Object Calculi

Martín Abadi

joint work with Luca Cardelli

Digital Equipment Corporation

Systems Research Center

Understanding Objects

- Many characteristics of object-oriented languages are different presentations of a few general ideas.
- The situation is analogous in procedural programming.

  The λ-calculus has provided a basic, flexible model, and a better understanding of actual languages.

From Functions to Objects

- We develop a calculus of objects, analogous to the λ-calculus but independent.
  - ~ It is entirely based on objects, not on functions.
  - ~ We go in this direction because object types are not easily, or at all, definable in most standard formalisms.
- The calculus of objects is intended as a paradigm and a foundation for object-oriented languages.

- We have, in fact, a family of object calculi:
  - ~ functional and imperative;
  - ~ untyped, first-order, and higher-order.

Untyped and first-order object calculi

Calculus:	ς	$\mathbf{Ob_1}$	$\mathbf{Ob_{1<:}}$	*nn*	$\mathbf{Ob_{1\mu}}$	$\mathbf{Ob_{1<:\mu}}$	*nn*	$\mathbf{imp\varsigma}$	*nn*
objects	•	•	•	•	•	•	•	•	•
object types		•	•	•	•	•	•		•
subtyping			•	•		•	•		•
variance				•					
recursive types					•	•	•		
dynamic types							•		
side-effects								•	•

Higher-order object calculi

Calculus:	**Ob**	**Ob$_{\mu}$**	**Ob$_{<:}$**	**Ob$_{<:\mu}$**	**ςOb**	**S**	**S$_{\forall}$**	***nn***	***Ob***$_{\omega<:\mu}$
objects	•	•	•	•	•	•	•	•	•
object types	•	•	•	•	•	•	•	•	•
subtyping			•	•	•	•	•	•	•
variance			○	○		•	•	•	•
recursive types		•		•					•
dynamic types									
side-effects								•	
quantified types	•	•	•	•			•	•	•
Self types				○	•	•	•	•	○
structural rules						•	•	•	•
type operators									•

There are several other calculi (*e.g.*, Castagna's, Fisher&Mitchell's).

Object Calculi

- As in λ-calculi, we have:
  - ~ operational semantics,
  - ~ denotational semantics,
  - ~ type systems,
  - ~ type inference algorithms (due to J. Palsberg),
  - ~ equational theories,
  - ~ a theory of bisimilarity (due to A. Gordon and G. Rees),
  - ~ examples,
  - ~ (small) language translations,
  - ~ guidance for language design.

The Role of "Functional" Object Calculi

- Functional object calculi are object calculi without side-effects (with or without syntax for functions).
- We have developed both functional and imperative object calculi.
- Functional object calculi have simpler operational semantics.

- "Functional object calculus" sounds odd: objects are supposed to encapsulate state!
- However, many of the techniques developed in the context of functional calculi carry over to imperative calculi.

- Sometimes the same code works functionally and imperatively. Often, imperative versions require just a little more care.
- All transparencies make sense functionally, except those that are obviously imperative.

Plan

- Introduction
- Untyped object calculi
  - ~ functional
  - ~ imperative
- First-order object calculi
  - ~ basic first-order typing and subtyping
  - ~ variance annotations, recursive types, typecase
- Higher-order object calculi
  - ~ type parameterization
  - ~ translations into λ-calculi
  - ~ Self types
- Conclusions

Untyped Object Calculi (Functional)

An Untyped Object Calculus: Syntax

An object is a collection of methods. (Their order does not matter.)

Each method has:

- a bound variable for self (which denotes the object itself),
- a body that produces a result.

The only operations on objects are:

- method invocation,
- method update.

Syntax of the ς-calculus

$a,b ::=$	terms
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

First Examples

An object o with two methods, l and m:

$$o \triangleq [l = \varsigma(x)\ [],\ m = \varsigma(x)\ x.l]$$

- l returns an empty object.
- m invokes l through self.

A storage cell with two methods, *contents* and *set*:

$$cell \triangleq [contents = \varsigma(x)\ 0,\ set = \varsigma(x)\ \lambda(n)\ x.contents \Leftarrow \varsigma(y)\ n]$$

- *contents* returns 0.
- *set* updates *contents* through self.

An Untyped Object Calculus: Reduction

- The notation $b \to c$ means that b reduces to c in one step.
- The substitution of a term c for the free occurrences of a variable x in a term b is written $b\{\!\{x \leftarrow c\}\!\}$, or $b\{\!\{c\}\!\}$ when x is clear from context.

Let $o \equiv [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}]$ $\quad$ (l_i distinct)

$$o.l_j \to b_j\{\!\{x_j \leftarrow o\}\!\} \qquad (j \in 1..n)$$

$$o.l_j \Leftarrow \varsigma(y)b \to [l_j = \varsigma(y)b,\ l_i = \varsigma(x_i) b_i \ ^{i \in (1..n)-\{j\}}] \qquad (j \in 1..n)$$

In addition, if $a \to b$ then $C[a] \to C[b]$ where $C[\text{-}]$ is any context.

We are dealing with a calculus of objects, not of functions.

The semantics is deterministic (Church-Rosser).
It is not imperative or concurrent.

Some Example Reductions

Let $o \triangleq [l=\varsigma(x)x.l]$ divergent method

then $o.l \rightarrow x.l\{\!\{x \leftarrow o\}\!\} \equiv o.l \rightarrow \ldots$

Let $o' \triangleq [l = \varsigma(x)x]$ self-returning method

then $o'.l \rightarrow x\{\!\{x \leftarrow o'\}\!\} \equiv o'$

Let $o'' \triangleq [l = \varsigma(y)\,(y.l \Leftarrow \varsigma(x)x)]$ self-modifying method

then $o''.l \rightarrow (o''.l \Leftarrow \varsigma(x)x) \rightarrow o'$

Expressiveness

- Our calculus is based entirely on methods;
  fields can be seen as methods that do not use their self parameter:

$$[..., l{=}b, ...] \triangleq [..., l{=}\varsigma(y)b, ...] \quad \text{for an unused } y$$

$$o.l{:=}b \triangleq o.l \Leftarrow \varsigma(y)b \quad \text{for an unused } y$$

Terminology

		object attributes	
		fields	methods
object operations	selection	field selection	method invocation
	update	field update	method update

- Method update is the most exotic construct, but:
  - ~ it leads to simpler rules, and
  - ~ it corresponds to features of several languages.

- In addition, we can represent:
  - ~ basic data types,
  - ~ functions,
  - ~ classes and subclasses.

- Some operations on objects are not available:
  - ~ method extraction,
  - ~ object extension,
  - ~ object concatenation,

  because they are atypical and in conflict with subtyping.

Some Examples

These examples are:

- easy to write in the untyped calculus,
- patently object-oriented (in a variety of styles),
- sometimes hard to type.

A Cell

Let $cell \triangleq$

$[contents = 0,$

$set = \varsigma(x)\ \lambda(n)\ x.contents := n]$

Then $cell.set(3)$

$\rightarrow\ (\lambda(n)[contents = 0, set = \varsigma(x)\ \lambda(n)\ x.contents := n]$

$.contents{:=}n)(3)$

$\rightarrow\ [contents = 0, set = \varsigma(x)\lambda(n)\ x.contents := n]$

$.contents{:=}3$

$\rightarrow\ [contents = 3, set = \varsigma(x)\ \lambda(n)\ x.contents := n]$

and $cell.set(3).contents$

$\rightarrow\ \ldots$

$\rightarrow\ 3$

A Cell with an Accessor

$$
\begin{aligned}
&\mathit{gcell} \triangleq \\
&\quad [\mathit{contents} = 0, \\
&\quad\ \mathit{set} = \varsigma(x)\, \lambda(n)\, x.\mathit{contents} := n, \\
&\quad\ \mathit{get} = \varsigma(x)\, x.\mathit{contents}]
\end{aligned}
$$

- The *get* method fetches *contents*.
- A user of the cell may not even know about *contents*.

A Cell with Undo

$$
\begin{aligned}
&\mathit{uncell} \triangleq \\
&\quad [\mathit{contents} = 0, \\
&\quad\ \mathit{set} = \varsigma(x)\, \lambda(n)\, (x.\mathit{undo} := x).\mathit{contents} := n, \\
&\quad\ \mathit{undo} = \varsigma(x)\, x]
\end{aligned}
$$

- The *undo* method returns the cell before the latest call to *set*.
- The *set* method updates the *undo* method, keeping it up to date.

Object-Oriented Booleans

true and *false* are objects with methods *if*, *then*, and *else*.
Initially, *then* and *else* are set to diverge when invoked.

$$true \triangleq [if = \varsigma(x)\, x.then,\ then = \varsigma(x)\, x.then,\ else = \varsigma(x)\, x.else]$$
$$false \triangleq [if = \varsigma(x)\, x.else,\ then = \varsigma(x)\, x.then,\ else = \varsigma(x)\, x.else]$$

then and *else* are updated in the conditional expression:

$$cond(b,c,d) \triangleq ((b.then{:=}c).else{:=}d).if$$

So:

$$cond(true, false, true) \equiv ((true.then{:=}false).else{:=}true).if$$
$$\rightarrow ([if = \varsigma(x)\, x.then,\ then = false,\ else = \varsigma(x)\, x.else].else{:=}true).if$$
$$\rightarrow [if = \varsigma(x)\, x.then,\ then = false,\ else = true].if$$
$$\rightarrow [if = \varsigma(x)\, x.then,\ then = false,\ else = true].then$$
$$\rightarrow false$$

Object-Oriented Natural Numbers

- Each numeral has a *case* field that contains either $\lambda(z)\lambda(s)z$ for zero, or $\lambda(z)\lambda(s)s(x)$ for non-zero, where x is the predecessor (self).

  Informally: $n.case(z)(s)$ = if n is *zero* then z else $s(n\text{-}1)$

- Each numeral has a *succ* method that can modify the *case* field to the non-zero version.

zero is a prototype for the other numerals:

$$zero \triangleq [case = \lambda(z)\,\lambda(s)\,z,\ succ = \varsigma(x)\,x.case := \lambda(z)\,\lambda(s)\,s(x)]$$

So:

$$zero \equiv [case = \lambda(z)\,\lambda(s)\,z,\ succ = \ldots]$$

$$one \triangleq zero.succ \equiv [case = \lambda(z)\,\lambda(s)\,s(zero),\ succ = \ldots]$$

$$pred \triangleq \lambda(n)\,n.case(zero)(\lambda(p)p)$$

A Calculator

The calculator uses method update for storing pending operations.

$$
\begin{aligned}
&\mathit{calculator} \triangleq \\
&\quad [\mathit{arg} = 0.0, \\
&\quad \mathit{acc} = 0.0, \\
&\quad \mathit{enter} = \varsigma(s)\, \lambda(n)\, s.\mathit{arg} := n, \\
&\quad \mathit{add} = \varsigma(s)\, (s.\mathit{acc} := s.\mathit{equals}).\mathit{equals} \Leftarrow \varsigma(s')\, s'.\mathit{acc}+s'.\mathit{arg}, \\
&\quad \mathit{sub} = \varsigma(s)\, (s.\mathit{acc} := s.\mathit{equals}).\mathit{equals} \Leftarrow \varsigma(s')\, s'.\mathit{acc}-s'.\mathit{arg}, \\
&\quad \mathit{equals} = \varsigma(s)\, s.\mathit{arg}]
\end{aligned}
$$

We obtain the following calculator-style behavior:

$$
\begin{aligned}
&\mathit{calculator}\ .\mathit{enter}(5.0)\ .\mathit{equals}=5.0 \\
&\mathit{calculator}\ .\mathit{enter}(5.0)\ .\mathit{sub}\ .\mathit{enter}(3.5)\ .\mathit{equals}=1.5 \\
&\mathit{calculator}\ .\mathit{enter}(5.0)\ .\mathit{add}\ .\mathit{add}\ .\mathit{equals}=15.0
\end{aligned}
$$

Functions as Objects

A function is an object with two slots:

- one for the argument (initially undefined),
- one for the function code.

Translation of the untyped λ-calculus

$$\langle\!\langle x \rangle\!\rangle \triangleq x$$

$$\langle\!\langle \lambda(x)b \rangle\!\rangle \triangleq [arg = \varsigma(x)\, x.arg,\ val = \varsigma(x)\, \langle\!\langle b \rangle\!\rangle \{\!\{x \leftarrow x.arg\}\!\}]$$

$$\langle\!\langle b(a) \rangle\!\rangle \triangleq (\langle\!\langle b \rangle\!\rangle.arg := \langle\!\langle a \rangle\!\rangle).val$$

Self variables get statically nested. A keyword **self** would not suffice.

