An Imperative Object Calculus
Basic Typing and Soundness

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract

We develop an imperative calculus of objects that is both tiny and expressive. Our calculus provides a minimal setting in which to study the operational semantics and the typing rules of object-oriented languages. We prove type soundness using a simple subject-reduction approach.

1. Introduction

Procedural languages are generally well-understood; their constructs are by now standard, and their formal underpinnings are solid. The fundamental features of procedural languages have been distilled into formalisms that prove useful in identifying and explaining issues of implementation, static analysis, semantics, and verification.

An analogous understanding has not yet emerged in object-oriented programming. There is no widespread agreement on the choice of basic constructs and on their properties. Consequently, practical object-oriented languages support many features and programming techniques, often with little concern for orthogonality.

With the aim of clarifying the fundamental features of object-oriented languages, we introduce a tiny but expressive imperative calculus. 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.

When fields and methods are identified it is trivial to convert one into the other, conceptually turning passive data into active computation and vice versa. 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 eliminating fields.

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. However, method update can be seen as a form of dynamic inheritance [25] which is a feature found in object-based languages [7] but not yet in class-based languages [6]. 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 [23]. Method update is one of these better-behaved mechanisms, especially in the absence of delegation, as in our calculus. Method update is statically typable, and can be used to emulate the mode-switching applications of dynamic inheritance [13]. With method update we avoid some dangerous aspects of dynamic inheritance [14, 23], while maintaining its dynamic specialization aspects originally advocated by the Treaty of Orlando [22].

In this paper, we study an untyped calculus (section 2), and then we present a type structure for it (section 3). The only type constructor is 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. We prove the consistency of our rules using a subject-reduction approach (section 4). Our technique is an extension of Harper's [16], using closures and stacks instead of formal substitutions. This approach yields a manageable proof for a realistic implementation strategy.

Elsewhere we have considered functional calculi [2-4]. The main novelty here is the treatment of imperative features, with corresponding proof techniques. In further work [5] we treat second-order type structures (with Self types) for an imperative calculus.

A few other object formalisms have been defined and studied. Many of these rely on purely functional models, with an emphasis on types [1, 8, 10, 12, 17, 19-21, 26]. Others deal with imperative features in the context of concurrency; see for example [28]. The works most closely related to ours are that of Eifrig *et al.* on LOOP [15] and that of Bruce and van Gent on TOIL [9]. LOOP and TOIL are typed, imperative, object-oriented languages with procedures, objects, and classes. Both take procedures, objects, and classes as primitive, with fairly complex rules; they also distinguish methods from fields. LOOP is translated into a somewhat simpler calculus, which includes record, function, reference, recursive, and F-bounded types. Our calculus is centered entirely on objects: procedures and classes can be defined from them. The collections of programs that can be written and typed in these formalisms are different. In spite of this, we all share the goal of modeling imperative object-oriented languages by precise semantic structures and sound type systems.

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 and 2.3 we show how to express convenient constructs such as fields and procedures. We omit how to encode basic data types, control structures, and classes, which can be treated much as in [4]. In section 2.5 we give an operational semantics.

2.1 Syntax

Syntax of the imp-ς calculus

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

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. The letter ς (sigma) is used as a binder for the self parameter of a method; ς(x)b is a method with self parameter x, to be bound to the host object, and body b.

A method invocation o.l results in the evaluation of the body of the method named l, with o bound to the self parameter.

A method update $o.l \Leftarrow \varsigma(y)b$ replaces the method named l with ς(y)b in o, and returns the modified object.

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

The let construct evaluates a term, binds it to a variable, and then evaluates a second term with that variable in scope. Sequential evaluation can be defined from let, by:

$a;b \triangleq \text{let } x=a \text{ in } b,$

for $x \notin FV(b)$.

2.2 Fields

In our **imp**-ς calculus, every component of an object contains a method. However, we can encode fields with eagerly evaluated contents by using the let construct. 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:

Encoding of fields

$[l_i=b_i\ ^{i\in 1..n}, l_j=\varsigma(x_j)b_j\ ^{j\in 1..m}]$		for $y_i \notin FV(b_i\ ^{i\in 1..n}, b_j\ ^{j\in 1..m})$, y_i distinct, $i\in 0..n$
	$\triangleq$ let $y_1=b_1$ in ... let $y_n=b_n$ in $[l_i=\varsigma(y_0)y_i\ ^{i\in 1..n}, l_j=\varsigma(x_j)b_j\ ^{j\in 1..m}]$	
a.l:=b	$\triangleq$ let $y_1=a$ in let $y_2=b$ in $y_1.l \Leftarrow \varsigma(y_0)y_2$	for $y_i \notin FV(b)$, y_i distinct, $i\in 0..2$

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.

By an update, a method can be changed into a field and vice versa. Thus, we use somewhat interchangeably the names selection and invocation.

2.3 Procedures

The **imp**-ς calculus is so minimal that it does not include procedures, but these can be expressed too. We begin by considering informally a call-by-value λ-calculus with side-effects, **imp**-λ, that includes abstraction, application, and assignment to λ-bound variables. For example, assuming arithmetic primitives, (λ(x) x:=x+1; x)(3) is an **imp**-λ term yielding 4. We translate **imp**-λ into **imp**-ς.

