A Theory of Objects

Martín Abadi & Luca Cardelli

Digital Equipment Corporation
Systems Research Center

OOPSLA'96 Tutorial

Outline

- Topic of this tutorial: a foundation for object-oriented languages based on object calculi.
- Part 1: Object-oriented features.
- Part 2: Object calculi.
- Part 3: Interpretation of object-oriented languages.

Object-Oriented Features

CLASS-BASED LANGUAGES

- The mainstream.
- We review only common, kernel properties.

Classes and Objects

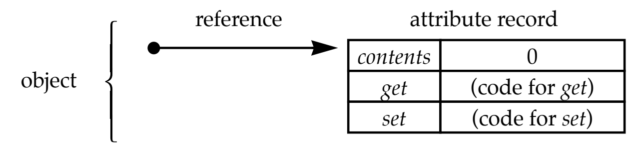
- Classes are descriptions of objects.
- Example: storage cells.

class *cell* **is**
    var *contents*: *Integer* := 0;
    method *get*(): *Integer* **is**
        return self.*contents*;
    end;
    method *set*(*n*: *Integer*) **is**
        self.*contents* := *n*;
    end;
end;

- Classes generate objects.
- Objects can refer to themselves.

Naive Storage Model

- Object = reference to a record of attributes.



Naive storage model

Object Operations

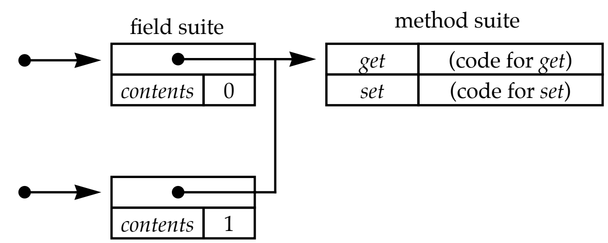
- Object creation.
  - ~ *InstanceTypeOf(c)* indicates the type of an object of class *c*.

    var *myCell*: *InstanceTypeOf*(*cell*) := **new** *cell*;
- Field selection.
- Field update.
- Method invocation.

procedure *double*(*aCell*: *InstanceTypeOf*(*cell*)) **is**
    aCell.*set*(2 * *aCell*.*get*());
end;

The Method-Suites Storage Model

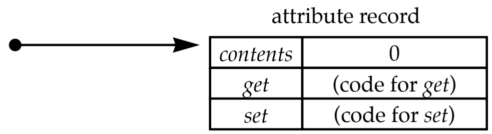
- A more refined storage model for class-based languages.



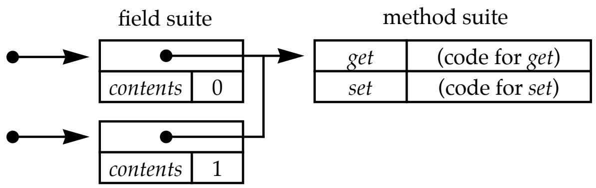
Method suites

Embedding vs. Delegation

- In the naive storage model, methods are *embedded* in objects.



- In the methods-suites storage model, methods are *delegated* to the method suites.



- Naive and method-suites models are semantically equivalent for class-based languages.
- They are not equivalent (as we shall see) in object-based languages, where the difference between embedding and delegation is critical.

Method Lookup

- Method lookup is the process of finding the code to run on a method invocation *o.m*(…). The details depend on the language and the storage model.
- In class-based languages, method lookup gives the *illusion* that methods are embedded in objects (cf. *o.x*, *o.m*(...)), hiding storage model details.
- Self is always the *receiver*: the object that *appears* to contain the method.
- Features that would distinguish embedding from delegation implementations (e.g., method update) are usually avoided.

Subclasses and Inheritance

- A *subclass* is a differential description of a class.
- The *subclass relation* is the partial order induced by the subclass declarations.
- Example: restorable cells.

```
subclass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;
```

Subclasses and Self

- Because of subclasses, the meaning of **self** becomes dynamic.

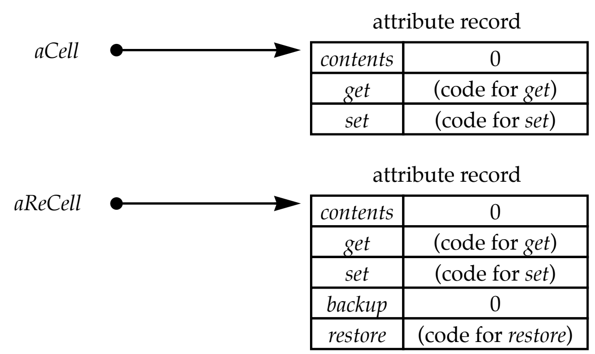
  self.*m*(...)

- Because of subclasses, the concept of **super** becomes useful.

  super.*m*(...)