The translation validates the β rule:

$$\langle\!\langle (\lambda(x)b)(a) \rangle\!\rangle \quad \twoheadrightarrow \quad \langle\!\langle b\{\!\{x \leftarrow a\}\!\} \rangle\!\rangle$$

where $\twoheadrightarrow$ is the reflexive and transitive closure of $\rightarrow$.

For example:

$$\begin{aligned}
\langle\!\langle (\lambda(x)x)(y) \rangle\!\rangle \quad &\triangleq \quad ([arg = \varsigma(x)\ x.arg,\ val = \varsigma(x)\ x.arg].arg := y).val \\
&\rightarrow \quad [arg = \varsigma(x)\ y,\ val = \varsigma(x)\ x.arg].val \\
&\rightarrow \quad [arg = \varsigma(x)\ y,\ val = \varsigma(x)\ x.arg].arg \\
&\rightarrow \quad y \\
&\triangleq \quad \langle\!\langle y \rangle\!\rangle
\end{aligned}$$

The translation has typed and imperative variants.

An Operational Semantics

The reduction rules given so far do not impose any evaluation order.

We now define a deterministic reduction system for the closed terms of the ς-calculus.

- Our intent is to describe an evaluation strategy of the sort commonly used in programming languages.
  - ~ A characteristic of such evaluation strategies is that they are weak in the sense that they do not work under binders.
  - ~ In our setting this means that when given an object $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ we defer reducing the body b_i until l_i is invoked.

An Operational Semantics: Results

- The purpose of the reduction system is to reduce every closed expression to a *result*.
- For the pure ς-calculus, we define a result to be a term of the form $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$.
  - ~ A result is itself an expression.
  - ~ For example, both $[l_1=\varsigma(x)[]]$ and $[l_2=\varsigma(y)[l_1=\varsigma(x)[]].l_1]$ are results.
  - ~ (If we had constants such as natural numbers, we would include them among the results.)

- Our weak reduction relation is denoted $\rightsquigarrow$.
- We write $\vdash a \rightsquigarrow v$ to mean that a reduces to a result v, or that v is the result of a.
- This relation is axiomatized with three rules.

Operational semantics

(Red Object) (where $v \equiv [l_i = \varsigma(x_i) b_i \,^{i \in 1..n}]$)

$$\frac{}{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv [l_i = \varsigma(x_i) b_i \{x_i\} \,^{i \in 1..n}]$)

$$\frac{\vdash a \rightsquigarrow v' \qquad \vdash b_j\{\!\{v'\}\!\} \rightsquigarrow v \qquad j \in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Update)

$$\frac{\vdash a \rightsquigarrow [l_i = \varsigma(x_i) b_i \,^{i \in 1..n}] \qquad j \in 1..n}{\vdash a.l_j \Leftarrow \varsigma(x) b \rightsquigarrow [l_j = \varsigma(x) b,\ l_i = \varsigma(x_i) b_i \,^{i \in (1..n) - \{j\}}]}$$

Operational semantics

(Red Object) (where $v \equiv [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}]$)

$$\frac{}{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv [l_i = \varsigma(x_i) b_i \{x_i\} \ ^{i \in 1..n}]$)

$$\frac{\vdash a \rightsquigarrow v' \qquad \vdash b_j\{\!\{v'\}\!\} \rightsquigarrow v \qquad j \in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Update)

$$\frac{\vdash a \rightsquigarrow [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] \qquad j \in 1..n}{\vdash a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow [l_j = \varsigma(x)b, \ l_i = \varsigma(x_i) b_i \ ^{i \in (1..n) - \{j\}}]}$$

1. Results are not reduced further.

2. In order to evaluate $a.l_j$ we should first calculate the result of a, check that it is in the form $[l_i = \varsigma(x_i) b_i \{x_i\} \ ^{i \in 1..n}]$ with $j \in 1..n$, and then evaluate $b_j\{\!\{[l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}]\}\!\}$.

3. In order to evaluate $a.l_j \Leftarrow \varsigma(x)b$ we should first calculate the result of a, check that it is in the form $[l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}]$ with $j \in 1..n$, and return $[l_j = \varsigma(x)b, \ l_i = \varsigma(x_i) b_i \ ^{i \in (1..n) - \{j\}}]$. We do not compute inside b or the b_i.

The reduction system is deterministic:

If $\vdash a \rightsquigarrow v$ and $\vdash a \rightsquigarrow v'$, then $v \equiv v'$.

The rules for $\rightsquigarrow$ immediately suggest an algorithm for reduction, which constitutes an interpreter for ς-terms.

The next proposition says that $\rightsquigarrow$ is sound with respect to $\twoheadrightarrow$.

Proposition (Soundness of weak reduction)

If $\vdash a \rightsquigarrow v$, then $a \twoheadrightarrow v$.

□

Further, $\rightsquigarrow$ is complete with respect to $\twoheadrightarrow$, in the following sense:

Theorem (Completeness of weak reduction)

Let a be a closed term and v be a result.

If $a \twoheadrightarrow v$, then there exists v' such that $\vdash a \rightsquigarrow v'$.

□

This theorem was proved by Melliès.

Classes

A class is an object with:

- a *new* method, for generating new objects,
- code for methods for the objects generated from the class.

For generating the object:

$$o \triangleq [l_i = \varsigma(x_i)\, b_i \,^{i \in 1..n}]$$

we use the class:

$$c \triangleq [new = \varsigma(z)\, [l_i = \varsigma(x)\, z.l_i(x) \,^{i \in 1..n}],\ l_i = \lambda(x_i)\, b_i \,^{i \in 1..n}]$$

The method *new* is a **generator**. The call $c.new$ yields o.

Each field l_i is a **pre-method**.

A Class for Cells

$$cellClass \triangleq$$
$$[new = \varsigma(z)$$
$$[contents = \varsigma(x)\ z.contents(x),\ set = \varsigma(x)\ z.set(x)],$$
$$contents = \lambda(x)\ 0,$$
$$set = \lambda(x)\ \lambda(n)\ x.contents := n]$$

Writing the *new* method is tedious but straightforward.

Writing the pre-methods is like writing the corresponding methods.

cellClass.new yields a standard cell:

$$[contents = 0,\ set = \varsigma(x)\ \lambda(n)\ x.contents := n]$$

Inheritance

Inheritance is the reuse of pre-methods.

Given a class c with pre-methods $c.l_i^{\,i \in 1..n}$
we may define a new class c':

$$c' \triangleq [new=..., l_i=c.l_i^{\,i \in 1..n}, l_j=...^{\,j \in n+1..m}]$$

We may say that c' is a subclass of c.

Inheritance for Cells

$$
\begin{aligned}
&\mathit{cellClass} \triangleq \\
&\quad [\mathit{new} = \varsigma(z) \\
&\qquad [\mathit{contents} = \varsigma(x)\ z.\mathit{contents}(x),\ \mathit{set} = \varsigma(x)\ z.\mathit{set}(x)], \\
&\quad \mathit{contents} = \lambda(x)\ 0, \\
&\quad \mathit{set} = \lambda(x)\ \lambda(n)\ x.\mathit{contents} := n]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{uncellClass} \triangleq \\
&\quad [\mathit{new} = \varsigma(z)\ [\ldots], \\
&\quad \mathit{contents} = \mathit{cellClass}.\mathit{contents}, \\
&\quad \mathit{set} = \lambda(x)\ \mathit{cellClass}.\mathit{set}(x.\mathit{undo} := x), \\
&\quad \mathit{undo} = \lambda(x)\ x]
\end{aligned}
$$

- The pre-method *contents* is inherited.
- The pre-method *set* is overridden, though using a call to **super**.
- The pre-method *undo* is added.

Untyped Object Calculi (Imperative)

An Imperative Untyped Object Calculus

- An object is still a collection of methods.
- Method update works by side-effect ("in-place").
- Some new operations make sense:
  - ~ let (for controlling execution order),
  - ~ object cloning ("shallow copying").

Syntax of the impς-calculus

a,b ::=	programs
...	(as before)
$let\ x = a\ in\ b$	let
$clone(a)$	cloning

- The semantics is given in terms of stacks and stores.

A Cell with Undo (Revisited)

$$
\begin{aligned}
&\mathit{uncell} \triangleq \\
&\quad [\mathit{contents} = 0, \\
&\quad\ \mathit{set} = \varsigma(x)\, \lambda(n)\, (x.\mathit{undo} := x).\mathit{contents} := n, \\
&\quad\ \mathit{undo} = \varsigma(x)\, x]
\end{aligned}
$$

- The *undo* method returns the cell before the latest call to *set*.
- The *set* method updates the *undo* method, keeping it up to date.

The previous code works only if update has a functional semantics.
An imperative version is:

$$
\begin{array}{l}
\mathit{uncell} \triangleq \\
\quad [\mathit{contents} = 0, \\
\quad\ \mathit{set} = \varsigma(x)\, \lambda(n) \\
\qquad\qquad \mathit{let}\ y = \mathit{clone}(x) \\
\qquad\qquad \mathit{in}\ (x.\mathit{undo} := y).\mathit{contents} := n, \\
\quad\ \mathit{undo} = \varsigma(x)\, x]
\end{array}
$$

(Or write a top-level definition: *let uncell* = [...] ;.)

A Prime-Number Sieve

The next example is an implementation of the prime-number sieve.

This example is meant to illustrate advanced usage of object-oriented features, and not necessarily transparent programming style.

```
let sieve =
    [m = ς(s) λ(n)
            let sieve' = clone(s)
            in   s.prime := n;
                 s.next := sieve';
                 s.m ⇐ ς(s') λ(n')
                         case (n' mod n)
                         when 0 do [],
                         when p+1 do sieve'.m(n');
                 [],
     prime = ς(x) x.prime,
     next = ς(x) x.next];
```

- The sieve starts as a root object which, whenever it receives a prime *p*, splits itself into a filter for multiples of *p*, and a clone of itself.
- As filters accumulate in a pipeline, they prevent multiples of known primes from reaching the root object.
- After the integers from 2 to *n* have been fed to the sieve, there are as many filter objects as there are primes smaller than or equal to *n*, plus a root object.
- Each prime is stored in its filter; the *n*-th prime can be recovered by scanning the pipeline for the *n*-th filter.
- The sieve is used, for example, in the following way:

```
for i in 1..99 do sieve.m(i.succ);   (accumulate the primes ð 100)
sieve.next.next.prime                (returns the third prime)
```

Procedures as Imperative Objects

Translation of an imperative λ-calculus

$$
\begin{aligned}
&\langle\!\langle x \rangle\!\rangle \triangleq x \\
&\langle\!\langle x := a \rangle\!\rangle \triangleq \\
&\quad \mathit{let}\ y = \langle\!\langle a \rangle\!\rangle \\
&\quad \mathit{in}\ x.\mathit{arg} := y \\
&\langle\!\langle \lambda(x)b \rangle\!\rangle \triangleq \\
&\quad [\mathit{arg} = \varsigma(x)\ x.\mathit{arg}, \\
&\quad\ \mathit{val} = \varsigma(x)\ \langle\!\langle b \rangle\!\rangle\{\!\{x \leftarrow x.\mathit{arg}\}\!\}] \\
&\langle\!\langle b(a) \rangle\!\rangle \triangleq \\
&\quad \mathit{let}\ f = \mathit{clone}(\langle\!\langle b \rangle\!\rangle) \\
&\quad \mathit{in}\ \mathit{let}\ y = \langle\!\langle a \rangle\!\rangle \\
&\quad \mathit{in}\ (f.\mathit{arg} := y).\mathit{val}
\end{aligned}
$$

Cloning on application corresponds to allocating a new stack frame.

Imperative Operational Semantics

We give an operational semantics that relates terms to results in a global store.

We say that a term b reduces to a result v to mean that, operationally, b yields v.

Object terms reduce to object results consisting of sequences of store locations, one location for each object component:

$$[l_i = \mathfrak{u}_i \; {}^{i \in 1..n}]$$

To imitate usual implementations, we do not rely on substitutions. The semantics is based on stacks and closures.

- A stack associates variables with results.
- A closure 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.

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:

$$\sigma \bullet S \vdash b \rightsquigarrow v \bullet \sigma'$$

This 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.