Translation of procedures

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

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. (This encoding has similarities with the mechanism of method activation in the Self language [25].)

2.4 A Small Example

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

let m = [get = 0, set = ς(self) λ(b) self.get:=b, dup = ς(self) clone(self)]
in m.set(1); m.get yields 1

This cell can be used as a prototype for building cells, which can then be customized by method update. For example, we may create a cell that accepts only non-negative integers through the set method:

let m = [get = 0, set = ς(self) λ(b) self.get:=b, dup = ς(self) clone(self)]
in m.dup.set ⇐ ς(self) λ(b) if b<0 then self.get:=0 else self.get:=b

2.5 Operational Semantics

We now give an operational semantics that relates terms to results in a global store. Object terms reduce to results 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 ⟨ς(x)b,*S*⟩ is a pair of a method together with a stack that is used for the reduction of the method body. A store maps locations 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 write σ.ι⇐m for the result of putting 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 σ•*S* ⊢ b ⇝ v•σ', 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. The operational semantics is presented formally as follows.

Operational semantics

ι			store location	(e.g., an integer)
v	::=	$[l_i=\iota_i\ ^{i\in 1..n}]$	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 ø) $\dfrac{}{\emptyset \vdash \diamond}$ (Store ι) $\dfrac{\sigma\bullet S \vdash \diamond \quad \iota \notin \mathrm{dom}(\sigma)}{\sigma, \iota\mapsto\langle\varsigma(x)b,S\rangle \vdash \diamond}$

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

(Stack ø) $\dfrac{\sigma \vdash \diamond}{\sigma\bullet\emptyset \vdash \diamond}$ (Stack x) $\dfrac{\sigma\bullet S \vdash \diamond \quad \iota_i \in \mathrm{dom}(\sigma) \quad x \notin \mathrm{dom}(S) \quad l_i,\iota_i \text{ distinct} \quad \forall i\in 1..n}{\sigma \bullet S,\ x\mapsto[l_i=\iota_i\ ^{i\in 1..n}] \vdash \diamond}$

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

(Red x) $\dfrac{\sigma \bullet S',x\mapsto v,S'' \vdash \diamond}{\sigma \bullet S',x\mapsto v,S'' \vdash x \rightsquigarrow v\bullet\sigma}$ (Red Object) $\dfrac{\sigma\bullet S \vdash \diamond \quad \iota_i \notin \mathrm{dom}(\sigma) \quad \iota_i \text{ distinct} \quad \forall i\in 1..n}{\sigma\bullet S \vdash [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}] \rightsquigarrow [l_i=\iota_i\ ^{i\in 1..n}] \bullet (\sigma, \iota_i\mapsto\langle\varsigma(x_i)b_i,S\rangle\ ^{i\in 1..n})}$

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

(Red Update) $\dfrac{\sigma\bullet S \vdash a \rightsquigarrow [l_i=\iota_i\ ^{i\in 1..n}]\bullet\sigma' \quad j\in 1..n \quad \iota_j\in\mathrm{dom}(\sigma')}{\sigma\bullet S \vdash a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow [l_i=\iota_i\ ^{i\in 1..n}] \bullet \sigma'.\iota_j \leftarrow\langle\varsigma(x)b,S\rangle}$

(Red Clone) $\dfrac{\sigma\bullet S \vdash a \rightsquigarrow [l_i=\iota_i\ ^{i\in 1..n}]\bullet\sigma' \quad \iota_i \in \mathrm{dom}(\sigma') \quad \iota'_i \notin \mathrm{dom}(\sigma') \quad \iota'_i \text{ distinct} \quad \forall i\in 1..n}{\sigma\bullet S \vdash \mathrm{clone}(a) \rightsquigarrow [l_i=\iota'_i\ ^{i\in 1..n}] \bullet (\sigma', \iota'_i\mapsto\sigma'(\iota_i)\ ^{i\in 1..n})}$

(Red Let) $\dfrac{\sigma\bullet S \vdash a \rightsquigarrow v'\bullet\sigma' \quad \sigma'\bullet S, x\mapsto v' \vdash b \rightsquigarrow v''\bullet\sigma''}{\sigma\bullet S \vdash \mathrm{let}\ x = a\ \mathrm{in}\ b \rightsquigarrow v''\bullet\sigma''}$

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 reduces its object to a result, and activates the appropriate method closure. An update operation reduces its object to a result, and updates the appropriate store loca-

tion with a new method closure. 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. A let construct reduces to the result of reducing its body in a stack extended with the bound variable and the result of its associated term.

