger \vDash \sigma^\dagger$. Since $\Sigma' \vDash \sigma'$ must come from (Store Typing), σ' has the shape $\varepsilon_k \mapsto \langle \varsigma(x_k) b_k, S_k \rangle^{\,k\in 1..m}$, and for all $k\in 1..m$ and for some E_k we have $\Sigma' \vDash S_k \bullet \emptyset : E_k$ and $E_k, x_k{:}\Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma'_2(\Sigma'_1(\varepsilon_k),\varepsilon_k)$. Then also $E_k, x_k{:}\Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\Sigma^\dagger_1(\varepsilon_k),\varepsilon_k)$, and by Lemma 5-6 $\Sigma^\dagger \vDash S_k \bullet \emptyset : E_k$. Let $f{:}1..n \to 1..m$ be $\varepsilon^{-1} \cdot \iota$, so that for all $i\in 1..n$, $\iota_i = \varepsilon_{f(i)}$. We have $E_{f(i)}, x_{f(i)}{:}\Sigma'_1(\varepsilon_{f(i)}) \vdash b_{f(i)} : \Sigma'_2(\Sigma'_1(\varepsilon_{f(i)}),\varepsilon_{f(i)})$ for $i\in 1..n$, so $E_{f(i)}, x_{f(i)}{:}\Sigma'_1(\iota_i) \vdash b_{f(i)} : \Sigma'_2(\Sigma'_1(\iota_i),\iota_i)$. Moreover, since $\Sigma'(\iota_i)=\Sigma^\dagger(\iota_i')$, we have $E_{f(i)}, x_{f(i)}{:}\Sigma^\dagger_1(\iota_i') \vdash b_{f(i)} : \Sigma^\dagger_2(\Sigma^\dagger_1(\iota_i'),\iota_i')$. The result follows by (Store Typing) from $\Sigma^\dagger \vDash S_k \bullet \emptyset : E_k$, and $\Sigma^\dagger \vDash S_{f(i)} \bullet \emptyset : E_{f(i)}$, and $E_k, x_k{:}\Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\Sigma^\dagger_1(\varepsilon_k),\varepsilon_k)$ and $E_{f(i)}, x_{f(i)}{:}\Sigma^\dagger_1(\iota_i') \vdash b_{f(i)} : \Sigma^\dagger_2(\Sigma^\dagger_1(\iota_i'),\iota_i')$, for $k\in 1..m$ and $i\in 1..n$.
- We show that $\Sigma^\dagger \vDash [l_i=\iota_i'^{\,i\in 1..n}] : A^\dagger$. First, $\Sigma' \vDash [l_i=\iota_i^{\,i\in 1..n}] : A^\dagger$ must come from the (Result Object) rule with $A^\dagger \equiv \Sigma'_1(\iota_i) \equiv Obj(X)[l_i \upsilon_i' {:} \Sigma'_2(X,\iota_i)^{\,i\in 1..n}]$ for $i\in 1..n$, and $\Sigma' \vDash \diamond$. But $\Sigma^\dagger(\iota_i') \equiv \Sigma'(\iota_i)$ for $i\in 1..n$. So, $\Sigma^\dagger_1(\iota_i') \equiv \Sigma'_1(\iota_i) \equiv A^\dagger \equiv Obj(X)[l_i \upsilon_i' {:} \Sigma'_2(X,\iota_i)^{\,i\in 1..n}] \equiv Obj(X)[l_i \upsilon_i' {:} \Sigma^\dagger_2(X,\iota_i')^{\,i\in 1..n}]$, and by (Result Object) $\Sigma^\dagger \vDash [l_i=\iota_i'^{\,i\in 1..n}] : Obj(X)[l_i \upsilon_i' {:} \Sigma^\dagger_2(X,\iota_i')^{\,i\in 1..n}]$.

Case (Red Fun2)

$$\frac{}{\sigma \bullet S \vdash \lambda()b \rightsquigarrow \langle \lambda()b, S\rangle \bullet \sigma}$$

By hypothesis, $E \vdash \lambda()b : A \;\wedge\; \Sigma \vDash \sigma \;\wedge\; dom(\sigma)=dom(\Sigma) \;\wedge\; \Sigma \vDash S \bullet T : E$.