We represent stacks and stores as finite sequences.

- $\iota_i \mapsto m_i \,^{i \in 1..n}$ is the store that maps the location ι_i to the closure m_i, for $i \in 1..n$.
- $\sigma.\iota_j \leftarrow m$ is the result of storing *m* in the location ι_j of σ, so if $\sigma \equiv \iota_i \mapsto m_i \,^{i \in 1..n}$ and $j \in 1..n$ then $\sigma.\iota_j \leftarrow m \equiv \iota_j \mapsto m, \iota_i \mapsto m_i \,^{i \in 1..n-\{j\}}$.

Operational semantics

ι	store location	(e.g., an integer)
$v ::= [l_i = \iota_i \ ^{i \in 1..n}]$	result	(l_i distinct)
$\sigma ::= \iota_i \mapsto \langle \varsigma(x_i) b_i, S_i \rangle \ ^{i \in 1..n}$	store	(ι_i distinct)
$S ::= x_i \mapsto v_i \ ^{i \in 1..n}$	stack	(x_i distinct)
$\sigma \vdash \diamond$	well-formed store judgment	
$\sigma \bullet S \vdash \diamond$	well-formed stack judgment	
$\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$	term reduction judgment	

(Store $\emptyset$)

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

(Store ι)

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

(Stack $\emptyset$)

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

(Stack x) (l_i, ι_i distinct)

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

(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)

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

(Red Clone) (ι_i' distinct)

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

(Red Let)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow v' \bullet \sigma' \qquad \sigma' \bullet (S, x \mapsto v') \vdash b \rightsquigarrow v'' \bullet \sigma''}{\sigma \bullet S \vdash \mathit{let}\ x = a\ \mathit{in}\ b \rightsquigarrow v'' \bullet \sigma''}$$

A variable reduces to the result it denotes in the current stack.

An object reduces to a fresh collection of locations, while 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 location with a new method closure.

A cloning operation reduces its object to a result; then it allocates a collection of locations and maps them to the method closures from the object.

Finally, a *let* reduces to the result of reducing its body in a stack extended with the bound variable and the result of its associated term.

Example Executions

- The first example is a simple terminating reduction.

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

- The next one is a divergent reduction.

  An attempt to prove a judgment of the form $\varnothing\bullet\varnothing \vdash [l=\varsigma(x)x.l].l \rightsquigarrow ?\bullet?$ yields an incomplete derivation.

$$\begin{array}{l} \left\downarrow\begin{array}{l} \varnothing\bullet\varnothing \vdash [l=\varsigma(x)x.l] \rightsquigarrow [l=0]\bullet(0\mapsto\langle\varsigma(x)x.l,\varnothing\rangle) \quad \text{by (Red Object)} \\ \quad \left\downarrow\begin{array}{l} (0\mapsto\langle\varsigma(x)x.l,\varnothing\rangle)\bullet(x\mapsto[l=0]) \vdash x \rightsquigarrow [l=0]\bullet(0\mapsto\langle\varsigma(x)x.l,\varnothing\rangle) \quad \text{by (Red } x\text{)} \\ \quad \ldots \quad \ldots \\ \quad (0\mapsto\langle\varsigma(x)x.l,\varnothing\rangle)\bullet(x\mapsto[l=0]) \vdash x.l \rightsquigarrow ?\bullet? \quad \text{by (Red Select)} \end{array}\right. \\ (0\mapsto\langle\varsigma(x)x.l,\varnothing\rangle)\bullet(x\mapsto[l=0]) \vdash x.l \rightsquigarrow ?\bullet? \quad \text{(Red Select)} \end{array}\right. \\ \varnothing\bullet\varnothing \vdash [l=\varsigma(x)x.l].l \rightsquigarrow ?\bullet? \quad \text{(Red Select)} \end{array}$$

  An infinite branch has a repeating pattern.

- As a variation of this example, we can have a divergent reduction that keeps allocating storage.

  Read from the bottom up, the derivation for this reduction has judgments with increasingly large stores, σ_0, σ_1, …:

$$\sigma_0 \triangleq 0 \mapsto \langle \varsigma(x)\mathit{clone}(x).l, \emptyset \rangle$$

$$\sigma_1 \triangleq \sigma_0, 1 \mapsto \langle \varsigma(x)\mathit{clone}(x).l, \emptyset \rangle$$

$$\begin{array}{ll}
\emptyset\bullet\emptyset \vdash [l=\varsigma(x)\mathit{clone}(x).l] \rightsquigarrow [l=0]\bullet\sigma_0 & \text{by (Red Object)} \\
\quad \sigma_0\bullet(x\mapsto[l=0]) \vdash x \rightsquigarrow [l=0]\bullet\sigma_0 & \text{by (Red } x\text{)} \\
\quad \sigma_0\bullet(x\mapsto[l=0]) \vdash \mathit{clone}(x) \rightsquigarrow [l=1]\bullet\sigma_1 & \text{(Red Clone)} \\
\quad \ldots & \ldots \\
\quad \sigma_1\bullet(x\mapsto[l=0]) \vdash \mathit{clone}(x).l \rightsquigarrow ?\bullet? & \text{by (Red Select)} \\
\sigma_0\bullet(x\mapsto[l=0]) \vdash \mathit{clone}(x).l \rightsquigarrow ?\bullet? & \text{(Red Select)} \\
\emptyset\bullet\emptyset \vdash [l=\varsigma(x)\mathit{clone}(x).l].l \rightsquigarrow ?\bullet? & \text{(Red Select)}
\end{array}$$

- Another sort of incomplete derivation arises from dynamic errors.

  In the next example, the error consists in attempting to invoke a method from an object that does not have it.

$\emptyset \bullet \emptyset \vdash [] \rightsquigarrow []\bullet\emptyset$ by (Red Object)

STUCK

$\emptyset \bullet \emptyset \vdash [].l \rightsquigarrow ?\bullet ?$ (Red Select)

- The final example illustrates method updating, and creating loops:

$\sigma_0 \triangleq 0 \mapsto \langle \varsigma(x)x.l \Leftarrow \varsigma(y)x, \emptyset \rangle$

$\sigma_1 \triangleq 0 \mapsto \langle \varsigma(y)x, (x \mapsto [l=0]) \rangle$

$\emptyset \bullet \emptyset \vdash [l=\varsigma(x)x.l \Leftarrow \varsigma(y)x] \rightsquigarrow [l=0] \bullet \sigma_0$ by (Red Object)

$\sigma_0 \bullet (x \mapsto [l=0]) \vdash x \rightsquigarrow [l=0] \bullet \sigma_0$ by (Red x)

$\sigma_0 \bullet (x \mapsto [l=0]) \vdash x.l \Leftarrow \varsigma(y)x \rightsquigarrow [l=0] \bullet \sigma_1$ (Red Update)

$\emptyset \bullet \emptyset \vdash [l=\varsigma(x)x.l \Leftarrow \varsigma(y)x].l \rightsquigarrow [l=0] \bullet \sigma_1$ (Red Select)

The store σ_1 contains a loop: it maps the index 0 to a closure that binds the variable x to a value that contains index 0.

An attempt to read out the result of $[l=\varsigma(x)x.l \Leftarrow \varsigma(y)x].l$ by "inlining" the store and stack mappings would produce the infinite term $[l=\varsigma(y)[l=\varsigma(y)[l=\varsigma(y)\ldots]]]$.

These loops are characteristic of imperative semantics.

Loops in the store complicate reasoning about programs and proofs of type soundness.

First-Order Object Calculi

Object Types and Subtyping

An **object type** is a set of method names and of result types:

$$[l_i{:}B_i{}^{\,i\in 1..n}]$$

An object has type $[l_i{:}B_i{}^{\,i\in 1..n}]$ if it has at least the methods $l_i{}^{i\in 1..n}$, with a self parameter of some type $A <: [l_i{:}B_i{}^{\,i\in 1..n}]$ and a result of type B_i,
e.g., [] and $[l_1 : [], l_2 : []]$.

An object type with more methods is a **subtype** of one with fewer:

$$[l_i{:}B_i{}^{\,i\in 1..n+m}] \;<:\; [l_i{:}B_i{}^{\,i\in 1..n}]$$

A longer object can be used instead of a shorter one by **subsumption**:

$$a{:}A \quad \wedge \quad A{<:}B \quad \Rightarrow \quad a{:}B$$

A First-Order Calculus

Environments:

$$E \equiv x_i{:}A_i^{\ i \in 1..n}$$

Judgments:

$E \vdash \diamond$	environment E is well-formed
$E \vdash A$	A is a type in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash a : A$	a has type A in E

Types:

$A,B \ ::= Top$	the biggest type
$[l_i{:}B_i^{\ i \in 1..n}]$	object type

Terms: as for the untyped calculus (but with types for variables).

First-order type rules for the ς-calculus: rules for objects

(Type Object) (l_i distinct)

$$\frac{E \vdash B_i \qquad \forall i \in 1..n}{E \vdash [l_i{:}B_i{}^{\,i\in 1..n}]}$$

(Sub Object) (l_i distinct)

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

(Val Object) (where $A \equiv [l_i{:}B_i{}^{\,i\in 1..n}]$)

$$\frac{E, x_i{:}A \vdash b_i : B_i \qquad \forall i \in 1..n}{E \vdash [l_i{=}\varsigma(x_i{:}A)b_i{}^{\,i\in 1..n}] : A}$$

(Val Select)

$$\frac{E \vdash a : [l_i{:}B_i{}^{\,i\in 1..n}] \qquad j \in 1..n}{E \vdash a.l_j : B_j}$$

(Val Update) (where $A \equiv [l_i{:}B_i{}^{\,i\in 1..n}]$)

$$\frac{E \vdash a : A \qquad E, x{:}A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : A}$$

(Val Clone) (where $A \equiv [l_i{:}B_i{}^{\,i\in 1..n}]$)

$$\frac{E \vdash a : A}{E \vdash clone(a) : A}$$

First-order type rules for the ς-calculus: standard rules

(Env ∅)

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

(Env x)

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

(Val x)

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

(Sub Refl)

$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)

$$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Val Subsumption)

$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Type Top)

$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Sub Top)

$$\frac{E \vdash A}{E \vdash A <: Top}$$

(Val Let)

$$\frac{E \vdash a : A \qquad E, x{:}A \vdash b : B}{E \vdash let\ x{=}a\ in\ b : B}$$

An Operational Semantics (with Types)

We extend the functional operational semantics to typed terms.

A *result* is a term of the form $[l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$.

Operational semantics

(Red Object) (where $v \equiv [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$)

$$\frac{}{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv [l_i=\varsigma(x_i:A_i)b_i\{x_i\}\ ^{i\in 1..n}]$)

$$\frac{\vdash a \rightsquigarrow v' \qquad \vdash b_j\{\!\{v'\}\!\} \rightsquigarrow v \qquad j\in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

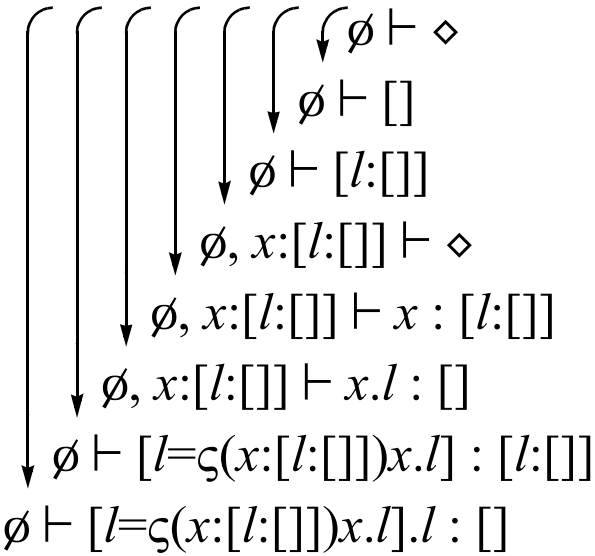
(Red Update)

$$\frac{\vdash a \rightsquigarrow [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}] \qquad j\in 1..n}{\vdash a.l_j \Leftarrow \varsigma(x:A)b \rightsquigarrow [l_j=\varsigma(x:A_j)b,\ l_i=\varsigma(x_i:A_i)b_i\ ^{i\in(1..n)-\{j\}}]}$$

A Typed Divergent Term

The first-order object calculus is not normalizing: there are typable terms whose evaluation does not terminate.