We illustrate reduction with two examples. The first one is a simple terminating reduction for the term [l = ς(x)[]].l. The following represents a (partial) derivation tree, with bracketed subtrees:

$$\begin{array}{ll} \left[\begin{array}{ll} \emptyset\bullet\emptyset \vdash [l=\varsigma(x)[]] \rightsquigarrow [l=0]\bullet(0\mapsto\langle\varsigma(x)[],\emptyset\rangle) & \text{by (Red Object)} \\ (0\mapsto\langle\varsigma(x)[],\emptyset\rangle)\bullet(x\mapsto[l=0]) \vdash [] \rightsquigarrow []\bullet(0\mapsto\langle\varsigma(x)[],\emptyset\rangle) & \text{by (Red Object)} \end{array}\right. & \\ \emptyset\bullet\emptyset \vdash [l=\varsigma(x)[]].l \rightsquigarrow []\bullet(0\mapsto\langle\varsigma(x)[],\emptyset\rangle) & \text{by (Red Select)} \end{array}$$

We illustrate method overriding, and the creation of loops through the store, by evaluating the term [l = ς(x) x.l⇐ς(y)x].l.

let $\sigma_0 \equiv 0\mapsto\langle\varsigma(x)x.l\Leftarrow\varsigma(y)x, \emptyset\rangle$ and $\sigma_1 \equiv 0\mapsto\langle\varsigma(y)x, (x\mapsto[l=0])\rangle$

$$\begin{array}{ll} \left[\begin{array}{ll} \emptyset\bullet\emptyset \vdash [l=\varsigma(x)x.l\Leftarrow\varsigma(y)x] \rightsquigarrow [l=0]\bullet\sigma_0 & \text{by (Red Object)} \\ \left[\ \sigma_0\bullet(x\mapsto[l=0]) \vdash x \rightsquigarrow [l=0]\bullet\sigma_0 \right. & \text{by (Red x)} \\ \sigma_0\bullet(x\mapsto[l=0]) \vdash x.l\Leftarrow\varsigma(y)x \rightsquigarrow [l=0]\bullet\sigma_1 & \text{by (Red Update)} \end{array}\right. & \\ \emptyset\bullet\emptyset \vdash [l=\varsigma(x)x.l\Leftarrow\varsigma(y)x].l \rightsquigarrow [l=0]\bullet\sigma_1 & \text{by (Red Select)} \end{array}$$

The store σ_1 contains a loop, because it maps the index 0 to a closure that binds the variable x to a value that contains index 0. Hence, an attempt to read out the result of [l=ς(x)x.l⇐ς(y)x].l by "inlining" the store and stack mappings would produce the infinite term [l=ς(y)[l=ς(y)[l=ς(y)...]]].

The potential for creating loops in the store is characteristic of imperative semantics. Loops in the store complicate reasoning about programs and, as we see in the next chapter, they also demand special attention in the treatment of type soundness.

3. Typing

We define a type system for the untyped calculus of section 2, and give a typed example.

3.1 Typing Rules

The typing rules for objects are the same ones we would have for a functional semantics. They are in fact a superset of those of [4], except that terms do not contain type annotations (to match our untyped operational semantics).

Typing rules

Well-formed environment and type judgments: $E \vdash \diamond$, $E \vdash A$

(Env ∅) $\dfrac{}{\emptyset \vdash \diamond}$

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

(Type Object) (l_i distinct) $\dfrac{E \vdash B_i \qquad \forall i \in 1..n}{E \vdash [l_i:B_i{}^{\,i\in 1..n}]}$

Subtyping judgment: $E \vdash A <: B$

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

(Sub Trans) $\dfrac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$

(Sub Object) (l_i distinct) $\dfrac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i:B_i\ ^{i \in 1..n+m}] <: [l_i:B_i\ ^{i \in 1..n}]}$

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

(Val Subsumption) $\dfrac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$

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

(Val Object) (where $A \equiv [l_i:B_i\ ^{i \in 1..n}]$) $\dfrac{E, x_j:A \vdash b_j : B_j \quad \forall j \in 1..n}{E \vdash [l_i = \varsigma(x_i)b_i\ ^{i \in 1..n}] : A}$

(Val Select) $\dfrac{E \vdash a : [l_i:B_i\ ^{i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j}$

(Val Update) (where $A \equiv [l_i:B_i\ ^{i \in 1..n}]$) $\dfrac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x)b : A}$

(Val Clone) (where $A \equiv [l_i:B_i\ ^{i \in 1..n}]$) $\dfrac{E \vdash a : A}{E \vdash \text{clone}(a) : A}$

(Val Let) $\dfrac{E \vdash a : A \quad E, x:A \vdash b : B}{E \vdash \text{let } x = a \text{ in } b : B}$

The first two groups of rules concern typing environments, types, and the subtyping relation. An object type is a collection of method names and associated method result types. A longer object type is a subtype of a shorter one, without variation in the common type components. The final group concerns typing of values. There is one rule for each construct in the calculus; in addition, a subsumption rule connects value typing with subtyping judgments.

3.2 A Typed Example

This section illustrates how to type a simple imperative example: movable points. In this example we rely on fields and procedures, as encoded in section 2. The encoding of procedures typechecks with $A \rightarrow B$ translated as [arg:A, val:B].

Trivial as it may seem, the example of movable points has been a notorious source of difficulties in functional settings (see [4]). These difficulties have resulted in the use of sophisticated type theories. In an imperative setting, however, some of these difficulties can be avoided altogether.