From $\Sigma \vDash S \bullet T : E$ and $E \vdash \lambda()b : A$ we have $E\{\leftarrow T\} \vdash \lambda()b : A\{\leftarrow T\}$ by Lemma 5-5(3). Since $E\{\leftarrow T\} \vdash \lambda()b : A\{\leftarrow T\}$ we must have $E\{\leftarrow T\} \vdash \lambda()b : \forall(X{<:}C)B$ for some type $\forall(X{<:}C)B$ such that $E\{\leftarrow T\}, X{<:}C \vdash b : B$ and $E\{\leftarrow T\} \vdash \forall(X{<:}C)B <: A\{\leftarrow T\}$.

We have $\Sigma \vDash S \bullet \emptyset : E\{\leftarrow T\}$ by Lemma 5-5(2), and hence $\Sigma \vDash \langle \lambda()b, S\rangle : \forall(X{<:}C)B$ by (Result Fun2<:). Moreover, $\emptyset \vdash \forall(X{<:}C)B <: A\{\leftarrow T\}$ by Lemma 5-4(2), since $E\{\leftarrow T\}$ has no type variables.

Take $A^\dagger \equiv \forall(X{<:}C)B$ and $\Sigma^\dagger \equiv \Sigma$. We conclude that $\Sigma^\dagger \succcurlyeq \Sigma \wedge\; \Sigma^\dagger \vDash \sigma \wedge dom(\sigma)=dom(\Sigma^\dagger) \;\wedge\; \Sigma^\dagger \vDash \langle \lambda()b, S\rangle : A^\dagger \;\wedge\; \emptyset \vdash A^\dagger <: A\{\leftarrow T\}$.

Case (Red Appl2)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow \langle \lambda()b, S'\rangle \bullet \sigma' \qquad \sigma' \bullet S' \vdash b \rightsquigarrow v \bullet \sigma''}{\sigma \bullet S \vdash a() \rightsquigarrow v \bullet \sigma''}$$

By hypothesis, $E \vdash a() : A \;\wedge\; \Sigma \vDash \sigma \;\wedge\; dom(\sigma)=dom(\Sigma) \;\wedge\; \Sigma \vDash S \bullet T : E$.

Since $E \vdash a() : A$ we must have $E \vdash a() : B\{\!|C|\!\}$ with $E \vdash B\{\!|C|\!\} <: A$ by (Val Appl2<:) for some types C and $\forall(X{<:}D)B\{X\}$ such that $E \vdash a : \forall(X{<:}D)B\{X\}$ and $E \vdash C <: D$. We obtain $\emptyset \vdash C\{\leftarrow T\}{<:}D\{\leftarrow T\}$, by Lemma 5-5(3) and Lemma 5-4(2).

By induction hypothesis, since $E \vdash a : \forall(X{<:}D)B \;\wedge\; \sigma \bullet S \vdash a \rightsquigarrow \langle \lambda()b, S'\rangle \bullet \sigma' \;\wedge\; \Sigma \vDash \sigma \wedge dom(\sigma)=dom(\Sigma) \wedge \Sigma \vDash S \bullet T : E$, there exist a closed type A' and a store type Σ' such

that $\Sigma' \geqslant \Sigma \ \wedge \ \Sigma' \vDash \sigma' \ \wedge \ dom(\sigma')=dom(\Sigma') \ \wedge \ \Sigma' \vDash \langle\lambda()b,S'\rangle : A' \ \wedge \emptyset \vdash A' <: (\forall(X<:D)B)\{\leftarrow T\}$.

Now, $\Sigma' \vDash \langle\lambda()b,S'\rangle : A'$ must come from (Result Fun2<:), with $A' \equiv \forall(X<:D')B'$, $\Sigma \vDash S'\bullet\emptyset : E'$, and $E', X<:D' \vdash b : B'$. We have also $\emptyset \vdash D\{\leftarrow T\} <: D'$ and $\emptyset, X<:D\{\leftarrow T\} \vdash B' <: B\{\leftarrow T\}$ since $\emptyset \vdash A' <: (\forall(X<:D)B)\{\leftarrow T\}$, by Lemma 5-2.