For example, the untyped term $[l{=}\varsigma(x)x.l].l$ can be annotated to obtain the typed term $[l{=}\varsigma(x{:}[l{:}[]])x.l].l$, which is typable as follows.



$\emptyset \vdash \diamond$ (Env $\emptyset$)

$\emptyset \vdash []$ (Type Object) with $n = 0$

$\emptyset \vdash [l{:}[]]$ (Type Object) with $n = 1$

$\emptyset, x{:}[l{:}[]] \vdash \diamond$ (Env x)

$\emptyset, x{:}[l{:}[]] \vdash x : [l{:}[]]$ (Val x)

$\emptyset, x{:}[l{:}[]] \vdash x.l : []$ (Val Select)

$\emptyset \vdash [l{=}\varsigma(x{:}[l{:}[]])x.l] : [l{:}[]]$ (Val Object) with $n = 1$

$\emptyset \vdash [l{=}\varsigma(x{:}[l{:}[]])x.l].l : []$ (Val Select)

(Val Object) enables us to assume that the self variable x has the type $[l{:}[]]$ when checking that the body $x.l$ of the method l has the type $[]$.

Typed Object-Oriented Booleans

Notation

- $x : A \triangleq a$ stands for $x \triangleq a$ and $E \vdash a : A$
  where E is determined from the preceding context.

We do not have a single type for our booleans; instead, we have a type $Bool_A$ for every type A.

$$Bool_A \triangleq [if : A, then : A, else : A]$$

$$true_A : Bool_A \triangleq$$
$$[if = \varsigma(x{:}Bool_A)\ x.then,$$
$$then = \varsigma(x{:}Bool_A)\ x.then,$$
$$else = \varsigma(x{:}Bool_A)\ x.else]$$

$$false_A : Bool_A \triangleq$$
$$[if = \varsigma(x{:}Bool_A)\ x.else, \ldots]$$

The terms of type $Bool_A$ can be used in conditional expressions whose result type is A.

For c and d of type A, and fresh variable x, we define:

$$if_A\ b\ then\ c\ else\ d : A \triangleq ((b.then \Leftarrow \varsigma(x{:}Bool_A)c).else \Leftarrow \varsigma(x{:}Bool_A)\ d).if$$

Moreover, we get some subtypings, *e.g.*:

$$[if : A,\ then : A,\ else : A] <: [if : A] <: []$$
$$[if : A,\ then : A,\ else : A] <: [else : A] <: []$$

Typed Cells

- We assume an imperative semantics (in order to postpone the use of recursive types).
- If *set* works by side-effect, its result type can be uninformative. (We can write *x.set*(3) ; *x.contents* instead of *x.set*(3).*contents*.)

Assuming a type *Nat* and function types, we let:

$$Cell \triangleq [contents : Nat, set : Nat \to []]$$
$$GCell \triangleq [contents : Nat, set : Nat \to [], get : Nat]$$

We get:

$$GCell <: Cell$$
$$cell \triangleq [contents = 0, set = \varsigma(x{:}Cell)\, \lambda(n{:}Nat)\, x.contents := n]$$

has type *Cell*

$$gcell \triangleq [..., get = \varsigma(x{:}GCell)\, x.contents]$$

has types *GCell* and *Cell*

Some Results

For the functional calculus (named $\mathbf{Ob}_{1<:}$):

Each well-typed term has a minimum type:

Theorem (Minimum types)

If $E \vdash a : A$ then there exists B such that $E \vdash a : B$ and,
for any A', if $E \vdash a : A'$ then $E \vdash B <: A'$.

The type system is sound for the operational semantics:

Theorem (Subject reduction)

If $\emptyset \vdash a : C$
and $\vdash a \rightsquigarrow v$
then $\emptyset \vdash v : C$.

Minimum Types

Because of subsumption, terms do not have unique types.

However, a weaker property holds: every term has a minimum type (if it has a type at all).

The minimum-types property is potentially useful for developing typechecking algorithms:

- It guarantees the existence of a "best" type for each typable term.
- Its proof suggests how to calculate this "best" type.

For proving the minimum-types property for $\mathbf{Ob}_{1<:}$, we consider a modified system ($\mathbf{MinOb}_{1<:}$) obtained by:

~ removing (Val Subsumption), and

~ modifying the (Val Object) and (Val Update) rules as follows:

Modified rules

(Val Min Object) (where $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$)

$$\frac{E, x_i{:}A \vdash b_i : B_i' \qquad \emptyset \vdash B_i' <: B_i \qquad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i{:}A) b_i\ ^{i\in 1..n}] : A}$$

(Val Min Update) (where $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$)

$$\frac{E \vdash a : A' \qquad \emptyset \vdash A' <: A \qquad E, x{:}A \vdash b : B_j' \qquad \emptyset \vdash B_j' <: B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A) b : A}$$

Typing in $\mathbf{MinOb}_{1<:}$ is unique, as we show next.

We can extract from $\mathbf{MinOb}_{1<:}$ a typechecking algorithm that, given any E and a, computes the type A such that $E \vdash a : A$ if one exists.

The next three propositions are proved by easy inductions on the derivations of $E \vdash a : A$ in $\mathbf{MinOb_{1<:}}$.

Proposition ($\mathbf{MinOb_{1<:}}$ typings are $\mathbf{Ob_{1<:}}$ typings)

If $E \vdash a : A$ is derivable in $\mathbf{MinOb_{1<:}}$,
then it is also derivable in $\mathbf{Ob_{1<:}}$.

□

Proposition ($\mathbf{MinOb_{1<:}}$ has unique types)

If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in $\mathbf{MinOb_{1<:}}$,
then $A \equiv A'$.

□

Proposition ($\mathbf{MinOb_{1<:}}$ has smaller types than $\mathbf{Ob_{1<:}}$)

If $E \vdash a : A$ is derivable in $\mathbf{Ob_{1<:}}$,
then $E \vdash a : A'$ is derivable in $\mathbf{MinOb_{1<:}}$ for some A' such that
$E \vdash A' <: A$ is derivable (in either system).

□

We obtain:

Proposition ($\mathbf{Ob_{1<:}}$ has minimum types)

In $\mathbf{Ob_{1<:}}$, if $E \vdash a : A$
then there exists B such that $E \vdash a : B$ and, for any A',
if $E \vdash a : A'$ then $E \vdash B <: A'$.

Proof

Assume $E \vdash a : A$. So $E \vdash a : B$ is derivable in $\mathbf{MinOb_{1<:}}$ for some B such that $E \vdash B <: A$.

Hence, $E \vdash a : B$ is also derivable in $\mathbf{Ob_{1<:}}$.

If $E \vdash a : A'$, then $E \vdash a : B'$ is also derivable in $\mathbf{MinOb_{1<:}}$ for some B' such that $E \vdash B' <: A'$.

Finally, $B \equiv B'$, so $E \vdash B <: A'$.

□

Lack of type annotations in ς-binders destroys the minimum-types property. For example, let:

$$A \equiv [l{:}[]]$$
$$A' \equiv [l{:}A]$$
$$a \equiv [l{=}\varsigma(x)[l{=}\varsigma(x)[]]]$$

then:

$$\emptyset \vdash a : A \quad \text{and} \quad \emptyset \vdash a : A'$$

but A and A' have no common subtype.

This example also shows that minimum typing is lost for objects with fields (where the ς-binders are omitted entirely).

The term $a.l{:=}[]$ typechecks using $\emptyset \vdash a : A$ but not using $\emptyset \vdash a : A'$.

Naive type inference algorithms might find the type A' for a, and fail to find any type for $a.l{:=}[]$. This poses problems for type inference.

(But cf. Palsberg's lectures.)

In contrast, with annotations, both

$$\emptyset \vdash [l=\varsigma(x:A)[l=\varsigma(x:A)[]]] : A$$

and

$$\emptyset \vdash [l=\varsigma(x:A')[l=\varsigma(x:A)[]]] : A'$$

are minimum typings.

The former typing can be used to construct a typing for $a.l:=[]$.

Subject Reduction

We start the proof with two standard lemmas.

Lemma (Bound weakening)

If $E, x{:}D, E' \vdash \mathfrak{I}$ and $E \vdash D' <: D$, then $E, x{:}D', E' \vdash \mathfrak{I}$.

□

Lemma (Substitution)

If $E, x{:}D, E' \vdash \mathfrak{I}\{x\}$ and $E \vdash d : D$, then $E, E' \vdash \mathfrak{I}\{\!\{d\}\!\}$.

□

Using these lemmas, we obtain:

Theorem (Subject reduction)

Let c be a closed term and v be a result, and assume $\vdash c \rightsquigarrow v$.

If $\emptyset \vdash c : C$, then $\emptyset \vdash v : C$.

Proof

The proof is by induction on the derivation of $\vdash c \rightsquigarrow v$.

Case (Red Object)

This case is trivial, since $c = v$.

Case (Red Select)

Suppose $\vdash a \rightsquigarrow [l_i = \varsigma(x_i{:}A_i)b_i\{x_i\}^{\,i\in 1..n}]$ and $\vdash b_j\{\!\![[l_i = \varsigma(x_i{:}A_i)b_i\{x_i\}^{\,i\in 1..n}]\!\!\} \rightsquigarrow v$ have yielded $\vdash a.l_j \rightsquigarrow v$.

Assume that $\emptyset \vdash a.l_j : C$.

This must have come from an application of (Val Select) with assumption $\emptyset \vdash a : A$ where A has the form $[l_j{:}B_j, \ldots]$, and with conclusion $\emptyset \vdash a.l_j : B_j$, followed by a number of subsumption steps implying $\emptyset \vdash B_j <: C$ by transitivity.

By induction hypothesis, we have $\emptyset \vdash [l_i = \varsigma(x_i{:}A_i)b_i\{x_i\}^{\,i\in 1..n}] : A$.

This implies that there exists A' such that $\emptyset \vdash A' <: A$, that all A_i equal A', that $\emptyset \vdash [l_i = \varsigma(x_i{:}A')b_i\{x_i\}^{\,i\in 1..n}] : A'$, and that $\emptyset, x_j{:}A' \vdash b_j : B_j$.

By a lemma, it follows that $\emptyset \vdash b_j\{\!\![[l_i = \varsigma(x_i{:}A')b_i\{x_i\}^{\,i\in 1..n}]\!\!\} : B_j$.

By induction hypothesis, we obtain $\emptyset \vdash v : B_j$ and, by subsumption, $\emptyset \vdash v : C$.

Case (Red Update)

Suppose $\vdash a \rightsquigarrow [l_i = \varsigma(x_i{:}A_i)b_i \ ^{i \in 1..n}]$ has yielded $\vdash a.l_j \Leftarrow \varsigma(x{:}A)b \rightsquigarrow [l_j = \varsigma(x{:}A_j)b, \ l_i = \varsigma(x_i{:}A_i)b_i \ ^{i \in (1..n)-\{j\}}]$.

Assume that $\emptyset \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : C$.

This must have come from an application of (Val Update) with assumptions $\emptyset \vdash a : A$ and $\emptyset, x{:}A \vdash b : B$ where A has the form $[l_j{:}B, \ldots]$, and with conclusion $\emptyset \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : A$, followed by a number of subsumption steps implying $\emptyset \vdash A <: C$ by transitivity.

By induction hypothesis, we have $\emptyset \vdash [l_i = \varsigma(x_i{:}A_i)b_i \ ^{i \in 1..n}] : A$.

This implies that A_j has the form $[l_j{:}B, \ l_i{:}B_i \ ^{i \in (1..n)-\{j\}}]$, that $\emptyset \vdash A_j <: A$, that A_i equals A_j, and that $\emptyset, x_i{:}A_j \vdash b_i : B_i$ for all i.

By a lemma, it follows that $\emptyset, x{:}A_j \vdash b : B$.

Therefore by (Val Object), $\emptyset \vdash [l_j = \varsigma(x{:}A_j)b, \ l_i = \varsigma(x_i{:}A_j)b_i \ ^{i \in (1..n)-\{j\}}] : A_j$.

We obtain $\emptyset \vdash [l_j = \varsigma(x{:}A_j)b, \ l_i = \varsigma(x_i{:}A_j)b_i \ ^{i \in (1..n)-\{j\}}] : C$ by subsumption.

□

The proof of subject reduction is simply a sanity check.