Consider one-dimensional and two-dimensional points, with coordinate fields (x and y) and methods that modify these fields (mv_x and mv_y). The coordinates are integers. We assume that integers and reals are available, perhaps through an encoding. For example, the origin points are:

$p_1 \triangleq [x = 0, mv\_x = \varsigma(s)\ \lambda(dx)\ s.x := s.x + dx]$

$p_2 \triangleq [x = 0,$
$y = 0,$
$mv\_x = \varsigma(s)\ \lambda(dx)\ s.x := s.x + dx,$
$mv\_y = \varsigma(s)\ \lambda(dy)\ s.y := s.y + dy]$

In the type system of section 3.1, p_1 and p_2 can be given the types:

$P_1 \triangleq [x:Int, mv\_x:Int \rightarrow []]$

$P_2 \triangleq [x,y:Int, mv\_x,mv\_y:Int \rightarrow []]$

where P_2 is a subtype of P_1. This result type [] is obtained by subsumption. The imperative operational semantics produces the desired effect of moving a point, without requiring any particular result type for move methods. In contrast, in a functional framework an informative result type is necessary.

Imperatively, there is no loss of type information when moving a point. For example, suppose that f is defined with one-dimensional points in mind, with the type $P_1 \rightarrow []$, and $norm_2$ is defined for two-dimensional points, with the type $P_2 \rightarrow$Real:

$$f: P_1 \rightarrow [] \triangleq \lambda(p)\ p.mv\_x(1)$$
$$norm_2: P_2 \rightarrow \text{Real} \triangleq \lambda(p)\ sqrt(p.x\text{\^{}}2 + p.y\text{\^{}}2)$$

Since P_2 is a subtype of P_1, $f(p_2)$ is a legal call for $p_2:P_2$. Therefore, the following code typechecks and, as expected, returns 1:

$$f(p_2);\ norm_2(p_2)$$

Thus, we have applied a P_1 procedure to a P_2 point, and after this we are still able to use the point as a member of P_2. In contrast, in a functional setting we may try to write $norm_2(f(p_2))$, which is not well-typed if $f:P_1 \rightarrow []$.

Even in an imperative setting, however, it is common to define methods that produce new objects, as opposed to modifying existing ones. If mv_x is to return a new object, one must declare it with result type P_1 or P_2, to take advantage of any change to the x coordinate. One may try to redefine P_1 and P_2 as recursive types (for example, $P_1 \triangleq \mu(X)[x:\text{Int},\ mv\_x:\text{Int} \rightarrow X]$), but then P_2 is not a subtype of P_1. With this definition, all the typing difficulties common in functional settings resurface.

4. Soundness

We show the type soundness of our operational semantics, using an approach similar to subject reduction. We build on the techniques developed by Tofte, Wright and Felleisen, Leroy, and Harper [16, 18, 24, 27]. Our proof technique is an extension of Harper's, in that we deal with closures and stacks and thus avoid introducing locations into the term language.

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 below, 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. Next we formalize store types and other notions necessary for the proof of soundness.

A store type Σ associates a method type to each store location. A method type has the form $[l_i:B_i\ ^{i \in 1..n}] \Rightarrow B_j$, where $[l_i:B_i\ ^{i \in 1..n}]$ is the type of self, and B_j is the result type, for $j \in 1..n$. The statement of soundness relies on a new judgment, result typing: $\Sigma \vDash v : A$. 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.

To connect stores and store types, we use a judgment $\Sigma \vDash \sigma$. 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 and store types contain method types, 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. We write $\Sigma \vDash S : E$ to mean that the stack S is compatible with the environment E in Σ. Now, since stacks contain results and environments contain types, we can define $\Sigma \vDash S : E$ via the result typing judgment, which we have already discussed. The rule for store typing deals with each closure with respect to the whole store, accounting for cycles in the store.

Store typing rules

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

Well-formed method type and store type judgments: $\vDash M \in \text{Meth},\ \Sigma \vDash \diamond$

(Method Type)

$$\frac{j\in 1..n}{\vDash [l_i:B_i\ ^{i\in 1..n}]\Rightarrow B_j \in \text{Meth}}$$

(Store Type)

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

Result typing judgment: $\Sigma \vDash v : A$

(Result Object)

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

Stack typing judgment: $\Sigma \vDash S : E$

(Stack ø Typing)

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

(Stack x Typing)

$$\frac{\Sigma \vDash S : E \quad \Sigma \vDash [l_i=\iota_i\ ^{i\in 1..n}] : A \quad x \notin \text{dom}(E)}{\Sigma \vDash S,x\mapsto[l_i=\iota_i\ ^{i\in 1..n}] : E,x:A}$$

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

(Store Typing)

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

We say that Σ' is an extension of Σ (and write $\Sigma' \succcurlyeq \Sigma$) iff $\text{dom}(\Sigma) \subseteq \text{dom}(\Sigma')$ and for all $\iota\in\text{dom}(\Sigma)$, $\Sigma'(\iota) = \Sigma(\iota)$.

Lemma 4-1

If $\Sigma \vDash S : E$ and $\Sigma' \vDash \diamond$ with $\Sigma' \succcurlyeq \Sigma$, then $\Sigma' \vDash S : E$.