Take $T' \equiv \emptyset$, $X \mapsto C\{\leftarrow T\}$. We have $\Sigma' \vDash S'\bullet\emptyset : E'$ by Lemma 5-6, and $\emptyset \vdash C\{\leftarrow T\} <: D'$ by transitivity. Therefore, $\Sigma' \vDash S'\bullet T' : E', X<:D'$ by (Stack X Typing).

By induction hypothesis, since $E', X<:D' \vdash b : B' \ \wedge \ \sigma'\bullet S' \vdash b \rightsquigarrow v\bullet\sigma'' \ \wedge \ \Sigma' \vDash \sigma' \ \wedge \ dom(\sigma')=dom(\Sigma') \wedge \Sigma' \vDash S'\bullet T' : E', X<:D'$, there exist a closed type $A^\dagger$ and a store type $\Sigma^\dagger$ such that $\Sigma^\dagger \geqslant \Sigma' \ \wedge \ \Sigma^\dagger \vDash \sigma'' \ \wedge \ dom(\sigma'')=dom(\Sigma^\dagger) \ \wedge \ \Sigma^\dagger \vDash v : A^\dagger \ \wedge \ \emptyset \vdash A^\dagger <: B'\{\leftarrow T'\}$.

We have $(B'\{X\})\{\leftarrow T'\} \equiv B'\{\!|C\{\leftarrow T\}|\!\}$ by definition of T', hence $\emptyset \vdash A^\dagger <: B'\{\!|C\{\leftarrow T\}|\!\}$. Since $\emptyset \vdash C\{\leftarrow T\}<:D\{\leftarrow T\}$ and $\emptyset, X<:D\{\leftarrow T\} \vdash B'\{X\} <: B\{X\}\{\leftarrow T\}$, we have $\emptyset \vdash B'\{\!|C\{\leftarrow T\}|\!\} <: B\{\!|C\{\leftarrow T\}|\!\}\{\leftarrow T\}$ by Lemma 5-4(1). Since $E \vdash B\{\!|C|\!\} <: A$, we have $\emptyset \vdash (B\{\!|C|\!\})\{\leftarrow T\} <: A\{\leftarrow T\}$ by Lemmas 5-5(3) and 5-4(2). Finally $\emptyset \vdash A^\dagger <: A\{\leftarrow T\}$ by transitivity.

We conclude that $\Sigma^\dagger \geqslant \Sigma \ \wedge \ \Sigma^\dagger \vDash \sigma'' \ \wedge \ dom(\sigma'')=dom(\Sigma^\dagger) \ \wedge \ \Sigma^\dagger \vDash v : A^\dagger \ \wedge \ \emptyset \vdash A^\dagger <: A\{\leftarrow T\}$.

□

As an immediate corollary we have a simpler statement of subject reduction:

Corollary 5-8

If $\emptyset \vdash a : A$ and $\emptyset\bullet\emptyset \vdash a \rightsquigarrow v\bullet\sigma$
then there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that
$\Sigma^\dagger \vDash \sigma$ and $\Sigma^\dagger \vDash v : A^\dagger$, with $\emptyset \vdash A^\dagger <: A$.

□

6 Applications

We study some challenging examples. Then we show that quantifiers are useful for factoring out methods as generic procedures, and we describe how collections of generic procedures can be organized into classes and subclasses.

6.1 Typing Challenges and Solutions

We examine some delicate typing issues in the context of simple examples. We use procedures, as defined in section 2.3, and booleans. Procedure types can be defined as:

$$A \rightarrow B \ \triangleq \ [arg^-: A, val^+: B]$$

The variance annotations yield the expected contravariant/covariant subtyping rule for procedure types. The typing of procedures relies on the inclusion $[arg^o: A, val^o: B] <: A \rightarrow B$.