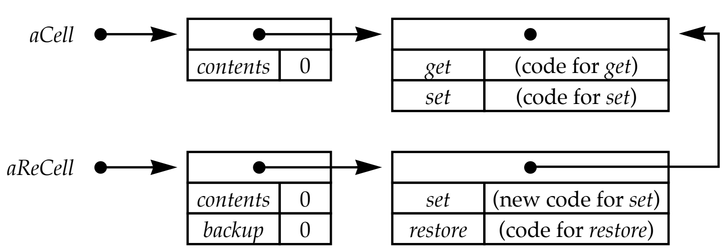
Subclasses and Naive Storage

- In the naive implementation, the existence of subclasses does not cause any change in the storage model.



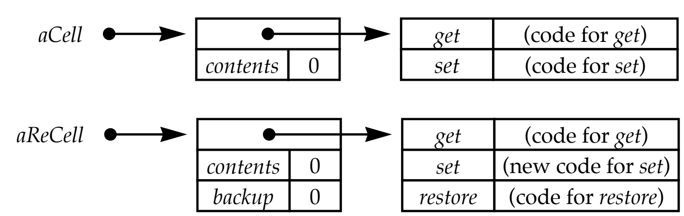
Subclasses and Method Suites

- Because of subclasses, the method-suites model has to be reconsidered. In dynamically-typed class-based languages, method suites are chained:



Hierarchical method suites

- In statically-typed class-based languages, however, the method-suites model can be maintained in its original form.



Collapsed method suites

Embedding/Delegation View of Class Hierarchies

- Hierarchical method suites: *delegation* (of objects to suites) combined with *delegation* (of sub-suites to super-suites).
- Collapsed method suites: *delegation* (of objects to suites) combined with *embedding* (of super-suites in sub-suites).

Class-Based Summary

- In analyzing the meaning and implementation of class-based languages we end up inventing and analyzing sub-structures of objects and classes.
- These substructures are independently interesting: they have their own semantics, and can be combined in useful ways.
- What if these substructures were directly available to programmers?

OBJECT-BASED LANGUAGES

- Slow to emerge.
- Simple and flexible.
- Usually untyped.

- Just objects and dynamic dispatch.
- When typed, just object types and subtyping.
- Direct object-to-object inheritance.

An Object, All by Itself

- Classes are replaced by object constructors.
- Object types are immediately useful.

```
ObjectType Cell is
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;

object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;
```

An Object Generator

- Procedures as object generators.

```
procedure newCell(m: Integer): Cell is
    object cell: Cell is
        var contents: Integer := m;
        method get(): Integer is return self.contents end;
        method set(n: Integer) is self.contents := n end;
    end;
    return cell;
end;
var cellInstance: Cell := newCell(0);
```

- Quite similar to classes!

Decomposing Class-Based Features

- General idea: decompose class-based notions and orthogonally recombine them.
- We have seen how to decompose simple classes into objects and procedures.
- We will now investigate how to decompose inheritance.
  - ~ Object generation by parameterization.
  - ~ Vs. object generation by cloning and mutation.

Prototypes and Clones

- Classes describe objects.
- Prototypes describe objects and *are* objects.
- Regular objects are clones of prototypes.

```
var cellClone: Cell := clone cellInstance;
```

- **clone** is a bit like **new**, but operates on objects instead of classes.

Mutation of Clones

- Clones are customized by mutation (e.g., update).
- Field update.

```
cellClone.contents := 3;
```

- Method update.

```
cellClone.get :=
    method (): Integer is
        if self.contents < 0 then return 0 else return self.contents end;
    end;
```

- Self-mutation possible.

Object-Based Inheritance

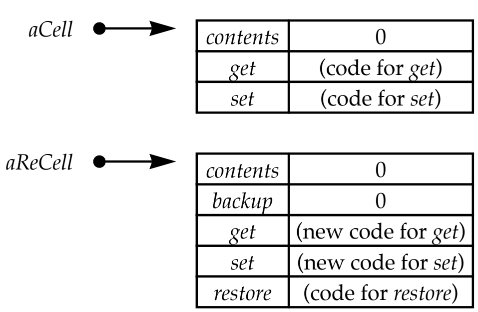
- Object generation can be obtained by procedures, but with no real notion of inheritance.
- Object inheritance can be achieved by cloning (reuse) and update (override), but with no shape change.
- How can one inherit with a change of shape?
- An option is object extension. But:
  - ~ Not easy to typecheck.
  - ~ Not easy to implement efficiently.
  - ~ Provided rarely or restrictively.

Donors and Hosts

- General object-based inheritance: building new objects by "reusing" attributes of existing objects.
- Two orthogonal aspects:
  - ~ obtaining the attributes of a ***donor*** object, and
  - ~ incorporating those attributes into a new ***host*** object.
- Four categories of object-based inheritance:
  - ~ The attributes of a donor may be obtained ***implicitly*** or ***explicitly***.
  - ~ Orthogonally, those attributes may be either ***embedded*** into a host, or ***delegated*** to a donor.

Embedding

- Host objects contain <u>copies</u> of the attributes of donor objects.