□

If a term has a type, and the term reduces to a result in a store, then the result can be assigned that type in that store:

Soundness Theorem

If $E \vdash a : A \ \wedge\ \sigma\bullet S \vdash a \rightsquigarrow v\bullet\sigma^\dagger \ \wedge\ \Sigma \vDash \sigma \ \wedge\ \text{dom}(\sigma)=\text{dom}(\Sigma) \ \wedge\ \Sigma \vDash S : E$
then there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that:
$\Sigma^\dagger \succcurlyeq \Sigma \ \wedge\ \Sigma^\dagger \vDash \sigma^\dagger \ \wedge\ \text{dom}(\sigma^\dagger)=\text{dom}(\Sigma^\dagger) \ \wedge\ \Sigma^\dagger \vDash v : A^\dagger \ \wedge\ A^\dagger <: A.$

Proof

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

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 $E \vdash x : A \ \wedge \ \Sigma \vDash \sigma \ \wedge \ dom(\sigma)=dom(\Sigma) \ \wedge \ \Sigma \vDash S', x \mapsto [l_i = \iota_i \ ^{i \in 1..n}], S'' : E$. Because of $E \vdash x : A$, we must have $E \equiv E', x:A^\dagger, E''$ for some $A^\dagger <: A$.

Now, $\Sigma \vDash S', x \mapsto [l_i = \iota_i \ ^{i \in 1..n}], S'' : E$ must have been derived via several applications of (Stack x Typing) from, among others, $\Sigma \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A^\dagger$.

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

Case (Red Object)

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

By hypothesis $E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : A \ \wedge \ \Sigma \vDash \sigma \ \wedge \ dom(\sigma)=dom(\Sigma) \ \wedge \ \Sigma \vDash S : E$. Because of $E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : A$, we must have $E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : [l_i : B_i \ ^{i \in 1..n}]$ by (Val Object), for some $[l_i : B_i \ ^{i \in 1..n}] <: A$. Take $A^\dagger \equiv [l_i : B_i \ ^{i \in 1..n}]$.

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

(1) Since $\Sigma^\dagger$ is an extension of Σ, by Lemma 4-1 we also have $\Sigma^\dagger \vDash S : E$. Since $E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : A^\dagger$, we must have $E, x_i : A^\dagger \vdash b_i : B_i$, that is, $E, x_i : \Sigma^\dagger{}_1(\iota_i) \vdash b_i : \Sigma^\dagger{}_2(\iota_i)$.

(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 $\Sigma \vDash S_k : E_k$ and $E_k, x_k : \Sigma_1(\varepsilon_k) \vdash b_k : \Sigma_2(\varepsilon_k)$. By Lemma 4-1, $\Sigma^\dagger \vDash S_k : E_k$; moreover $E_k, x_k : \Sigma^\dagger{}_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger{}_2(\varepsilon_k)$, because $\Sigma^\dagger(\varepsilon_k)=\Sigma(\varepsilon_k)$ for $k \in 1..m$ since $dom(\sigma) = dom(\Sigma) = \{\varepsilon_k \ ^{k \in 1..m}\}$ and $\Sigma^\dagger$ extends Σ.

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

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

Case (Red Select)

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

By hypothesis $E \vdash a.l_j : A \ \wedge \ \Sigma \vDash \sigma \ \wedge \ dom(\sigma)=dom(\Sigma) \ \wedge \ \Sigma \vDash S : E$. Since $E \vdash a.l_j : A$, we must have $E \vdash a : [l_j : B_j \ ...]$, for some $[l_j : B_j \ ...]$ with $B_j <: A$.

By the first induction hypothesis:

Since $E \vdash a : [l_j : B_j \ ...] \ \wedge \sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet \sigma' \wedge \Sigma \vDash \sigma \wedge dom(\sigma)=dom(\Sigma) \wedge \Sigma \vDash S : E$,
there exist a type A' and a store type Σ' such that:
$\Sigma' \succcurlyeq \Sigma \ \wedge \ \Sigma' \vDash \sigma' \ \wedge \ dom(\sigma')=dom(\Sigma') \ \wedge \ \Sigma' \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A' \wedge \ A' <: [l_j : B_j \ ...]$.

Since $\sigma'(\iota_j) = \langle \varsigma(x_j) b_j, S' \rangle$, the judgment $\Sigma' \vDash \sigma'$ must come via (Store Typing) from $\Sigma' \vDash S' : E_j$ and $E_j, x_j : \Sigma'{}_1(\iota_j) \vdash b_j : \Sigma'{}_2(\iota_j)$ for some E_j. Since $\Sigma' \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A'$ must come from (Result Object), we have $A' \equiv [l_i : \Sigma'{}_2(\iota_j) \ ^{i \in 1..n}] \equiv \Sigma'{}_1(\iota_j)$. Since $A' <: [l_j : B_j \ ...]$, we have $\Sigma'{}_2(\iota_j) \equiv B_j$. Then, from $E_j, x_j : \Sigma'{}_1(\iota_j) \vdash b_j : \Sigma'{}_2(\iota_j)$ we obtain $E_j, x_j : A' \vdash b_j : B_j$. Moreover, by the (Stack x Typing) rule we get $\Sigma' \vDash S', x_j \mapsto [l_i = \iota_i \ ^{i \in 1..n}] : E_j, x : A'$.