- We define the type of memory cells with a *get* field and with a *set* method. Given a

boolean, the *set* method stores it in the cell and returns the modified cell. For *get*, we use a method. (Using a field, with the encoding of section 2.2, leads to simpler code but makes it harder to check the typings of the examples against the rules.)

$$\begin{array}{lcl} \mathit{Mem} & \triangleq & \mathit{Obj}(X)[\mathit{get}{:}\mathit{Bool}, \mathit{set}{:}\mathit{Bool}{\to}X] \\ m{:}\,\mathit{Mem} & \triangleq & [\mathit{get} = \varsigma(x)\, \mathit{false}, \mathit{set} = \varsigma(x)\, \lambda(b)\, x.\mathit{get} \Leftarrow \varsigma(z)\, b] \end{array}$$

• Interesting uses of (Val Update) are required when updating methods that return a value of type Self. Here we update the method *set* with a method that updates *get* to $\varsigma(z)\mathit{false}$:

$$m.\mathit{set} \Leftarrow \varsigma(x)\, \lambda(b)\, x.\mathit{get} \Leftarrow \varsigma(z)\, \mathit{false} \;\; : \;\; \mathit{Mem}$$

To obtain this typing using (Val Update), we give type Y to $x.\mathit{get}\Leftarrow\varsigma(z)\mathit{false}$ for an arbitrary $Y{<:}\mathit{Mem}$, parametrically in Y.

• With quantifiers, we can define "parametric pre-methods" as polymorphic procedures that can later be used in updating. The method previously used for updating *m.set*, for example, can be isolated as a procedure of type $\forall(X{<:}\mathit{Mem})X{\to}X$:

$$\lambda()\, \lambda(m)\, m.\mathit{set} \Leftarrow \varsigma(x)\, \lambda(b)\, x.\mathit{get} \Leftarrow \varsigma(z)\, \mathit{false} \;\; : \;\; \forall(X{<:}\mathit{Mem})X{\to}X$$

The derivation makes an essential use of the structural subtyping assumption in (Val Update); without it we would obtain at best the type $\forall(X{<:}\mathit{Mem})X{\to}\mathit{Mem}$.

• It is natural to expect both components of *Mem* to be protected against external update. To this end we can use covariance annotations, which block (Val Update). Take:

$$\mathit{ProtectedMem} \quad \triangleq \quad \mathit{Obj}(X)[\mathit{get}^{+}{:}\mathit{Bool}, \mathit{set}^{+}{:}\mathit{Bool}{\to}X]$$

Since *Mem* <: *ProtectedMem*, any memory cell can be subsumed into *ProtectedMem* and thus protected against updating from the outside. However, after subsumption, the *set* method can still update the *get* field because it was originally typechecked with X equal to *Mem*.

• Consider a type of memory cells with a duplicate method, as in section 2.4.

$$\mathit{MemDup} \quad \triangleq \quad \mathit{Obj}(X)[\mathit{get}{:}\mathit{Bool}, \mathit{set}{:}\mathit{Bool}{\to}X, \mathit{dup}{:}X]$$

We have $\mathit{MemDup} <: \mathit{Mem}$, thanks to the subtyping rule for object types. This subtyping would have failed had we used recursive type instead of Self types [4].

• Consider now a type of memory cells with backup and restore methods, and a candidate implementation:

$$\begin{array}{lcl} \mathit{MemBk} & \triangleq & \mathit{Obj}(X)[\mathit{restore}{:}X, \mathit{backup}{:}X, \mathit{get}{:}\mathit{Bool}, \mathit{set}{:}\mathit{Bool}{\to}X] \\ o & \triangleq & [\mathit{restore} = \varsigma(\mathit{self})\, \mathit{self}, \mathit{backup} = \varsigma(\mathit{self})\, \mathit{self}.\mathit{restore} := \mathit{clone}(\mathit{self}), \\ & & \mathit{get} = \ldots\,, \mathit{set} = \ldots\,] \end{array}$$

The initial *restore* method is set to return the current memory cell. Whenever the *backup* method is invoked, it places a clone of self into *restore*. Note that *backup* saves the self that is current at backup-invocation time, not the self that will be current at restore-in-

vocation time.

The untyped behavior of *backup* and *restore* are the desired ones, but *o* cannot be given the type *MemBk*. We can see why by expanding the definition of :=, obtaining:

$$backup = \varsigma(self)\ self.restore \Leftarrow (y,z=clone(self))\ \varsigma(x)\ z$$

In typing $self.restore \Leftarrow (y,z=clone(self))\varsigma(x)z$, we have $self:MemBk$ and $z:MemBk$ as well. The update requires that for an arbitrary $Y<:MemBk$ we be able to show that $z:Y$. This cannot be achieved.