Embedding

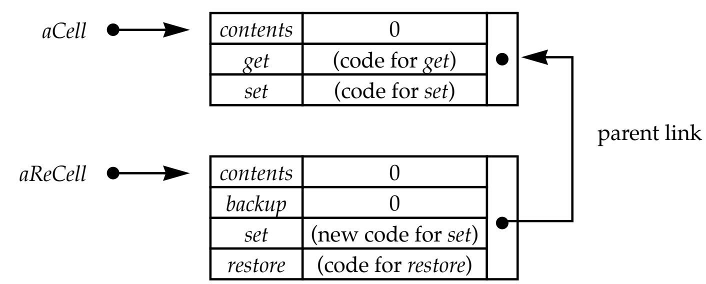
Embedding-Based Languages

- Embedding provides the simplest explanation of the standard semantics of **self** as the receiver.
- Embedding was described by Borning as part of one of the first proposals for prototype-based languages.
- Recently, it has been adopted by languages like Kevo and Obliq. We call these languages *embedding-based* (*concatenation-based*, in Kevo terminology).

Delegation

- Host objects contain *links* to the attributes of donor objects.
- Prototype-based languages that permit the sharing of attributes across objects are called *delegation-based*.
- Operationally, delegation is the redirection of field access and method invocation from an object or prototype to another, <u>in such a way that an object can be seen as an extension of another</u>.
- A crucial aspect of delegation inheritance is the interaction of donor links with the binding of **self**.

Delegation Inheritance



(Single-parent) Delegation

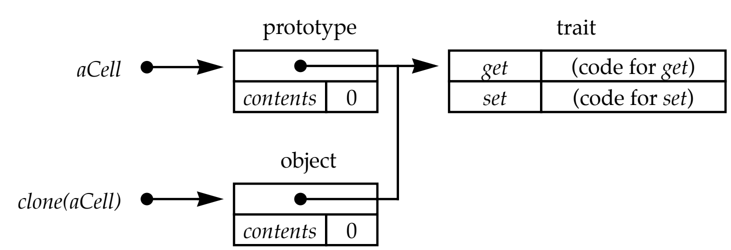
- Note: similar to hierarchical method suites.

Traits: from Prototypes back to Classes?

- Prototypes were initially intended to replace classes.
- Several prototype-based languages, however, seem to be moving towards a more traditional approach based on class-like structures.
- Prototypes-based languages like Omega, Self, and Cecil have evolved usage-based distinctions between objects.

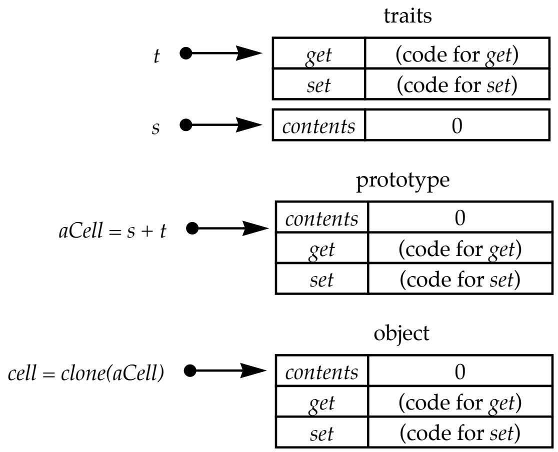
Different Kinds of Objects

- Trait objects.
- Prototype objects.
- Normal objects.



Traits

Embedding-Style Traits



Traits

Traits are not Prototypes

- This separation of roles violates the original spirit of prototype-based languages: traits objects cannot function on their own. They typically lack instance variables.
- With the separation between traits and other objects, we seem to have come full circle back to class-based languages and to the separation between classes and instances.
- Trait-based techniques looks exactly like implementation techniques for classes.

Contributions of the Object-Based Approach

- The achievement of object-based languages is to make clear that classes are just one of the possible ways of generating objects with common properties.
- Objects are more primitive than classes, and they should be understood and explained before classes.
- Different class-like constructions can be used for different purposes; hopefully, more flexibly than in strict class-based languages.

Going Further

- Language analysis:
  - ~ Class-based langs. → Object-based langs. → Object calculi
- Language synthesis:
  - ~ Object calculi → Object-based langs. → Class-based langs.

Our Approach to Modeling

- We have identified embedding and delegation as underlying many object-oriented features.
- In our object calculi, we choose embedding over delegation as the principal object-oriented paradigm.
- The resulting calculi can model classes well, although they are not class-based (since classes are not built-in).
- They can model delegation-style traits just as well, but not “true” delegation. (Object calculi for delegation exist but are more complex.)

Object Calculi

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	**Ob$_1$**	**Ob$_{1<:}$**	*nn*	**Ob$_{1\mu}$**	**Ob$_{1<:\mu}$**	*nn*	**impς**	*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 say "imperative" explicitly.

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 $b \rightsquigarrow c$ means that b reduces to c.
- 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 $o \equiv [l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]$ (l_i distinct)

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

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

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

Let $o' \triangleq [l = \varsigma(x)x]$ — self-returning method
then $o'.l \rightsquigarrow x\{\!\{x{\leftarrow}o'\}\!\} \equiv o'$

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

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.

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.