It is an easy proof, with just one subtle point: the proof would have failed if we had defined (Red Update) so that

$$\vdash a.l_j \Leftarrow \varsigma(x{:}A)b \rightsquigarrow [l_j{=}\varsigma(x{:}A)b,\ l_i{=}\varsigma(x_i{:}A_i)b_i\ ^{i\in(1..n)-\{j\}}]$$

with an A instead of an A_j in the bound for x.

Unsoundness of Covariance

Object types are **invariant** (not co/contravariant) in components.

$U \triangleq []$ The unit object type.

$L \triangleq [l{:}U]$ An object type with just l.

$L <: U$

$P \triangleq [x{:}U, f{:}U]$

$Q \triangleq [x{:}L, f{:}U]$

Assume $Q <: P$ by an (erroneous) covariant rule.

$q : Q \triangleq [x = [l{=}[]], f = \varsigma(s{:}Q)\ s.x.l]$

then $q : P$ by subsumption with $Q <: P$

hence $q.x{:=}[] : P$ that is $[x = [], f = \varsigma(s{:}Q)\ s.x.l] : P$

But $(q.x{:=}[]).f$ fails!

Classes, with Types

If $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ is an object type,
then $Class(A)$ is the type of the classes for objects of type A:

$$Class(A) \triangleq [new{:}A,\ l_i{:}A{\to}B_i\ ^{i\in 1..n}]$$

$new{:}A$ is a **generator** for objects of type A.
$l_i{:}A{\to}B_i$ is a **pre-method** for objects of type A.

$$c : Class(A) \triangleq$$
$$[new = \varsigma(z{:}Class(A))\ [l_i = \varsigma(x{:}A)\ z.l_i(x)\ ^{i\in 1..n}],$$
$$l_i = \lambda(x_i{:}A)\ b_i\{x_i\}\ ^{i\in 1..n}]$$
$$c.new : A$$

- Types are distinct from classes.
- More than one class may generate objects of a type.

Inheritance, with Types

Let $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ and $A' \equiv [l_i{:}B_i\ ^{i\in 1..n}, l_j{:}B_j\ ^{j\in n+1..m}]$, with $A' <: A$.

Note that $Class(A)$ and $Class(A')$ are not related by subtyping.

Let c: $Class(A)$, then for $i \in 1..n$

$$c.l_i: A \to B_i <: A' \to B_i.$$

Hence $c.l_i$ is a good pre-method for a class of type $Class(A')$.

We may define a subclass c' of c:

$$c' : Class(A') \triangleq [new=..., l_i=c.l_i\ ^{i\in 1..n}, l_j=...\ ^{j\in n+1..m}]$$

where class c' inherits the methods l_i from class c.

So inheritance typechecks:

If $A' <: A$ then a class for A' may inherit from a class for A.

Class Types for Cells

$$Class(Cell) \triangleq$$
$$[new : Cell,$$
$$contents : Cell \to Nat,$$
$$set : Cell \to Nat \to []]$$

$$Class(GCell) \triangleq$$
$$[new : GCell,$$
$$contents : GCell \to Nat,$$
$$set : GCell \to Nat \to [],$$
$$get : GCell \to Nat]$$

$Class(GCell) <: Class(Cell)$ does not hold, but inheritance is possible:

$$Cell \to Nat <: GCell \to Nat$$
$$Cell \to Nat \to [] <: GCell \to Nat \to []$$

Typed Reasoning

In addition to a type theory, we have a simple typed proof system.

There are some subtleties in reasoning about objects.

Consider:

$$\begin{array}{lcl} A & \triangleq & [x : Nat, f : Nat] \\ a : A & \triangleq & [x = 1, f = 1] \\ b : A & \triangleq & [x = 1, f = \varsigma(s{:}A)\ s.x] \end{array}$$

Informally, we may say that $a.x = b.x : Nat$ and $a.f = b.f : Nat$.

So, do we have $a = b$?

It would follow that $(a.x{:=}2).f = (b.x{:=}2).f$

and then $1 = 2$.

Hence:

$$a \nmid b : A$$

Still, as objects of $[x : Nat]$, a and b are indistinguishable from $[x = 1]$.

Hence:

$$a = b : [x : Nat]$$

Finally, we may ask:

$$a \stackrel{?}{=} b : [f : Nat]$$

This is sound; it can be proved via bisimilarity.

In summary, there is a notion of typed equality that may support some interesting transformations (inlining of methods).

Second-Order Object Calculi

Type Quantification

We add second-order type quantifiers to our first-order calculi.

- These quantifiers are standard.
- They are generally useful for polymorphism and data abstraction.
- However, these quantifiers have interesting interactions with object types.

The notations $b\{X\}$ and $B\{X\}$ show the free occurrences of X in b and in B, respectively.

$b\{\!\{A\}\!\}$ stands for $b\{\!\{X \leftarrow A\}\!\}$ and $B\{\!\{A\}\!\}$ stands for $B\{\!\{X \leftarrow A\}\!\}$ when X is clear from context.

The Universal Quantifier

- A term $\lambda(X)b\{X\}$ represents a term b parameterized with respect to a type variable X; this is a *type abstraction*.
- Correspondingly, a term $a(A)$ is the application of a term a to a type A; this is a *type application*.
- $b\{\!\{A\}\!\}$ is an instantiation of the type abstraction $\lambda(X)b\{X\}$ for a specific type A. It is the result of a type application $(\lambda(X)b\{X\})(A)$.
- $\forall(X)B\{X\}$ is the type of those type abstractions $\lambda(X)b\{X\}$ that for any type A produce a result $b\{\!\{A\}\!\}$ of type $B\{\!\{A\}\!\}$.

For example:

$id : \forall(X)\ X \to X \ \triangleq\ \lambda(X)\ \lambda(x{:}X)\ x$ the identity function

$id(Int) : Int \to Int$ its instantiation to the type *Int*

$id(Int)(3) : Int$ its application to an integer

(Env $X<:$)

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

(Type $X<:$)

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

(Sub X)

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

(Type All)

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

(Val Fun2)

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

(Val Appl2)

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

- E', X, E'' is an abbreviation for $E', X<:Top, E''$.
- (Type All) forms a quantified type $\forall(X)B$ in E, provided that B is well-formed in E extended with X.
- (Val Fun2) constructs a type abstraction $\lambda(X)b$ of type $\forall(X)B$, provided that the body b has type B for an arbitrary type parameter X (which may occur in b and B).

- (Val Appl2) applies such a type abstraction to a type A.

The Bounded Universal Quantifier

- We extend universally quantified types $\forall(X)B$ to bounded universally quantified types $\forall(X<:A)B$, where A is the bound on X.
- The bounded type abstraction $\lambda(X<:A)b\{X\}$ has type $\forall(X<:A)B\{X\}$ if, for any subtype A' of A, the instantiation $b\{\!\{A'\}\!\}$ has type $B\{\!\{A'\}\!\}$.

(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<:)

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

(Val Appl2<:)

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

Structural Update

While adding type variables and type quantifiers, we have not changed the rules for objects.

However, it is tempting to change the rule (Val Update) as follows:

(Val Structural Update) (where $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$)

$$\frac{E \vdash a : C \qquad E \vdash C <: A \qquad E, x{:}C \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}C)b : C}$$

The difference between (Val Update) and (Val Structural Update) can be seen when C is a type variable:

$$\lambda(C<:[l{:}Nat])\ \lambda(a{:}C)\ a.l{:=}3 \quad : \quad \forall(C<:[l{:}Nat])\ C \to [l{:}Nat]$$

via (Val Update)

$$\lambda(C<:[l{:}Nat])\ \lambda(a{:}C)\ a.l{:=}3 \quad : \quad \forall(C<:[l{:}Nat])\ C \to C$$

via (Val Structural Update)

The new rule (Val Structural Update) appears intuitively sound.

- It implicitly relies on the invariance of object types, and on the assumption that every subtype of an object type is an object type.
- Such an assumption is quite easily realized in programming languages, and holds in formal systems such as ours.

But this assumption is false in standard denotational semantics where the subtype relation is simply the subset relation.

- In such semantics, $\forall(C<:[l:Nat])\ C\to C$ contains only an identity function and its approximations.
- $\forall(C<:[l:Nat])\ C\to C$ does not contain $\lambda(C<:[l:Nat])\ \lambda(a:C)\ a.l:=3$.

This difficulty suggests that we should proceed with caution.

Hence we do not adopt the rule (Val Structural Update) at once (but we will later on).

The Bounded Existential Quantifier

- The existentially quantified type $\exists(X<:A)B\{X\}$ is the type of the pairs $\langle A',b\rangle$ where A' is a subtype of A and b is a term of type $B\{\!\{A'\}\!\}$.
  - ~ The type $\exists(X<:A)B\{X\}$ can be seen as a partially abstract data type with *interface* $B\{X\}$ and with *representation type* X known only to be a subtype of A.
  - ~ It is partially abstract in that it gives some information about the representation type, namely, a bound.
- The pair $\langle A',b\rangle$ describes an element of the partially abstract data type with representation type A' and *implementation* b.

In order to be fully explicit, we write the pair $\langle A', b\rangle$ more verbosely:

$$pack\ X<:A=A'\ with\ b\{X\}:B\{X\}$$

where $X<:A=A'$ indicates that $X<:A$ and $X=A'$.

An element c of type $\exists(X<:A)B\{X\}$ can be used in the construct:

$$open\ c\ as\ X<:A,x:B\{X\}\ in\ d\{X,x\}:D$$

where

- d has access to the representation type X and the implementation x of c;
- d produces a result of a type D that does not depend on X.

At evaluation time, if c is $\langle A', b\rangle$, then the result is $d\{\!\{A', b\}\!\}$ of type D.

For example, we may write:

$$p : \exists(X{<:}Int)X{\times}(X{\to}X) \triangleq$$
$$\quad pack\ X{<:}Int{=}Nat\ with\ \langle 0, succ_{Nat}\rangle : X{\times}(X{\to}X)$$
$$a : Int \triangleq$$
$$\quad open\ p\ as\ X{<:}Int, x{:}X{\times}(X{\to}X)\ in\ snd(x)(fst(x)){:}Int$$

and then $a = 1$.

(Type Exists<:)

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

(Sub Exists)

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

(Val Pack<:)

$$\frac{E \vdash C <: A \qquad E \vdash b\{\!\{C\}\!\} : B\{\!\{C\}\!\}}{E \vdash \mathit{pack}\ X{<:}A{=}C\ \mathit{with}\ b\{X\}{:}B\{X\} : \exists(X{<:}A)B\{X\}}$$

(Val Open<:)

$$\frac{E \vdash c : \exists(X{<:}A)B \qquad E \vdash D \qquad E, X{<:}A, x{:}B \vdash d : D}{E \vdash \mathit{open}\ c\ \mathit{as}\ X{<:}A{,}x{:}B\ \mathit{in}\ d{:}D : D}$$

To Make a Long Story Short

We added quantifiers to our object calculus.

We extended our results and examples:

- semantics,
- encodings of functions,
- representation of classes,
- ...

We developed some ideas:

- for the treatment of Self types,
- for the representation of objects in terms of functions.

Translations

Objects vs. Procedures

- Object-oriented programming languages have introduced (or popularized) a number of ideas and techniques.
- However, on a case-by-case basis, one can often emulate objects in some procedural languages.

  Are object-oriented concepts reducible to procedural concepts?

  ~ It is easy to emulate the operational semantics of objects.

  ~ It is a little harder to translate object types.

  ~ It is much harder, or impossible, to preserve subtyping.

  ~ Apparently, this reduction is not feasible or attractive in practice.

The Translation Problem

- The problem is to find a translation from an object calculus to a λ-calculus:
  - ~ The object calculus should be reasonably expressive.
  - ~ The λ-calculus should be standard enough.
  - ~ The translation should be faithful; in particular it should preserve sub-typing.

  We prefer to deal with calculi rather than programming languages.
- The goal of explaining objects in terms of λ-calculi is not new.
  - ~ There have been a number of more or less successful attempts (by Kamin, Cardelli, Cook, Reddy, Mitchell, the John Hopkins group, Pierce, Turner, Hofmann, Remy, Bruce, ...).
  - ~ We will review some of them (fairly informally), and then see our translations (joint work with Ramesh Viswanathan.)

The Self-Application Semantics

- The self-application interpretation maps an object to a records of functions.
- On method invocation, the whole object is passed to the method as a parameter.