Let $E' \equiv E_j, x : A'$. By the second induction hypothesis:

Since $E' \vdash b_j : B_j \ \wedge \ \sigma' \bullet S', x_j \mapsto [l_i = \iota_i \ ^{i \in 1..n}] \vdash b_j \rightsquigarrow v \bullet \sigma'' \ \wedge \ \Sigma' \vDash \sigma' \ \wedge \ dom(\sigma')=dom(\Sigma')$

$\wedge\ \Sigma' \vDash S'$, $x_j \mapsto [l_i=\iota_i\ ^{i\in 1..n}] : E'$,
there exist $A^\dagger$ and $\Sigma^\dagger$ such that:
$\Sigma^\dagger \succcurlyeq \Sigma'\ \wedge\ \Sigma^\dagger \vDash \sigma''\ \wedge\ \mathrm{dom}(\sigma'')=\mathrm{dom}(\Sigma^\dagger)\ \wedge\ \Sigma^\dagger \vDash v : A^\dagger \wedge\ A^\dagger <: B_j$.

We conclude:

- $\Sigma^\dagger \succcurlyeq \Sigma$ by transitivity from $\Sigma^\dagger \succcurlyeq \Sigma'$ and $\Sigma' \succcurlyeq \Sigma$,
- $\Sigma^\dagger \vDash \sigma''$ with $\mathrm{dom}(\sigma'')=\mathrm{dom}(\Sigma^\dagger)$,
- $\Sigma^\dagger \vDash v : A^\dagger$ with $A^\dagger <: A$, by transitivity from $A^\dagger <: B_j$ and $B_j <: A$.

Case (Red Update)

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

By hypothesis $E \vdash a.l_j \Leftarrow \varsigma(x)b : A\ \wedge\ \Sigma \vDash \sigma\ \wedge\ \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma)\ \wedge\ \Sigma \vDash S : E$.

Since $E \vdash a.l_j \Leftarrow \varsigma(x)b : A$, we must have $E \vdash a : [l_j:B_j\ ...]$ and $E, x:[l_j:B_j\ ...] \vdash b : B_j$ for some $[l_j:B_j\ ...] <: A$.

By induction hypothesis:

Since $E \vdash a : [l_j:B_j\ ...]\ \wedge\ \sigma{\bullet}S \vdash a \rightsquigarrow [l_i=\iota_i\ ^{i\in 1..n}]{\bullet}\sigma'\ \wedge\ \Sigma \vDash \sigma\ \wedge\ \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma) \wedge\ \Sigma \vDash S : E$,
there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that:
$\Sigma^\dagger \succcurlyeq \Sigma\ \wedge\ \Sigma^\dagger \vDash \sigma'\ \wedge\ \mathrm{dom}(\sigma')=\mathrm{dom}(\Sigma^\dagger)\ \wedge\ \Sigma^\dagger \vDash [l_i=\iota_i\ ^{i\in 1..n}] : A^\dagger \wedge\ A^\dagger <: [l_j:B_j\ ...]$.

By assumption $\iota_j\in\mathrm{dom}(\sigma')$, hence $\iota_j\in\mathrm{dom}(\Sigma^\dagger)$. But $\Sigma^\dagger \vDash [l_i=\iota_i\ ^{i\in 1..n}] : A^\dagger$ must have been derived via (Result Object) from $\Sigma^\dagger{}_1(\iota_i) \equiv [l_i:\Sigma^\dagger{}_2(\iota_i)\ ^{i\in 1..n}] \equiv A^\dagger$ for all $i \in 1..n$. Hence, since $A^\dagger <: [l_j:B_j\ ...]$, we have $\Sigma^\dagger{}_2(\iota_j) = B_j$. Take $\sigma^\dagger \equiv \sigma'.\iota_j \leftarrowtail \langle\varsigma(x)b,S\rangle$.

(1) We have $\Sigma^\dagger \vDash S : E$ by Lemma 4-1. We also have $E, x:A^\dagger \vdash b : B_j$ by a bound change lemma (from $E, x:[l_j:B_j\ ...] \vdash b : B_j$ and $A^\dagger <: [l_j:B_j\ ...]$), that is $E, x:\Sigma^\dagger{}_1(\iota_j) \vdash b : \Sigma^\dagger{}_2(\iota_j)$.

(2) 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 $\Sigma^\dagger \vDash S_k : E_k$, and $E_k, x_k:\Sigma^\dagger{}_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger{}_2(\varepsilon_k)$.

Then by (1) and (2) we have $\Sigma^\dagger \vDash \sigma'.\iota_j \leftarrowtail \langle\varsigma(x)b,S\rangle$, by the (Store Typing) rule.