Expressiveness

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

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

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

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

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 $cell.set(3)$
$\rightsquigarrow (\lambda(n)[contents = 0, set = \varsigma(x)\ \lambda(n)\ x.contents := n]$
$.contents{:=}n)(3)$
$\rightsquigarrow [contents = 0, set = \varsigma(x)\lambda(n)\ x.contents := n]$
$.contents{:=}3$
$\rightsquigarrow [contents = 3, set = \varsigma(x)\ \lambda(n)\ x.contents := n]$

and $cell.set(3).contents$
$\rightsquigarrow \ldots$
$\rightsquigarrow 3$

A Cell with an Accessor

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

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

A Cell with Undo

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

- 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 code above works only if update has a functional semantics.
An imperative version is:

$uncell \triangleq$
$[contents = 0,$
$set = \varsigma(x)\ \lambda(n)$
$let\ y = clone(x)\ in$
$(x.undo := y).contents := n,$
$undo = \varsigma(x)\ x]$

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:

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

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:

$$\begin{aligned}&zero \triangleq\\&\quad[case = \lambda(z)\,\lambda(s)\,z,\\&\quad\;\; succ = \varsigma(x)\,x.case := \lambda(z)\,\lambda(s)\,s(x)\,]\end{aligned}$$

So:

$$\begin{aligned}&zero && &&\equiv [case = \lambda(z)\,\lambda(s)\,z,\;\; succ = \ldots\,]\\&one &&\triangleq zero.succ &&\equiv [case = \lambda(z)\,\lambda(s)\,s(zero),\;\; succ = \ldots\,]\\&pred &&\triangleq \lambda(n)\,n.case(zero)(\lambda(p)p)\end{aligned}$$

A Calculator

The calculator uses method update for storing pending operations.

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

We obtain the following calculator-style behavior:

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

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

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

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

The translation validates the β rule:

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

For example:

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

The translation has typed and imperative variants.

Procedures as Imperative Objects

Translation of an imperative λ-calculus

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

$\langle\!\langle x := a \rangle\!\rangle \triangleq$

$\quad let\ y = \langle\!\langle a \rangle\!\rangle$

$\quad in\ x.arg := y$

$\langle\!\langle \lambda(x)b \rangle\!\rangle \triangleq$

$\quad [arg = \varsigma(x)\ x.arg,$

$\quad\ val = \varsigma(x)\ \langle\!\langle b \rangle\!\rangle\{\!\{x \leftarrow x.arg\}\!\}]$

$\langle\!\langle b(a) \rangle\!\rangle \triangleq$

$\quad let\ f = clone(\langle\!\langle b \rangle\!\rangle)$

$\quad in\ let\ y = \langle\!\langle a \rangle\!\rangle$

$\quad in\ (f.arg := y).val$

Cloning on application corresponds to allocating a new stack frame.

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

$cellClass \triangleq$

$\quad [new = \varsigma(z)$

$\quad\quad [contents = \varsigma(x)\ z.contents(x),\ set = \varsigma(x)\ z.set(x)],$

$\quad contents = \lambda(x)\ 0,$

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

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':

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

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

Inheritance for Cells

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

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

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

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)

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

(Sub Object) (l_i distinct)

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

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

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

(Val Select)

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

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

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

(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 ∅)
$$\frac{}{\varnothing \vdash \diamond}$$

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

Some Results (for the Functional Calculus)

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 $\varnothing \vdash a : C$
and a reduces to v
then $\varnothing \vdash v : C$.

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!

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*
$gcell \triangleq [\ldots, get = \varsigma(x{:}GCell)\, x.contents]$
has types *GCell* and *Cell*

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:

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

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

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

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

$Class(Cell)$ and $Class(GCell)$ are not related by subtyping,
but inheritance is possible.

Variance Annotations

In order to gain expressiveness within a first-order setting,
we extend the syntax of object types with variance annotations:

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

Each υ_i is a variance annotation; it is one of three symbols ^{o}, $^+$, and $^-$.

Intuitively,

- $^+$ means read-only: it prevents update, but allows covariant component subtyping;
- $^-$ means write-only: it prevents invocation, but allows contravariant component subtyping;
- ^{o} means read-write: it allows both invocation and update, but requires exact matching in subtyping.

By convention, any omitted annotations are taken to be equal to ^{o}.

Subtyping with Variance Annotations

$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^o{:}B'\ \ldots]$ if $B \equiv B'$ — invariant (read-write)

$[\ldots\ l^+{:}B\ \ldots] <: [\ldots\ l^+{:}B'\ \ldots]$ if $B <: B'$ — covariant (read-only)

$[\ldots\ l^-{:}B\ \ldots] <: [\ldots\ l^-{:}B'\ \ldots]$ if $B' <: B$ — contravariant (write-only)

$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^+{:}B'\ \ldots]$ if $B <: B'$ — invariant <: variant
$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^-{:}B'\ \ldots]$ if $B' <: B$

Protection by Subtyping

- Variance annotations can provide protection against updates from the outside.
- In addition, object components can be hidden by subsumption.