Untyped self-application interpretation

$$[l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}] \triangleq \langle l_i{=}\lambda(x_i)b_i\ ^{i\in 1..n}\rangle \qquad (l_i \text{ distinct})$$

$$o.l_j \triangleq o\cdot l_j(o) \qquad (j\in 1..n)$$

$$o.l_j{\Leftarrow}\varsigma(y)b \triangleq o\cdot l_j{:=}\lambda(y)b \qquad (j\in 1..n)$$

The Self-Application Semantics (Typed)

A typed version is obtained by representing object types as recursive record types:

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \mu(X)\langle l_i{:}X\to B_i\ ^{i\in 1..n}\rangle$$

Self-application interpretation

$$A \equiv [l_i{:}B_i\ ^{i\in 1..n}] \triangleq \mu(X)\langle l_i{:}X\to B_i\ ^{i\in 1..n}\rangle \qquad (l_i \text{ distinct})$$

$$[l_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}] \triangleq fold(A,\langle l_i{=}\lambda(x_i{:}A)b_i\ ^{i\in 1..n}\rangle)$$

$$o.l_j \triangleq unfold(o)\cdot l_j(o) \qquad (j\in 1..n)$$

$$o.l_j{\Leftarrow}\varsigma(y{:}A)b \triangleq fold(A,unfold(o)\cdot l_j{:=}\lambda(y{:}A)b) \qquad (j\in 1..n)$$

Unfortunately, the subtyping rule for object types fails to hold:
a contravariant X occurs in all method types.

The State-Application Semantics (Sketch)

For systems with only field update, it is natural to separate fields and methods:

- The fields are grouped into a state record *st*, separate from the method suite record *mt*.
- Methods receive *st* as a parameter on method invocation, instead of the whole object as in the self-application interpretation.
- The update operation modifies the *st* component and copies the *mt* component.
- The method suite is bound recursively with a μ, so that each method can invoke the others.

Untyped state-application interpretation

$[f_k{=}b_k{}^{k\in 1..m} \mid l_i{=}\varsigma(x_i)b_i{}^{i\in 1..n}] \triangleq$	(f_k, l_i distinct)
$\langle st{=}\langle f_k{=}b_k{}^{k\in 1..m}\rangle, mt{=}\mu(m)\langle l_i{=}\lambda(s)b_i{}'{}^{i\in 1..n}\rangle\rangle$	(for appropriate $b_i{}'$)
$o_{\square}f_j \triangleq o{\cdot}st{\cdot}f_j$	($j\in 1..m$) (external)
$o_{\square}f_j{:=}b \triangleq \langle st{=}(o{\cdot}st{\cdot}f_j{:=}b), mt{=}o{\cdot}mt\rangle$	($j\in 1..m$) (external)
$o.l_j \triangleq o{\cdot}mt{\cdot}l_j(o{\cdot}st)$	($j\in 1..n$) (external)

It is difficult to express the precise translation of method bodies (b_i).

Although it is fairly clear how to translate specific examples, it is hard to define a general interpretation, particularly without types.

Essentially this difficulty arises because self is split into two parts.

- Internal operations manipulate s directly, and are thus coded differently from external operations.
- Since the self parameter s gives access only to fields, internal method invocation is done through m.
- Methods that return self should produce a whole object, but s contains only fields, so a whole object must be regenerated.

Untyped state-application interpretation (continued)

in the context $\mu(m)\langle l_i=\lambda(s) \ldots \rangle$

$x_i{\scriptstyle\square}f_j \triangleq s\cdot f_j$ $(j\in 1..m)$ (internal)

$x_i{\scriptstyle\square}f_j:=b \triangleq s\cdot f_j:=b$ $(j\in 1..m)$ (internal)

$x_i.l_j \triangleq m\cdot l_j(s)$ $(j\in 1..n)$ (internal)

The State-Application Semantics (Typed)

The state of an object, represented by a collection of fields *st*, is hidden by existential abstraction, so external updates are not possible.

The troublesome method argument types are hidden as well, so this interpretation yields the desired subtypings.

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \exists(X)\,\langle st{:}\,X,\, mt{:}\,\langle l_i{:}X{\to}B_i\ ^{i\in 1..n}\rangle\rangle$$

- In general case, code generation is driven by types.
- The encoding is rather laborious.
- Still, it accounts well for class-based languages where methods are separate from fields, and method update is usually forbidden.

State-application interpretation

$A \equiv [l_i{:}B_i\ ^{i\in 1..n}] \triangleq$ (l_i distinct)

$\exists(X)\ C\{X\}$ where $C\{X\} \equiv \langle st{:}\ X,\ mt{:}\ \langle l_i{:}X{\to}B_i\ ^{i\in 1..n}\rangle\rangle$

$[f_k{=}b_k\ ^{k\in 1..m} \mid l_i{=}\varsigma(x_i{:}A)b_i\{x_i\}\ ^{i\in 1..n}] \triangleq$ (f_k, l_i distinct)

pack $X{=}\langle f_k{:}B_k\ ^{k\in 1..m}\rangle$

with $\langle st{=}\langle f_k{=}b_k\ ^{k\in 1..m}\rangle,$

$mt{=}\mu(m{:}\langle l_i{:}X{\to}B_i\ ^{i\in 1..n}\rangle)\ \langle l_i{=}\lambda(s{:}X)b_i{'}\ ^{i\in 1..n}\rangle\rangle$ (for appropriate $b_i{'}$)

$: C\{X\}$

$x_i{\scriptstyle\square}f_j \triangleq s{\cdot}f_j$ ($j\in 1..m$) (internal)

$x_i{\scriptstyle\square}f_j{:=}b \triangleq s{\cdot}f_j{:=}b$ ($j\in 1..m$) (internal)

$x_i.l_j \triangleq m{\cdot}l_j(s)$ ($j\in 1..n$) (internal)

$o.l_j \triangleq$ *open* o *as* $X, p{:}C\{X\}$ *in* $p{\cdot}mt{\cdot}l_j(p{\cdot}st) : B_j$ ($j\in 1..n$) (external)

The Recursive-Record Semantics (Example)

This interpretation is often used to code objects within λ-calculi, for specific examples.

A typical application concerns movable color points:

$$CPoint \triangleq Obj(X)[x{:}Int,\ c{:}Color \mid mv{:}Int \to X]$$

$$cPoint : CPoint \triangleq [x = 0,\ c = black \mid mv = \varsigma(s{:}CPoint)\ \lambda(dx{:}Int)\ s.x{:=}s.x{+}dx]$$

(Here X is the type of self, that is, the Self type of *CPoint*.)

The translation is:

$$CPoint \triangleq$$
$$\mu(X)\langle x{:}Int,\ c{:}Color,\ mv{:}Int\to X\rangle$$
$$cPoint : CPoint \triangleq$$
$$let\ rec\ init(x0{:}Int,\ c0{:}Color) =$$
$$\mu(s{:}CPoint)\ fold(CPoint,$$
$$\langle x = x0,\ c = c0,$$
$$mv = \lambda(dx{:}Int)\ init(unfold(s)\cdot x+dx,\ unfold(s)\cdot c)\rangle)$$
$$in\ init(0,\ black)$$

- An auxiliary function *init* is used both for field initialization and for the creation of modified objects during update.
- Only internal field update is handled correctly.
- This translation achieves the desired effect, yielding the expected behavior for *cPoint* and the expected subtypings for *CPoint*.
- If the code for *mv* had been $\lambda(dx{:}Int)\ f(s)\cdot x := s\cdot x+dx$, where *f* is of appropriate type, it would not have been clear how to proceed.

The Split-Methods Semantics

Untyped split-method interpretation

$$[l_i = \varsigma(x_i)b_i^{\,i\in 1..n}] \triangleq \qquad (l_i \text{ distinct})$$

$$\textit{let rec } \textit{create}(y_i^{\,i\in 1..n}) =$$

$$\langle l_i^{sel} = y_i,$$

$$l_i^{upd} = \lambda(y_i')\, \textit{create}(y_j^{\,j\in 1..i-1}, y_i', y_k^{\,k\in i+1..n})^{\,i\in 1..n}\rangle$$

$$\textit{in } \textit{create}(\lambda(x_i)b_i^{\,i\in 1..n})$$

$$o.l_j \triangleq o \cdot l_j^{sel}(o) \qquad (j\in 1..n)$$

$$o.l_j \Leftarrow \varsigma(y)b \triangleq o \cdot l_j^{upd}(\lambda(y)b) \qquad (j\in 1..n)$$

- A method l_j is represented by two record components, l_j^{sel} and l_j^{upd}.
- *create* takes a collection of functions and produces a record. The uses of *create* are encapsulated within the definition of *create*.
- A method l_j is updated by supplying the new code for l_j to the function l_j^{upd}. This code is passed on to *create*.
- A method l_j is invoked by applying the function l_j^{sel} to o.

The Split-Method Semantics (Typed)

- A first attempt at typing this interpretation could be to set:

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \mu(X)\ \langle l_i^{sel}{:}X\to B_i\ ^{i\in 1..n},\ l_i^{upd}{:}(X\to B_i)\to X\ ^{i\in 1..n}\rangle$$

  but this type contains contravariant occurrences of X.
  Subtypings fail.

- As a second attempt, we can use quantifiers to obtain covariance:

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq$$
$$\mu(Y)\ \exists(X{<:}Y)\ \langle l_i^{sel}{:}X\to B_i\ ^{i\in 1..n},\ l_i^{upd}{:}(X\to B_i)\to X\ ^{i\in 1..n}\rangle$$

  ~ Now the interpretation validates the subtypings for object types, since all occurrences of X, bound by $\exists$, are covariant.

  ~ Unfortunately, it is impossible to perform method invocations: after opening the $\exists$ we do not have an appropriate argument of type X to pass to l_i^{sel}.

~ But since this argument should be the object itself, we can solve the problem by adding a record component, r, bound recursively to the object:

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \mu(Y)\,\exists(X{<:}Y)\,\langle r{:}X,\ l_i^{sel}{:}X{\to}B_i\ ^{i\in 1..n},\ l_i^{upd}{:}(X{\to}B_i){\to}X\ ^{i\in 1..n}\rangle$$

Split-method interpretation

$$A \equiv [l_i{:}B_i\ ^{i\in 1..n}] \triangleq \mu(Y)\exists(X{<:}Y)C\{X\} \qquad (l_i \text{ distinct})$$

where

$$C\{X\} \equiv \langle r{:}X,\ l_i^{sel}{:}X{\to}B_i\ ^{i\in 1..n},\ l_i^{upd}{:}(X{\to}B_i){\to}X\ ^{i\in 1..n}\rangle$$

$$[l_i = \varsigma(x_i{:}A)b_i \ ^{i\in 1..n}] \triangleq$$

$$\begin{array}{l}
\quad \mathit{let\ rec\ create}(y_i{:}A\to B_i \ ^{i\in 1..n}){:}A = \\
\qquad \mathit{fold}(A, \\
\qquad\quad \mathit{pack}\ X{=}A \\
\qquad\quad \mathit{with} \\
\qquad\qquad \langle r{=}\mathit{create}(y_i \ ^{i\in 1..n}), \\
\qquad\qquad\ l_i^{sel}{=}y_i \ ^{i\in 1..n}, \\
\qquad\qquad\ l_i^{upd}{=}\lambda(y_i'{:}A\to B_i)\mathit{create}(y_j \ ^{j\in 1..i-1}, y_i', y_k \ ^{k\in i+1..n}) \ ^{i\in 1..n}\rangle \\
\qquad\quad : C\{X\}) \\
\quad \mathit{in\ create}(\lambda(x_i{:}A)b_i \ ^{i\in 1..n})
\end{array}$$

$$o_A.l_j \triangleq \qquad (j\in 1..n)$$

$$\begin{array}{l}
\quad \mathit{open\ unfold}(o)\ \mathit{as}\ X{<:}A, p{:}C\{X\} \\
\quad \mathit{in}\ p\cdot l_j^{sel}(p\cdot r) : B_j
\end{array}$$

$$o.l_j \Leftarrow \varsigma(y{:}A)b \triangleq \qquad (j\in 1..n)$$

$$\begin{array}{l}
\quad \mathit{open\ unfold}(o)\ \mathit{as}\ X{<:}A, p{:}C\{X\} \\
\quad \mathit{in}\ p\cdot l_j^{upd}(\lambda(y{:}A)b) : A
\end{array}$$

- We obtain both the expected semantics and the expected subtyping properties.
- The definition of the interpretation is syntax-directed.
- The interpretation covers all of the first-order object calculus (including method update).
- It extends naturally to other constructs:
  - ~ variance annotations,
  - ~ Self types (with some twists),
  - ~ a limited form of method extraction
    (but in general method extraction is unsound),
  - ~ imperative update,
  - ~ imperative cloning.
- It suggests principles for reasoning about objects.