We conclude $\Sigma^\dagger \succcurlyeq \Sigma\ \wedge\ \Sigma^\dagger \vDash \sigma^\dagger\ \wedge\ \Sigma^\dagger \vDash [l_i=\iota_i\ ^{i\in 1..n}] : A^\dagger$ and $A^\dagger <: A$ by transitivity from $A^\dagger <: [l_j:B_j\ ...]$ and $[l_j:B_j\ ...] <: A$.

Case (Red Clone)

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

By hypothesis $E \vdash \mathrm{clone}(a) : A\ \wedge\ \Sigma \vDash \sigma\ \wedge\ \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma)\ \wedge\ \Sigma \vDash S : E$. Since $E \vdash \mathrm{clone}(a) : A$, we must have $E \vdash a : A$ (possibly via subsumption).

By the induction hypothesis:

Since $E \vdash a : A\ \wedge\ \sigma{\bullet}S \vdash a \rightsquigarrow [l_i=\iota_i\ ^{i\in 1..n}]{\bullet}\sigma'\ \wedge\ \Sigma \vDash \sigma\ \wedge\ \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma)\ \wedge\ \Sigma \vDash S : E$,
there exist a type $A^\dagger$ and a store type Σ' such that:
$\Sigma' \succcurlyeq \Sigma\ \wedge\ \Sigma' \vDash \sigma'\ \wedge\ \mathrm{dom}(\sigma')=\mathrm{dom}(\Sigma')\ \wedge\ \Sigma' \vDash [l_i=\iota_i\ ^{i\in 1..n}] : A^\dagger \wedge\ A^\dagger <: A$.

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 $\iota'_i \notin \mathrm{dom}(\sigma')=\mathrm{dom}(\Sigma')$, ι'_i are all distinct, and $\Sigma' \vDash \diamond$ is a prerequisite of $\Sigma' \vDash \sigma'$.

We conclude:

- $A^\dagger <: A$.
- $\Sigma^\dagger \succcurlyeq \Sigma$, because $\Sigma' \succcurlyeq \Sigma$ and $\Sigma^\dagger \succcurlyeq \Sigma'$.
- $\mathrm{dom}(\sigma^\dagger)=\mathrm{dom}(\Sigma^\dagger)$, by construction and $\mathrm{dom}(\sigma')=\mathrm{dom}(\Sigma')$.

- $\Sigma^\dagger \vDash \sigma^\dagger$. 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 $\Sigma' \vDash S_k : E_k$ and $E_k, x_k{:}\Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma'_2(\varepsilon_k)$. Then also $E_k, x_k{:}\Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\varepsilon_k)$, and by Lemma 4-1 $\Sigma^\dagger \vDash S_k : E_k$. 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 $E_{f(i)}, x_{f(i)}{:}\Sigma'_1(\varepsilon_{f(i)}) \vdash b_{f(i)} : \Sigma'_2(\varepsilon_{f(i)})$ for $i \in 1..n$, so $E_{f(i)}, x_{f(i)}{:}\Sigma'_1(\iota_i) \vdash b_{f(i)} : \Sigma'_2(\iota_i)$. Moreover, since $\Sigma'(\iota_i) = \Sigma^\dagger(\iota'_i)$, we have $E_{f(i)}, x_{f(i)}{:}\Sigma^\dagger_1(\iota'_i) \vdash b_{f(i)} : \Sigma^\dagger_2(\iota'_i)$. The result follows by (Store Typing) from $\Sigma^\dagger \vDash S_k : E_k$, and $\Sigma^\dagger \vDash S_{f(i)} : E_{f(i)}$, and $E_k, x_k{:}\Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\varepsilon_k)$ and $E_{f(i)}, x_{f(i)}{:}\Sigma^\dagger_1(\iota'_i) \vdash b_{f(i)} : \Sigma^\dagger_2(\iota'_i)$, for $k \in 1..m$ and $i \in 1..n$.
- $\Sigma^\dagger \vDash [l_i = \iota'_i{}^{\,i \in 1..n}] : A^\dagger$. First, $\Sigma' \vDash [l_i = \iota_i{}^{\,i \in 1..n}] : A^\dagger$ must come from the (Result Object) rule with $A^\dagger \equiv \Sigma'_1(\iota_i) \equiv [l_i{:}\Sigma'_2(\iota_i)^{\,i \in 1..n}]$ for $i \in 1..n$, and $\Sigma' \vDash \diamond$. But $\Sigma^\dagger(\iota'_i) \equiv \Sigma'(\iota_i)$ for $i \in 1..n$. So, $\Sigma^\dagger_1(\iota'_i) \equiv [l_i{:}\Sigma^\dagger_2(\iota'_i)^{\,i \in 1..n}] \equiv A^\dagger$, and by (Result Object) $\Sigma^\dagger \vDash [l_i = \iota'_i{}^{\,i \in 1..n}] : [l_i{:}\Sigma^\dagger_2(\iota'_i)^{\,i \in 1..n}]$.