The following alternative code has the same problem if we assume the obvious typing rule for let:

$$backup = \varsigma(self)\ let\ z = clone(self)\ in\ self.restore \Leftarrow \varsigma(x)\ z$$

Therefore, for typing this example, it is not sufficient to adopt let and simple update as separate primitives.

The solution to this typing problem requires the general method update construct:

$$backup = \varsigma(self)\ self.restore \Leftarrow (y,z=clone(y))\ \varsigma(x)\ z$$

In typing $self.restore \Leftarrow (y,z=clone(y))\varsigma(x)z$, we have $self:MemBk$. For an arbitrary $Y<:MemBk$, the update rule assigns to y the type Y. Therefore, $clone(y)$ has type Y by (Val Clone), and hence z has the required type Y.

Note that this typing problem manifests itself with field update, and is not a consequence of allowing method update. It arises from the combination of Self and eager evaluation, when a component of a type that depends on Self is updated.

6.2 Classes as Collections of Pre-Methods

As shown in section 6.1, types of the form $\forall(X<:A)X\rightarrow B\{X\}$ (with $B\{X\}$ covariant in X) arise naturally for methods used in updating. In our type system, these types contain useful elements because of our structural assumptions. In contrast, they have no interesting elements in standard models of subtyping; see [8].

Types of this form can be used for defining classes as collections of pre-methods, where a pre-method is a procedure that is later used to construct a method. Each pre-method must work for all possible subclasses, parametrically in self, so that it can be inherited and instantiated to any of these subclasses. This is precisely what a type of the form $\forall(X<:A)X\rightarrow B\{X\}$ expresses.

We associate a class type $Class(A)$ to each object type A. (We make the components of $Class(A)$ invariant, for simplicity.)

$$\begin{array}{ll} \text{If} & A \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\ i\in 1..n}] \\ \text{then} & Class(A) \triangleq [new:A,\ l_i:\forall(X<:A)X\rightarrow B_i\{X\}^{\ i\in 1..n}] \end{array}$$

A class c of type $Class(A)$ consists of a particular collection of pre-methods l_i of the appropriate types, along with a method, called *new*, for constructing new objects. The implementation of *new* is uniform for all classes: it produces an object of type A by collecting all the pre-methods of the class and applying them to the self of the new object.

$$c : Class(A) \quad \triangleq \quad [new = \varsigma(z)\ [l_i = \varsigma(x)\ z.l_i()(x)^{\ i \in 1..n}],\ l_1 = ..., \ldots, l_n = ...\ ]$$

The methods l_i do not normally use the self of the class, but *new* does. For example:

$$\begin{array}{l} Class(Mem) \ \equiv \\ \quad [new: Mem, \\ \quad\ get: \forall(X<:Mem)\ X \to Bool, \\ \quad\ set: \forall(X<:Mem)\ X \to Bool \to X] \\ memClass: Class(Mem) \ \triangleq \\ \quad [new = \varsigma(z)\ [get = \varsigma(x)\ z.get()(x),\ set = \varsigma(x)\ z.set()(x)], \\ \quad\ get = \lambda()\ \lambda(x)\ false, \\ \quad\ set = \lambda()\ \lambda(x)\ \lambda(b)\ x.get := b] \\ m : Mem \quad \triangleq \quad memClass.new \end{array}$$

We can now consider the inheritance relation between classes. Suppose that we have another type $A' \equiv Obj(X)[l_i \upsilon_i' : B_i'\{X\}^{\ i \in 1..n+m}] <: A$, and a corresponding class type $Class(A') \equiv [new: A', l_i: \forall(X<:A')X \to B_i'\{X\}^{\ i \in 1..n+m}]$. For all $i \in 1..n$, we say that:

$$l_i \text{ is inheritable from } Class(A) \text{ to } Class(A') \quad \text{iff} \quad X<:A' \text{ implies } B_i\{X\}<:B_i'\{X\}$$