An Imperative Version

For an imperative split-method interpretation, it is not necessary to split methods, because updates can be handled imperatively.

The imperative version correctly deals with a cloning construct.

$$[f_k{:}B_k{}^{k\in 1..m} \mid l_i{:}B_i{}^{i\in 1..n}] \triangleq$$
$$\mu(Y)\,\exists(X{<:}Y)\,\langle r{:}X, f_k{:}B_k{}^{k\in 1..m}, l_i{:}X\to B_i{}^{i\in 1..n}, cl{:}\langle\rangle\to X\rangle$$

Imperative self-application interpretation

$$A \equiv [f_k{:}B_k{}^{k\in 1..m} \mid l_i{:}B_i{}^{i\in 1..n}] \triangleq \qquad (f_k,\ l_i \text{ distinct})$$
$$\mu(Y)\exists(X{<:}Y)C\{X\}$$
with
$$C\{X\} \equiv \langle r{:}X, f_k{:}B_k{}^{k\in 1..m}, l_i{:}X\to B_i{}^{i\in 1..n}, cl{:}\langle\rangle\to X\rangle$$

$$[f_k{=}b_k\,^{k\in 1..m} \mid l_i{=}\varsigma(x_i{:}A)b_i\,^{i\in 1..n}] \triangleq$$

$$\begin{array}{l}
\quad \mathit{let\ rec\ create}(y_k{:}B_k\,^{k\in 1..m}, y_i{:}A{\to}B_i\,^{i\in 1..n}){:}A = \\
\qquad \mathit{let}\ z{:}C\{\!\{A\}\!\} = \langle r{=}\mathit{nil}(A), f_k{=}y_k\,^{k\in 1..m}, l_i{=}y_i\,^{i\in 1..n}, \mathit{cl}{=}\mathit{nil}(\langle\rangle{\to}A)\rangle \\
\qquad \mathit{in}\ \ z{\cdot}r{:=}\mathit{fold}(A,\mathit{pack}\ X{<:}A{=}A\ \mathit{with}\ z : C\{\!\{X\}\!\}); \\
\qquad\quad z{\cdot}\mathit{cl}{:=}\lambda(x{:}\langle\rangle)\mathit{create}(z{\cdot}f_k\,^{k\in 1..m}, z{\cdot}l_i\,^{i\in 1..n}); \\
\qquad\quad z{\cdot}r \\
\quad \mathit{in}\ \mathit{create}(b_k\,^{k\in 1..m}, \lambda(x_i{:}A)b_i\,^{i\in 1..n})
\end{array}$$

$$o_{A\square}f_j \triangleq \mathit{open}\ \mathit{unfold}(o)\ \mathit{as}\ X{<:}A, p{:}C\{X\}\ \mathit{in}\ p{\cdot}f_j : B_j \qquad (j\in 1..m)$$

$$o_{A\square}f_j{:=}b \triangleq \qquad (j\in 1..m)$$

$$\begin{array}{l}
\quad \mathit{open}\ \mathit{unfold}(o)\ \mathit{as}\ X{<:}A, p{:}C\{X\} \\
\quad \mathit{in}\ \mathit{fold}(A,\mathit{pack}\ X'{<:}X{=}X\ \mathit{with}\ p{\cdot}f_j{:=}b : C\{\!\{X'\}\!\}) : A
\end{array}$$

$$o_A.l_j \triangleq \qquad (j\in 1..n)$$

$$\begin{array}{l}
\quad \mathit{open}\ \mathit{unfold}(o)\ \mathit{as}\ X{<:}A, p{:}C\{X\} \\
\quad \mathit{in}\ p{\cdot}l_j(p{\cdot}r) : B_j
\end{array}$$

$o.l_j \Leftarrow \varsigma(x{:}A)b \quad \triangleq \qquad (j \in 1..n)$

open unfold(o) *as* $X{<:}A, p{:}C\{X\}$

in fold$(A$,*pack* $X'{<:}X{=}X$

with $p \cdot l_j := \lambda(x{:}A)b : C\{\!\{X'\}\!\}) : A$

$clone(o_A) \quad \triangleq \qquad (j \in 1..n)$

open unfold(o) *as* $X{<:}A, p{:}C\{X\}$

in $p \cdot cl(\langle\rangle) : A$

Conclusion to the Encodings

In our interpretations:

- Objects are records of functions, after all.
- Object types combine recursive types and existential types (with a recursion going through a bound!).

- The interpretations are direct and general enough to explain objects.
- But they are elaborate, and perhaps not definitive, and hence not a replacement for primitive objects.

Self Types

Self Types

We now axiomatize Self types directly, taking Self as primitive.

In order to obtain a flexible type system, we need constructions that provide both covariance and contravariance.

- Both variances are necessary to define function types.

There are several possible choices at this point.

- One choice would be to take invariant object types plus the two bounded second-order quantifiers.
- Instead, we prefer to use variance annotations for object types.
  This choice is sensible because it increases expressiveness, delays the need to use quantifiers, and is relatively simple.

Object Types and Self

We consider object types with Self of the form:

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

where $B\{X^+\}$ indicates that X occurs only covariantly in B

Obj binds a type variable X, which represents the Self type (the type of self), as in $Cell \triangleq Obj(X)[contents^{o} : Nat, set^{o} : Nat \rightarrow X]$.

Each υ_i (a variance annotation) is one of $^{-}$, ^{o}, and $^{+}$, for contravariance, invariance, and covariance, respectively.

- Invariant components are the familiar ones. They can be regarded, by subtyping, as either covariant or contravariant.
- Covariant components allow covariant subtyping, but prevent updating.
- Symmetrically, contravariant components allow contravariant subtyping, but prevent invocation.

Syntax of types

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

Variant occurrences

$Y\{X^+\}$	whether $X = Y$ or $X \mid Y$
$Top\{X^+\}$	always
$Obj(Y)[l_i \upsilon_i : B_i \ ^{i \in 1..n}]\{X^+\}$	if $X = Y$ or for all $i \in 1..n$: if $\upsilon_i \equiv {}^+$, then $B_i\{X^+\}$ if $\upsilon_i \equiv {}^-$, then $B_i\{X^-\}$ if $\upsilon_i \equiv {}^o$, then $X \notin FV(B_i)$
$Y\{X^-\}$	if $X \mid Y$
$Top\{X^-\}$	always
$Obj(Y)[l_i \upsilon_i : B_i \ ^{i \in 1..n}]\{X^-\}$	if $X = Y$ or for all $i \in 1..n$: if $\upsilon_i \equiv {}^+$, then $B_i\{X^-\}$ if $\upsilon_i \equiv {}^-$, then $B_i\{X^+\}$ if $\upsilon_i \equiv {}^o$, then $X \notin FV(B_i)$
$A\{X^o\}$	if neither $A\{X^+\}$ nor $A\{X^-\}$

Terms with Self

Syntax of terms

a,b ::=	terms
x	variable
$obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i\ ^{i\in 1..n}]$	object (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow (Y{<:}A, y{:}Y)\varsigma(x{:}Y)b$	method update

An object has the form $obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i\ ^{i\in 1..n}]$, where A is the chosen implementation of the Self type.

Variance information for this object is given as part of the type A.

All the variables x_i have type X (so the syntax is redundant).

Method Update and Self

Method update is written $a.l \Leftarrow (Y<:A, y:Y)\varsigma(x:Y)b$, where

- a has type A,
- Y denotes the unknown Self type of a,
- y denotes the *old self* (a), and
- x denotes self (at the time the updating method is invoked).

To understand the necessity of the parameter y, consider the case where the method body b has result type Y.

- This method body cannot return an arbitrary object of type A, because the type A may not be the true Self type of a.
- Since a itself has the true Self type, the method could soundly return it.
- But the typing does not work because a has type A rather than Y.
- To allow a to be returned, it is bound to y with type Y.

Abbreviations

$[l_i \upsilon_i{:}B_i\ ^{i \in 1..n}] \triangleq Obj(X)[l_i \upsilon_i{:}B_i\ ^{i \in 1..n}]$ for $X \notin FV(B_i)$, $i \in 1..n$

$[l_i{:}B_i\ ^{i \in 1..n}] \triangleq Obj(X)[l_i^{\circ}{:}B_i\ ^{i \in 1..n}]$ for $X \notin FV(B_i)$, $i \in 1..n$

$[l_i{=}\varsigma(x_i{:}A)b_i\ ^{i \in 1..n}] \triangleq$ for $X \notin FV(b_i)$, $i \in 1..n$

$obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i\ ^{i \in 1..n}]$

$a.l_j \Leftarrow \varsigma(x{:}A)b \triangleq a.l_j \Leftarrow (Y{<:}A, y{:}Y)\varsigma(x{:}Y)b$ for $Y, y \notin FV(b)$

Cells

$$Cell \triangleq Obj(X)[contents : Nat, set : Nat \to X]$$

$$cell : Cell \triangleq$$
$$[contents = 0,$$
$$\ set = \varsigma(x{:}Cell)\ \lambda(n{:}Nat)\ x.contents := n]$$
$$\equiv\ obj(X{=}Cell)$$
$$[contents = \varsigma(x{:}X)\ 0,$$
$$\ set = \varsigma(x{:}X)\ \lambda(n{:}Nat)\ x.contents \Leftarrow (Y{<:}X, y{:}Y)\ \varsigma(z{:}Y)\ n]$$

$$GCell \triangleq Obj(X)[contents : Nat, set : Nat \to X, get : Nat]$$

$$GCell <: Cell$$

Cells with Undo

A difficulty arises when trying to update fields of type Self.

This difficulty is avoided by using the old-self parameter.

$$UnCell \triangleq Obj(X)[contents : Nat, set : Nat \to X, undo : X]$$

$$uncell : UnCell \triangleq$$

$$obj(X{=}UnCell)$$

$$[contents = \varsigma(x{:}X)\ 0,$$

$$set = \varsigma(x{:}X)\ \lambda(n{:}Nat)$$

$$(x.undo \Leftarrow (Y{<:}X, y{:}Y)\ \varsigma(z{:}Y)\ y)$$

$$.contents \Leftarrow (Y{<:}X, y{:}Y)\ \varsigma(z{:}Y)\ n,$$

$$undo = \varsigma(x{:}X)\ x]$$

The use of *y* in the update of *undo* is essential.

Operational Semantics

The operational semantics is given in terms of a reduction judgment, $\vdash a \rightsquigarrow v$.

The results are objects of the form $obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i\ ^{i\in 1..n}]$.

Operational semantics

(Red Object) (where $v \equiv obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i\ ^{i\in 1..n}]$)

$$\frac{}{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i\{X,x_i\}\ ^{i\in 1..n}]$)

$$\frac{\vdash a \rightsquigarrow v' \qquad \vdash b_j\{\!\{A,v'\}\!\} \rightsquigarrow v \qquad j\in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Update) (where $v \equiv obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i\ ^{i\in 1..n}]$)

$$\frac{\vdash a \rightsquigarrow v \qquad j\in 1..n}{\vdash a.l_j \Leftarrow (Y{<:}A',y{:}Y)\varsigma(x{:}Y)b\{Y,y\} \rightsquigarrow obj(X{=}A)[l_j{=}\varsigma(x{:}X)b\{\!\{X,v\}\!\},\ l_i{=}\varsigma(x_i{:}X)b_i\ ^{i\in 1..n-\{j\}}]}$$

Type Rules for Self

Judgments

$E \vdash \diamond$	well-formed environment judgment
$E \vdash A$	type judgment
$E \vdash A <: B$	subtyping judgment
$E \vdash \upsilon A <: \upsilon' B$	subtyping judgment with variance
$E \vdash a : A$	value typing judgment

The rules for the judgments $E \vdash \diamond$, $E \vdash A$, and $E \vdash A <: B$ are standard, except of course for the new rules for object types.

Environments, types, and subtypes

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

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

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

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

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

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

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

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

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

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

(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) $\dfrac{E \vdash B}{E \vdash {}^o B <: {}^o B}$

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

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

- The formation rule for object types (Type Object) requires that all the component types be covariant in Self.
- The subtyping rule for object types (Sub Object) says, to a first approximation, that a longer object type A on the left is a subtype of a shorter object type A' on the right.
  - ~ Because of variance annotations, we use an auxiliary judgment and auxiliary rules.
- The type $Obj(X)[\ldots]$ can be seen as an alternative to the recursive type $\mu(X)[\ldots]$, but with differences in subtyping.
  - ~ (Sub Object), with all components invariant, reads:

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

  - ~ An analogous property fails with μ instead of Obj.