For example:

Let $GCell \triangleq [contents : Nat,\ set : Nat \rightarrow [\,],\ get : Nat]$
$PGCell \triangleq [set : Nat \rightarrow [\,],\ get : Nat]$
$ProtectedGCell \triangleq [set^+ : Nat \rightarrow [\,],\ get^+ : Nat]$
$gcell : GCell$
then $GCell <: PGCell <: ProtectedGCell$
so $gcell : ProtectedGCell$.

Given a *ProtectedGCell*, one cannot access its *contents* directly.

From the inside, *set* and *get* can still update and read *contents*.

Encoding Function Types

An invariant translation of function types:

$$\langle\!\langle A \rightarrow B \rangle\!\rangle \triangleq [arg : \langle\!\langle A \rangle\!\rangle,\ val : \langle\!\langle B \rangle\!\rangle]$$

A covariant/contravariant translation, using annotations:

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

A covariant/contravariant translation, using quantifiers:

$$\langle\!\langle A \rightarrow B \rangle\!\rangle \triangleq \forall(X <: \langle\!\langle A \rangle\!\rangle)\, \exists(Y <: \langle\!\langle B \rangle\!\rangle)\, [arg : X,\ val : Y]$$

where $\forall$ is for polymorphism and $\exists$ is for data abstraction.

Recursive Types

Informally, we may want to define a recursive type as in:

$$Cell \triangleq [contents : Nat,\ set : Nat \rightarrow Cell]$$

Formally, we write instead:

$$Cell \triangleq \mu(X)[contents : Nat,\ set : Nat \rightarrow X]$$

Intuitively, $\mu(X)A\{X\}$ is the solution for the equation $X = A\{X\}$.

Subtyping Recursive Types

The basic subtyping rule for recursive types is:

$\mu(X)A\{X\} <: \mu(X)B\{X\}$
if
either $A\{X\}$ and $B\{X\}$ are equal for all X
or $A\{X\} <: B\{\!\{Y\}\!\}$ for all X and Y such that $X <: Y$

There are variants, for example:

$\mu(X)A\{X\} <: \mu(X)B\{X\}$
if
either $A\{X\}$ and $B\{X\}$ are equal for all X
or $A\{X\} <: B\{\!\{\mu(X)B\{X\}\}\!\}$ for all X such that $X <: \mu(X)B\{X\}$

But $A\{X\} <: B\{X\}$ does not imply $\mu(X)A\{X\} <: \mu(X)B\{X\}$.

Cells (with Recursive Types)

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

The type *Cell* is a recursive type.

Now we can typecheck *cell.set*(3)*.contents*.

Because of the recursion, we do not get interesting subtypings.

Let $GCell \triangleq [contents : Nat, set : Nat \rightarrow GCell, get : Nat]$
then *GCell* is not a subtype of *Cell*.

The fact that *GCell* is not a subtype of *Cell* is unacceptable, but necessary for soundness.

Consider the following correct but somewhat strange *GCell*:

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

If *GCell* were a subtype of *Cell* then we would have:

$gcell' : Cell$
$gcell'' : Cell \triangleq (gcell'.set := \lambda(n{:}Nat)\ cell)$

where *cell* is a fixed element of *Cell*, without a *get* method.
Then we can write:

$m : Nat \triangleq gcell''.contents$

But the computation of *m* yields a “message not understood” error.

Five Solutions (Overview)

• Avoid methods specialization, redefining *GCell*:

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

~ This is a frequent approach in common languages.

~ It requires dynamic type tests after calls to the *set* method. *E.g.*,

typecase *gcell.set*(3)
when (*x*:*GCell*) *x.get*
else ...

- Add variance annotations:

$$Cell \triangleq [contents : Nat, set^{+} : Nat \to Cell]$$
$$GCell \triangleq [contents : Nat, set^{+} : Nat \to GCell, get : Nat]$$

~ This approach yields the desired subtypings.

~ But it forbids even sound updates of the *set* method.

~ It would require reconsidering the treatment of classes in order to support inheritance of the *set* method.

- Go back to an imperative framework, where the typing problem disappears because the result type of *set* is [].

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

~ This works sometimes.

~ But methods that allocate a new object of the type of self still call for the use of recursive types:

$$UnCell \triangleq [contents : Nat, set : Nat \to [], undo : UnCell]$$

- Axiomatize some notion of Self types, and write:

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

~ But the rules for Self types are not trivial or obvious.

- Move up to higher-order calculi, and see what can be done there.

$$Cell \triangleq \exists(Y{<:}Cell)\, [contents : Nat, set : Nat \to Y]$$
$$GCell \triangleq \exists(Y{<:}GCell)\, [contents : Nat, set : Nat \to Y, get : Nat]$$

~ The existential quantifiers yield covariance, so $GCell <: Cell$.

~ Intuitively, the existentially quantified type is the type of self: the Self type.

~ This technique is general, and suggests sound rules for primitive Self types.

We obtain:

~ subtyping with methods that return self,

~ inheritance for methods that return self or that take arguments of the type of self (“binary methods”), but without subtyping.

Typed Reasoning

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

There are some subtleties in reasoning about objects.