Case (Red Let)

$$\frac{\sigma \bullet S \vdash b \rightsquigarrow v' \bullet \sigma' \qquad \sigma' \bullet S,\ x \mapsto v' \vdash c \rightsquigarrow v'' \bullet \sigma''}{\sigma \bullet S \vdash \text{let } x = c \text{ in } b \rightsquigarrow v'' \bullet \sigma''}.$$

By hypothesis $E \vdash \text{let } x = c \text{ in } b : A \ \wedge\ \Sigma \vDash \sigma \ \wedge\ \text{dom}(\sigma) = \text{dom}(\Sigma) \ \wedge\ \Sigma \vDash S : E$. Since $E \vdash \text{let } x = c \text{ in } b : A$, we must have $E \vdash c : C$ for some C, and $E, x{:}C \vdash b : A$ (possibly via subsumption).

By the first induction hypothesis:

> Since $E \vdash c : C \ \wedge\ \sigma \bullet S \vdash c \rightsquigarrow v' \bullet \sigma' \ \wedge\ \Sigma \vDash \sigma \ \wedge\ \text{dom}(\sigma) = \text{dom}(\Sigma) \ \wedge\ \Sigma \vDash S : E$,
> there exist a type C' and a store type Σ' such that:
> $\Sigma' \succcurlyeq \Sigma \ \wedge\ \Sigma' \vDash \sigma' \ \wedge\ \text{dom}(\sigma') = \text{dom}(\Sigma') \ \wedge\ \Sigma' \vDash v' : C' \wedge\ C' <: C$.

By Lemma 4-1, $\Sigma' \vDash S : E$, hence by (Stack x Typing) $\Sigma' \vDash S,\ x \mapsto v' : E, x{:}C'$. From $E, x{:}C \vdash b : A$ by bound weakening, $E, x{:}C' \vdash b : A$.

By the second induction hypothesis:

> Since $E, x{:}C' \vdash b : A \ \wedge\ \sigma' \bullet S,\ x \mapsto v' \vdash b \rightsquigarrow v'' \bullet \sigma'' \ \wedge\ \Sigma' \vDash \sigma' \ \wedge\ \text{dom}(\sigma') = \text{dom}(\Sigma')$
> $\wedge\ \Sigma' \vDash S,\ x \mapsto v' : E, x{:}C'$,
> there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that:
> $\Sigma^\dagger \succcurlyeq \Sigma' \ \wedge\ \Sigma^\dagger \vDash \sigma'' \ \wedge\ \text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger) \ \wedge\ \Sigma^\dagger \vDash v'' : A^\dagger \wedge\ A^\dagger <: A$.

We conclude that $\Sigma^\dagger \succcurlyeq \Sigma$ (by transitivity), $\Sigma^\dagger \vDash \sigma''$ with $\text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger)$, and $\Sigma^\dagger \vDash v'' : A^\dagger$ with $A^\dagger <: A$.

□

Corollary

> 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 $A^\dagger <: A$.

□

Therefore, if a term has a type, and the term reduces to a result in a store, then the result can be assigned that type in that store. That is, if a term produces a result, it does so by respecting the type that it had been assigned statically.

This statement is vacuous if the term does not produce a result. This can happen either because reduction diverges (the rules are applicable ad infinitum), or because it gets stuck (no rule is applicable at a certain stage).

For each term there is at most one rule whose conclusion matches the syntactic form of the term, and hence is potentially applicable to the term. The rule (Red x) is applicable to x unless x is not defined in the stack. Assuming that b reduces to v, the rule (Red Update) is applicable to $b.l_j \Leftarrow \varsigma(x)c$ provided v has l_j. The applicability of the rule (Red Select) is determined with an analo-

gous condition. Assuming the appropriate subterms converge, the rules (Red Object), (Red Clone), and (Red Let) are always applicable to terms of the corresponding forms. Examining these cases, we can prove that the reduction of a well-typed term in a well-typed store cannot get stuck (although it may diverge).

5. Conclusions

We view our calculus as a small kernel for object-oriented languages. (In fact, its primitives have been used in the Obliq distributed scripting language [11].) The calculus is not class-based, since classes are not built-in, nor delegation-based [25] since the method-lookup mechanism does not delegate invocations. However, the calculus models class-based languages well, as we show in [4, 5]. In delegation-based languages, traits play the role of classes. Our calculus can model traits just as easily as classes, along with dynamic inheritance based on traits. Interpreting delegation fully, though, would require significant formal complications, because of the complexity of method lookup in delegation.

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*. 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**. *Proc. TAPSOFT'95 (to appear)*. Springer-Verlag. 1995.

[6] Birtwistle, G.M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, **Simula Begin**. Studentlitteratur. 1979.

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

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

[9] Bruce, K. and R. van Gent, **TOIL: A new type-safe object-oriented imperative language**. Manuscript. 1993.

[10] 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.

[11] Cardelli, L., **Obliq: A language with distributed scope**. Report n.122. Digital Equipment Corporation, Systems Research Center. 1994.

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

[13] 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.

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

[15] Eifrig, J., S. Smith, V. Trifonov, and A. Zwarico, **An interpretation of typed OOP in a language with state**. Dept. of Computer Science, The Johns Hopkins University. 1993.

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

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

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

[19] 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.

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

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

[22] 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.

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

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

[25] Ungar, D. and R.B. Smith, **Self: the power of simplicity**. *Lisp and Symbolic Computation* **4**(3). 1991.

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

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

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