Terms with typing annotations

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

$$\frac{E, x_i{:}A \vdash b_i\{\!|A|\!\} : B_i\{\!|A|\!\} \qquad \forall i \in 1..n}{E \vdash obj(X{=}A)[l_i = \varsigma(x_i{:}X) b_i\{X\}^{\,i \in 1..n}] : A}$$

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

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad E, Y{<:}A, y{:}Y, x{:}Y \vdash b : B_j\{\!|Y|\!\} \qquad \upsilon_j \in \{^{o},^{-}\} \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow (Y{<:}A, y{:}Y)\varsigma(x{:}Y) b : A}$$

- (Val Object) can be used for building an object of a type A from code for its methods.
  - ~ In that code, the variable X refers to the Self type; in checking the code, X is replaced with A, and self is assumed of type A.
  - ~ Thus the object is built with knowledge that Self is A.
- (Val Select) treats method invocation, replacing the Self type X with a known type A for the object a whose method is invoked.
  - ~ The type A might not be the true type of a.
  - ~ The result type is obtained by examining a supertype A' of A.
- (Val Update) requires that an updating method work with a partially unknown Self type Y, which is assumed to be a subtype of a type A of the object a being modified.
  - ~ The updating method must be "parametric in Self": it must return self, the old self, or a modification of these.
  - ~ The result type is obtained by examining a supertype A' of A.

(Val Select) and (Val Update) rely on the structural assumption that every subtype of an object type is an object type.

In order to understand them, it is useful to compare them with the following more obvious alternatives:

(Val Non-Structural Select) (where $A \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i \in 1..n}]$)

$$\frac{E \vdash a : A \qquad \upsilon_j \in \{^{o}, ^{+}\} \qquad j \in 1..n}{E \vdash a.l_j : B_j\{\!\{A\}\!\}}$$

(Val Non-Structural Update) (where $A \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{\,i \in 1..n}]$)

$$\frac{E \vdash a : A \qquad E, Y<:A, y:Y, x:Y \vdash b : B_j\{\!\{Y\}\!\} \qquad \upsilon_j \in \{^{o}, ^{-}\} \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow (Y<:A, y:Y)\varsigma(x:Y)b : A}$$

These are special cases of (Val Select) and (Val Update) for $A \equiv A'$.

(Val Select) and (Val Update) are more general in that they allow A to be a variable.

Adding the Universal Quantifier

Syntax of type parameterization

A,B ::=	types
…	(as before)
$\forall(X<:A)B$	bounded universal type
a,b ::=	terms
…	(as before)
$\lambda(X<:A)b$	type abstraction
$a(A)$	type application

We add two rules to the operational semantics.

- According to these rules, evaluation stops at type abstractions and is triggered again by type applications.
- We let a type abstraction $\lambda(X<:A)b$ be a result.

Operational semantics for type parameterization

(Red Fun2) (where $v \equiv \lambda(X<:A)b$)

$$\frac{}{\vdash v \rightsquigarrow v}$$

(Red Appl2)

$$\frac{\vdash b \rightsquigarrow \lambda(X<:A)c\{X\} \qquad \vdash c\{\!\{A'\}\!\} \rightsquigarrow v}{\vdash b(A') \rightsquigarrow v}$$

Quantifier 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<:)

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

(Val Appl2<:)

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

Variant occurrences for quantifiers

$(\forall(Y<:A)B)\{X^+\}$ if $X = Y$ or both $A\{X^-\}$ and $B\{X^+\}$

$(\forall(Y<:A)B)\{X^-\}$ if $X = Y$ or both $A\{X^+\}$ and $B\{X^-\}$

Theorem (Subject reduction)

If $\emptyset \vdash a : A$ and $\vdash a \rightsquigarrow v$, then $\emptyset \vdash v : A$.

Classes and Self

As before, we associate a class type $Class(A)$ with each object type A.

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

$$Class(A) \triangleq [new:A, l_i:\forall(X<:A)X \rightarrow B_i\{X\}^{\,i \in 1..n}]$$

$$c : Class(A) \triangleq [new=\varsigma(z:Class(A))\, obj(X=A)[l_i=\varsigma(s:X)z.l_i(X)(s)^{\,i \in 1..n}], l_i = \lambda(Self<:A)\,\lambda(s:Self)\,\ldots^{\,i \in 1..n}]$$

Now pre-methods have polymorphic types.

For example:

$$
\begin{array}{l}
Class(Cell) \triangleq \\
\quad [new : Cell, \\
\quad\ contents : \forall(Self{<:}Cell)\ Self \to Nat, \\
\quad\ set : \forall(Self{<:}Cell)\ Self \to Nat \to Self\ ]
\end{array}
$$

$$
\begin{array}{l}
cellClass : Class(Cell) \triangleq \\
\quad [new = \varsigma(z{:}Class(Cell))\ obj(Self{=}Cell) \\
\qquad [contents = \varsigma(s{:}Self\ )\ z.contents(Self\ )(s), \\
\qquad\ set = \varsigma(s{:}Self\ )\ z.set(Self\ )(s)], \\
\quad\ contents = \lambda(Self{<:}Cell)\ \lambda(s{:}Self\ )\ 0, \\
\quad\ set = \lambda(Self{<:}Cell)\ \lambda(s{:}Self\ )\ \lambda(n{:}Nat)\ s.contents := n]
\end{array}
$$

Inheritance and Self

We can now reconsider the inheritance relation between classes.

Suppose that we have $A' <: A$:

$$A' \equiv Obj(X)[l_i \upsilon_i' : B_i'\{X\}^{\,i \in 1..n+m}]$$
$$Class(A') \equiv [new : A', l_i : \forall(X <: A')X \to B_i'\{X\}^{\,i \in 1..n+m}]$$

We say that:

l_i is *inheritable* from $Class(A)$ into $Class(A')$
if and only if $X <: A'$ implies $B_i\{X\} <: B_i'\{X\}$, for all $i \in 1..n$

- Inheritability is not an immediate consequence of $A' <: A$.
- Inheritability is expected between a class type C and another class type C' obtained as an extension of C.

When l_i is inheritable, we have:

$$\forall(X<:A)X\rightarrow B_i\{X\} <: \forall(X<:A')X\rightarrow B_i'\{X\}$$

So, if $c : Class(A)$ and l_i is inheritable, we have $c.l_i : \forall(X<:A')X\rightarrow B_i'\{X\}$.

Then $c.l_i$ can be reused when building a class $c' : Class(A')$.

For example, *set* is inheritable from *Class*(*Cell*) to *Class*(*GCell*):

$$
\begin{aligned}
&Class(GCell) \triangleq \\
&\quad [new : GCell, \\
&\quad\ contents : \forall(Self{<:}GCell)\ Self{\to}Nat, \\
&\quad\ set : \forall(Self{<:}GCell)\ Self{\to}Nat{\to}Self, \\
&\quad\ get : \forall(Self{<:}GCell)\ Self{\to}Nat]
\end{aligned}
$$

$$
\begin{aligned}
&gcellClass : Class(GCell) \triangleq \\
&\quad [new = \varsigma(z{:}Class(GCell))\ obj(Self{=}GCell)[\ldots], \\
&\quad\ contents = \lambda(Self{<:}GCell)\ \lambda(s{:}Self)\ 0, \\
&\quad\ set = cellClass.set, \\
&\quad\ get = \lambda(Self{<:}GCell)\ \lambda(s{:}Self)\ s.contents]
\end{aligned}
$$

Self Types

and

Higher-Order Object Calculi

Inheritance without Subtyping?

- Up to this point, subtyping justifies inheritance.
- This leads to a great conceptual economy.
- It corresponds well to the rules of most typed languages.
- But there are situations where one may want inheritance without subtyping.
- There are also a few languages that support inheritance without subtyping (*e.g.*, Theta, TOOPLE, Emerald).

The Problem

Consider cells with an equality method:

$$\begin{aligned} &\mathit{CellEq} \triangleq \\ &\quad \mu(X)[\mathit{contents} : \mathit{Nat}, \mathit{set} : \mathit{Nat} \to X, \mathit{eq} : X \to \mathit{Bool}] \\ &\mathit{CellSEq} \triangleq \\ &\quad \mu(X)[\mathit{contents} : \mathit{Nat}, \mathit{set} : \mathit{Nat} \to X, \mathit{sign} : \mathit{Bool}, \mathit{eq} : X \to \mathit{Bool}] \end{aligned}$$

But then *CellSEq* is not a subtype of *CellEq*.

This situation is typical when there are binary methods, such as *eq*.

Giving up on subtyping is necessary for soundness.

On the other hand, it would be good still to be able to reuse code, for example the code $eq = \varsigma(x)\lambda(y)\ x.\mathit{contents} = y.\mathit{contents}$.

Solutions

- Avoid contravariant occurrences of recursion variables, to preserve subtyping.

  $$CellEq' \triangleq \mu(X)[..., eq : Cell \to Bool]$$
  $$CellSEq' \triangleq \mu(X)[..., sign : Bool, eq : Cell \to Bool]$$

- Axiomatize a primitive matching relation between types <#, work out its theory, and relate it somehow to code reuse.

  $$CellSeq <\# CellEq$$

  (But the axioms are not trivial, and not unique.)

- Move up to higher-order calculi and see what can be done there. There are two approaches:

  ~ F-bounded quantification (Cook et al.);

  ~ higher-order subtyping (us).

The Higher-Order Path

- Let us define two type operators:

$$CellEqOp \triangleq$$
$$\lambda(X)[contents : Nat, set : Nat \to X, eq : X \to Bool]$$
$$CellSEqOp \triangleq$$
$$\lambda(X)[contents : Nat, set : Nat \to X, sign : Bool, eq : X \to Bool]$$

- We write:

$$CellEqOp :: Ty \Rightarrow Ty$$
$$CellSEqOp :: Ty \Rightarrow Ty$$

to mean that these are type operators.

- Then, for each type X, we have:

$$CellSEqOp(X) <: CellEqOp(X)$$

- This is higher-order subtyping: pointwise subtyping between type operators.
- We say that *CellSEqOp* is a suboperator of *CellEqOp*, and we write:

$$CellSEqOp <: CellEqOp :: Ty \Rightarrow Ty$$

- Object types can be obtained as fixpoints of these operators:

$$CellEq \triangleq \mu(X)CellEqOp(X)$$
$$CellSEq \triangleq \mu(X)CellSEqOp(X)$$

- So although *CellSEq* is not a subtype of *CellEq*, these types still have something in common:
  they are fixpoints of two suboperators of *CellEqOp*.

- We can then write polymorphic functions by quantifying over suboperators:

$$
\begin{aligned}
&eqF \triangleq \\
&\quad \lambda(F <: CellEqOp :: Ty \Rightarrow Ty)\ \lambda(x : \mu(X)F(X))\ \lambda(y : \mu(X)F(X)) \\
&\qquad x.contents = y.contents \\
&: \forall(F <: CellEqOp :: Ty \Rightarrow Ty)\ \mu(X)F(X) \to \mu(X)F(X) \to Bool
\end{aligned}
$$

- This function can be instantiated at both *CellEqOp* and *CellSEqOp*.
- This function can also be used to write pre-methods for classes. (For this we let pre-methods be polymorphic functions.)

Conclusions

- Functions vs. objects:
  - ~ Functions can be translated into objects.
    Therefore, pure object-based languages are at least as expressive as procedural languages.
    (Despite all the Smalltalk philosophy, to our knowledge nobody had proved that one can build functions from objects.)
  - ~ Conversely, using sophisticated type systems, it is possible to translate objects into functions.
    (But this translation is difficult and not practical.)

- Classes vs. objects:
  - ~ Classes can be encoded in object calculi, easily and faithfully. Therefore, object-based languages are just as expressive as class-based ones. (To our knowledge, nobody had shown that one can build type-correct classes out of objects.)
  - ~ Method update, a distinctly object-based construct, is tractable and can be useful.

- We can make (some) sense of object-oriented constructs.
  - ~ Object calculi are simple enough to permit precise definitions and proofs.
  - ~ Object calculi are quite expressive and object-oriented.