Consider:

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

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:

$$a \neq b : A$$

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).

(Work in progress:
specification and verification for a typed object-oriented language.)

Conclusions

Object calculi are both simple and expressive.

- 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.

Interpretation of Object-Oriented Languages

A FIRST-ORDER LANGUAGE

- Let's assess the contributions that object calculi bring to the task of modeling programming language constructs.
- For this purpose, we study a simple object-oriented language named O–1.
- We have studied more advanced languages that include Self types and matching.

Features of O–1

- Both class-based and object-based constructs.
- First-order object types with subtyping and variance annotations.
- Classes with single inheritance; method overridding and specialization.
- Recursion.
- Typecase.
- Separation interfaces from implementations, and inheritance from subtyping.

Syntax

Syntax of O–1 types

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

Syntax of O–1 terms

a,b,c ::=	terms
x	variable
object$(x{:}A)\ l_i{=}b_i\ ^{i\in 1..n}$ **end**	direct object construction
$a.l$	field selection / method invocation
$a.l := b$	update with a term
$a.l :=$ **method**$(x{:}A)\ b$ **end**	update with a method
new c	object construction from a class
root	root class
subclass of $c{:}C$ **with**$(x{:}A)$	subclass
$l_i{=}b_i\ ^{i\in n+1..n+m}$	additional attributes
override $l_i{=}b_i\ ^{i\in Ovr\subseteq 1..n}$ **end**	overridden attributes
$c\text{^}l(a)$	class selection
typecase a **when** $(x{:}A)b_1$ **else** b_2 **end**	typecase

- We could drop the object-based constructs (object construction and method update). The result would be a language expressive enough for traditional class-based programming.
- Alternatively, we could drop the class-based construct (root class, subclass, new, and class selection), obtaining an object-based language.
- Classes, as well as objects, are first-class values. A parametric class can be obtained as a function that returns a class.

Abbreviations

Root $\triangleq$
  Class(**Object**(X)[])

class with$(x{:}A)\ l_i{=}b_i\ ^{i\in 1..n}$ **end** $\triangleq$
  subclass of root:**Root with**$(x{:}A)\ l_i{=}b_i\ ^{i\in 1..n}$ **override end**

subclass of $c{:}C$ **with** $(x{:}A)$ … **super**.l … **end** $\triangleq$
  subclass of $c{:}C$ **with** $(x{:}A)$ … $c\text{^}l(x)$ … **end**

object$(x{:}A)$ … l **copied from** c … **end** $\triangleq$
  object$(x{:}A)$ … $l{=}c\text{^}l(x)$ … **end**

Examples

- We assume basic types (*Bool*, *Int*) and function types ($A{\to}B$, contravariant in A and covariant in B).

  $Point \triangleq$ **Object**$(X)[x{:}\ Int,\ eq^{+}{:}\ X{\to}Bool,\ mv^{+}{:}\ Int{\to}X]$

  $CPoint \triangleq$ **Object**$(X)[x{:}\ Int,\ c{:}\ Color,\ eq^{+}{:}\ Point{\to}Bool,\ mv^{+}{:}\ Int{\to}Point]$

- $CPoint <: Point$
- The type of mv in $CPoint$ is $Int{\to}Point$.
  One can explore the effect of changing it to $Int{\to}X$.
- The type of eq in $CPoint$ is $Point{\to}Bool$.
  If we were to change it to $X{\to}Bool$ we would lose the subtyping $CPoint <: Point$.

Class(Point)

pointClass : **Class**(*Point*) ≜
  class with (*self*: *Point*)
    x = 0,
    eq = **fun**(*other*: *Point*) *self.x* = *other.x* **end**,
    mv = **fun**(*dx*: *Int*) *self.x* := *self.x*+*dx* **end**
  end

Class(CPoint)

cPointClass : **Class**(*CPoint*) ≜
  subclass of *pointClass*: **Class**(*Point*)
  with (*self*: *CPoint*)
    c = *black*
  override
    eq = **fun**(*other*: *Point*)
        typecase other
        when (*other'*: *CPoint*) **super**.*eq*(*other'*) *and self.c* = *other'.c*
        else *false*
        end
      end
  end

Comments

- The class *cPointClass* inherits *x* and *mv* from its superclass *pointClass*.
- Although it could inherit *eq* as well, *cPointClass* overrides this method as follows.
  - ~ The definition of *Point* requires that *eq* work with any argument *other* of type *Point*.
  - ~ In the *eq* code for *cPointClass*, the typecase on *other* determines whether *other* has a color.
  - ~ If so, *eq* works as in *pointClass* and in addition tests the color of *other*.
  - ~ If not, *eq* returns *false*.