When l_i is inheritable, we have $\forall(X<:A)X \to B_i\{X\} <: \forall(X<:A')X \to B_i'\{X\}$. So, if $c:Class(A)$ and l_i is inheritable, we obtain $c.l_i : \forall(X<:A')X \to B_i'\{X\}$ by subsumption. Then, $c.l_i$ has the correct type to be reused when building a class $c':Class(A')$. That is, it can be inherited. For example, *get* and *set* are inheritable from *Class*(*Mem*) to *Class*(*MemDup*):

$$\begin{array}{l} Class(MemDup) \ \equiv \\ \quad [new: MemDup, \\ \quad\ get: \forall(X<:MemDup)\ X \to Bool, \\ \quad\ set: \forall(X<:MemDup)\ X \to Bool \to X, \\ \quad\ dup: \forall(X<:MemDup)\ X \to X] \\ memDupClass: Class(MemDup) \ \triangleq \\ \quad [new = \varsigma(z)\ [get = \varsigma(x)\ z.get()(x),\ set = \varsigma(x)\ z.set()(x),\ dup = \varsigma(x)\ z.dup()(x)], \\ \quad\ get = memClass.get, \\ \quad\ set = memClass.set, \\ \quad\ dup = \lambda()\ \lambda(x)\ clone(x)] \end{array}$$

The inheritability condition holds for any invariant component l_i, since in this case $B_i\{X\} \equiv B_i'\{X\}$. For contravariance, if $A' \equiv Obj(X)[l_i^-: B_i'\{X\}, ...]$ and $A \equiv Obj(X)[l_i^-: B_i\{X\}, ...]$ with $A'<:A$, then $X<:A'$ always implies $B_i\{X\}<:B_i'\{X\}$. Thus, inheritability of contravariant components is also guaranteed.

Covariant components do not necessarily correspond to inheritable pre-methods. For example, if $A' \equiv [l^+: Nat]$ and $A \equiv [l^+: Int]$ with $Nat<:Int$, and $c : [new:A, l:\forall(X<:A)\ X \to Int]$, then $c.l$ cannot be inherited into $Class(A') \equiv [new:A', l:\forall(X<:A')X \to Nat]$, because it would produce a bad result. However, a class $c' : Class(A')$ can include a differ-

ent method for l with result type Nat: this corresponds to method specialization on overriding.

In conclusion, covariant components induce mild restrictions in subclassing. Invariant and contravariant components induce no restrictions. In practice, inheritability is expected between a class type C and another class type C' obtained as an extension of C (so that C' and C have identical common components). In this case, inheritability trivially holds for components of any variance.

7 Conclusions

We have described a basic calculus for objects and their types. It includes a sound type system with Self types within an imperative framework. Because of its compactness and expressiveness, this calculus is appealing as a kernel for object-oriented languages that include subsumption and Self types.

The calculus is not class-based, since classes are not built-in, nor delegation-based [31], since the method-lookup mechanism does not delegate invocations. However, the calculus models class-based languages well: classes and inheritance arise from object types and polymorphic types. In delegation-based languages, traits play the role of classes; our calculus can model traits just as easily as classes, along with dynamic delegation based on traits.

Acknowledgments

We thank Jens Palsberg for suggesting improvements in the presentation and in the proofs.

References

[1] Abadi, M., **Baby Modula-3 and a theory of objects**. *Journal of Functional Programming* **4**(2), 249-283. 1994.

[2] Abadi, M. and L. Cardelli, **A semantics of object types**. *Proc. IEEE Symposium on Logic in Computer Science*, 332-341. 1994.

[3] Abadi, M. and L. Cardelli, **A theory of primitive objects: second-order systems**. *Proc. ESOP'94 - European Symposium on Programming*. Springer-Verlag. 1994.

[4] Abadi, M. and L. Cardelli, **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag. 1994.

[5] Abadi, M. and L. Cardelli, **An imperative object calculus: basic typing and soundness**. *Proc. Second ACM SIGPLAN Workshop on State in Programming Languages*, 19-32. Technical Report UIUCDCS-R-95-1900, University of Illinois at Urbana Champaign. 1995.

[6] Amadio, R.M. and L. Cardelli, **Subtyping recursive types**. *ACM Transactions on Programming Languages and Systems* **15**(4), 575-631. 1993.

[7] Bruce, K.B., **A paradigmatic object-oriented programming language: design, static typing and semantics**. *Journal of Functional Programming* **4**(2), 127-206. 1994.

[8] Bruce, K.B. and G. Longo, **A modest model of records, inheritance and bounded quantification**. *Information and Computation* **87**(1/2), 196-240. 1990.