- We can use *cPointClass* to create color points of type *CPoint*:

  cPoint : *CPoint* ≜ **new** *cPointClass*

- Calls to *mv* lose the color information.
- In order to access the color of a point after it has been moved, a typecase is necessary:

  movedColor : *Color* ≜
    typecase *cPoint.mv*(1)
    when (*cp*: *CPoint*) *cp.c*
    else *black*
    end

Typing

- The rules of O–1 are based on the following judgments:

Judgments

$E \vdash \diamond$	environment E is well-formed
$E \vdash A$	A is a well-formed type in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash \upsilon A <: \upsilon' B$	A is a subtype of B in E, with variance annotations υ and υ'
$E \vdash a : A$	a has type A in E

- The rules for environments are standard:

Environments

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

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

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

Type Formation Rules

Types

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

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

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

(Type Class) (where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i : B_i\{X\} \ ^{i \in 1..n}]$)
$$\frac{E \vdash A}{E \vdash \mathbf{Class}(A)}$$

Subtyping Rules

- Note that there is no rule for subtyping class types.

Subtyping

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

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

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

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

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

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

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

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

Term Typing Rules

Terms

(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 $A \equiv \mathbf{Object}(X)[l_i \upsilon_i : B_i\{X\} \ ^{i \in 1..n}]$)
$$\frac{E, x:A \vdash b_i : B_i\{\!\{A\}\!\} \qquad \forall i \in 1..n}{E \vdash \mathbf{object}(x:A) \ l_i = b_i \ ^{i \in 1..n} \ \mathbf{end} : A}$$

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

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

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

$$\frac{E \vdash a : A \qquad E \vdash b : B_j\{\!\{A\}\!\} \qquad \upsilon_j \in \{^{\circ}, ^{-}\} \qquad j \in 1..n}{E \vdash a.l_j := b : A}$$

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

$$\frac{E \vdash a : A \qquad E, x{:}A \vdash b : B_j\{\!\{A\}\!\} \qquad \upsilon_j \in \{^{\circ}, ^{-}\} \qquad j \in 1..n}{E \vdash a.l_j := \textbf{method}(x{:}A)b\ \textbf{end} : A}$$

(Val New)

$$\frac{E \vdash c : \textbf{Class}(A)}{E \vdash \textbf{new}\ c : A}$$

(Val Root)

$$\frac{E \vdash \diamond}{E \vdash \textbf{root} : \textbf{Class}(\textbf{Object}(X)[])}$$

(Val Subclass) (where $A \equiv \textbf{Object}(X)[l_i \upsilon_i{:}B_i\{X\}^{\,i\in 1..n+m}]$, $A' \equiv \textbf{Object}(X')[l_i \upsilon_i'{:}B_i'\{X'\}^{\,i\in 1..n}]$, $Ovr \subseteq 1..n$)

$$\frac{\begin{array}{c} E \vdash c' : \textbf{Class}(A') \qquad E \vdash A <: A' \\ E \vdash B_i'\{\!\{A'\}\!\} <: B_i\{\!\{A\}\!\} \qquad \forall i \in 1..n - Ovr \\ E, x{:}A \vdash b_i : B_i\{\!\{A\}\!\} \qquad \forall i \in Ovr \cup n+1..n+m \end{array}}{\begin{array}{c} E \vdash \textbf{subclass of}\ c'{:}\textbf{Class}(A')\ \textbf{with}(x{:}A)\ l_i{=}b_i^{\ i\in n+1..n+m}\ \textbf{override}\ l_i{=}b_i^{\ i\in Ovr}\ \textbf{end} \\ : \textbf{Class}(A) \end{array}}$$

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

$$\frac{E \vdash a : A \qquad E \vdash c : \textbf{Class}(A) \qquad j \in 1..n}{E \vdash c\hat{\ }l_j(a) : B_j\{\!\{A\}\!\}}$$

(Val Typecase)

$$\frac{E \vdash a : A' \qquad E, x{:}A \vdash b_1 : D \qquad E \vdash b_2 : D}{E \vdash \textbf{typecase}\ a\ \textbf{when}\ (x{:}A)b_1\ \textbf{else}\ b_2\ \textbf{end} : D}$$

- These rules are hard to read and understand.
- But they are the ultimate truth about typing in O–1.

Translation

- We give a translation into a functional calculus (with all the features described earlier).
- A similar translation could be given into an appropriate imperative calculus.
- At the level of types, the translation is simple.
  - ~ We write $\langle\!\langle A \rangle\!\rangle$ for the translation of A.
  - ~ We map an object type $\textbf{Object}(X)[l_i \upsilon_i{:}B_i^{\ i\in 1..n}]$ to a recursive object type $\mu(X)[l_i \upsilon_i{:}\langle\!\langle B_i \rangle\!\rangle^{\ i\in 1..n}]$.
  - ~ We map a class type $\textbf{Class}(\textbf{Object}(X)[l_i \upsilon_i{:}B_i\{X\}^{\ i\in 1..n}])$ to an object type that contains components for pre-methods and a *new* component.