[9] Bruce, K.B., A. Schuett, and R. van Gent, **PolyTOIL: a type-safe polymorphic object-oriented language.** Williams College. 1994.

[10] Borning, A.H., **Classes versus prototypes in object-oriented languages**. *Proc. ACM/IEEE Fall Joint Computer Conference*, 36-40. 1986.

[11] Cardelli, L., **Extensible records in a pure calculus of subtyping**. In *Theoretical Aspects of Object-Oriented Programming,* C.A. Gunter and J.C. Mitchell, ed. MIT Press. 373-425. 1994.

[12] Cardelli, L., **A language with distributed scope**. *Computing Systems,* **8**(1), 27-59. MIT Press. 1995.

[13] Cardelli, L. and J.C. Mitchell, **Operations on records**. *Mathematical Structures in Computer Science* **1**(1), 3-48. 1991.

[14] Cardelli, L., J.C. Mitchell, S. Martini, and A. Scedrov, **An extension of system F with subtyping**. *Information and Computation* **109**(1-2), 4-56. 1994.

[15] Chambers, C., D. Ungar, B.-W. Chang, and U. Hölzle, **Parents are shared parts of objects: inheritance and encapsulation in Self**. *Lisp and Symbolic Computation* **4**(3). 1991.

[16] Cook, W.R., **A proposal for making Eiffel type-safe**. *Proc. European Conference of Object-Oriented Programming*, 57-72. 1989.

[17] Dony, C., J. Malenfant, and P. Cointe, **Prototype-based languages: from a new taxonomy to constructive proposals and their validation.** P*roc. OOPSLA'92*, 201-217. 1992.

[18] Eifrig, J., S. Smith, V. Trifonov, and A. Zwarico, **An interpretation of typed OOP in a language with state**. *Lisp and Symbolic Computation*. (to appear). 1995.

[19] Harper, R., **A simplified account of polymorphic references**. *Information Processing Letters* **51**(4). 1994.

[20] Harper, R. and B. Pierce, **A record calculus based on symmetric concatenation**. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*. 1991.

[21] Leroy, X., **Polymorphic typing of an algorithmic language**. Rapport de Recherche no.1778 (Ph.D Thesis). INRIA. 1992.

[22] Madsen, O.L., B. Møller-Pedersen, and K. Nygaard, **Object-oriented programming in the Beta programming language**. Addison-Wesley. 1993.

[23] Meyer, B., **Object-oriented software construction**. Prentice Hall. 1988.

[24] Mitchell, J.C., F. Honsell, and K. Fisher, **A lambda calculus of objects and method specialization**. *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*. 1993.

[25] Pierce, B.C. and D.N. Turner, **Simple type-theoretic foundations for object-oriented programming**. *Journal of Functional Programming* **4**(2), 207-247. 1994.

[26] Rémy, D., **Typechecking records and variants in a natural extension of ML**. *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, 77-88. 1989.

[27] Szypersky, C., S. Omohundro, and S. Murer, **Engineering a programming language: the type and class system of Sather**. TR-93-064. ICSI, Berkeley. 1993.

[28] Stein, L.A., H. Lieberman, and D. Ungar, **A shared view of sharing: the treaty of Orlando**. In *Object-oriented concepts, applications, and databases,* W. Kim and F. Lochowsky, ed. Addison-Wesley. 31-48. 1988.

[29] Taivalsaari, A., **Object-oriented programming with modes**. *Journal of Object Oriented Programming* **6**(3), 25-32. 1993.

[30] Tofte, M., **Type inference for polymorphic references**. *Information and Computation* **89**, 1-34. 1990.

[31] Ungar, D. and R.B. Smith, **Self: the power of simplicity**. *Proc. OOPSLA'87*. ACM SIGPLAN Notices 2(12). 1987.

[32] Wand, M., **Type inference for record concatenation and multiple inheritance**. *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*, 92-97. 1989.

[33] Wright, A.K. and M. Felleisen, **A syntactic approach to type soundness**. *Information and Computation* **115**(1), 38-94. 1994.

[34] Yonezawa, A. and M. Tokoro, ed. **Object-oriented concurrent programming**. MIT Press. 1987.