Translation of Types

Translation of O–1 types

$\langle\!\langle X \rangle\!\rangle \triangleq X$

$\langle\!\langle \mathbf{Top} \rangle\!\rangle \triangleq Top$

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

$\langle\!\langle \mathbf{Class}(A) \rangle\!\rangle \triangleq [new^+ : \langle\!\langle A \rangle\!\rangle, \ l_i^+ : \langle\!\langle A \rangle\!\rangle \to \langle\!\langle B_i \rangle\!\rangle \{\langle\!\langle A \rangle\!\rangle\} \ ^{i \in 1..n}]$
where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i : B_i\{X\} \ ^{i \in 1..n}]$

Translation of O–1 environments

$\langle\!\langle \emptyset \rangle\!\rangle \triangleq \emptyset$

$\langle\!\langle E, X <: A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, X <: \langle\!\langle A \rangle\!\rangle$

$\langle\!\langle E, x : A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, x : \langle\!\langle A \rangle\!\rangle$

Translation of Terms

- The translation is guided by the type structure.
- The translation maps a class to a collection of pre-methods plus a *new* method.
  - ~ For a class **subclass of** c' … **end**, the collection of pre-methods consists of the pre-methods of c' that are not overridden, plus all the pre-methods given explicitly.
  - ~ The *new* method assembles the pre-methods into an object; **new** c is interpreted as an invocation of the *new* method of $\langle\!\langle c \rangle\!\rangle$.
  - ~ The construct $c\text{^}l(a)$ is interpreted as the extraction and the application of a pre-method.

(Simplified) Translation of O–1 terms

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

$\langle\!\langle \mathbf{object}(x{:}A) \ l_i = b_i \ ^{i \in 1..n} \ \mathbf{end} \rangle\!\rangle \triangleq [l_i = \varsigma(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b_i \rangle\!\rangle \ ^{i \in 1..n}]$

$\langle\!\langle a.l \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l$

$\langle\!\langle a.l := b \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l := \langle\!\langle b \rangle\!\rangle$

$\langle\!\langle a.l := \mathbf{method}(x{:}A) \ b \ \mathbf{end} \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l \Leftarrow \varsigma(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b \rangle\!\rangle$

$\langle\!\langle \mathbf{new} \ c \rangle\!\rangle \triangleq \langle\!\langle c \rangle\!\rangle.new$

$\langle\!\langle \mathbf{root} \rangle\!\rangle \triangleq [new = [\,]]$

$\langle\!\langle \mathbf{subclass\ of} \ c'{:}\mathbf{Class}(A') \ \mathbf{with}(x{:}A) \ l_i = b_i \ ^{i \in n+1..n+m} \ \mathbf{override} \ l_i = b_i \ ^{i \in Ovr} \ \mathbf{end} \rangle\!\rangle \triangleq$
$[new = \varsigma(z{:}\langle\!\langle \mathbf{Class}(A) \rangle\!\rangle)[l_i = \varsigma(s{:}\langle\!\langle A \rangle\!\rangle)z.l_i(s) \ ^{i \in 1..n+m}],$
$l_i = \langle\!\langle c' \rangle\!\rangle.l_i \ ^{i \in 1..n-Ovr},$
$l_i = \lambda(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b_i \rangle\!\rangle \ ^{i \in Ovr \cup n+1..n+m}]$

$\langle\!\langle c\text{^}l(a) \rangle\!\rangle \triangleq \langle\!\langle c \rangle\!\rangle.l(\langle\!\langle a \rangle\!\rangle)$

$\langle\!\langle \mathbf{typecase} \ a \ \mathbf{when} \ (x{:}A)b_1 \ \mathbf{else} \ b_2 \ \mathbf{end} \rangle\!\rangle \triangleq typecase \ \langle\!\langle a \rangle\!\rangle \mid (x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b_1 \rangle\!\rangle \mid \langle\!\langle b_2 \rangle\!\rangle$

Usefulness of the Translation

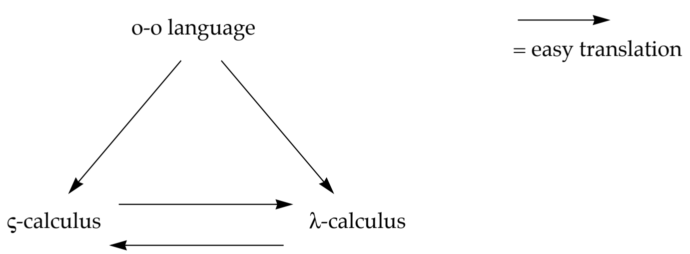
- The translation validates the typing rules of O–1. That is, if E ⊢ J is valid in O–1, then 《E ⊢ J》 is valid in the object calculus.
- The translation served as an important guide in finding sound typing rules for O–1, and for "tweaking" them to make them both simpler and more general.
- In particular, typing rules for subclasses are so inherently complex that it is difficult to "guess" them correctly without the aid of some interpretation.
- Thus, we have succeeded in using object calculi as a platform for explaining a relatively rich object-oriented language and for validating its type rules.

TRANSLATIONS

- In order to give insight into type rules for object-oriented languages, translations must be judgment-preserving (in particular, type and subtype preserving).
- Translating object-oriented languages directly to typed λ-calculi is just too hard. Object calculi provide a good stepping stone in this process, or an alternative endpoint.
- Translating object calculi into λ-calculi means, intuitively, "programming in object-oriented style within a procedural language". This is the hard part.

Untyped Translations

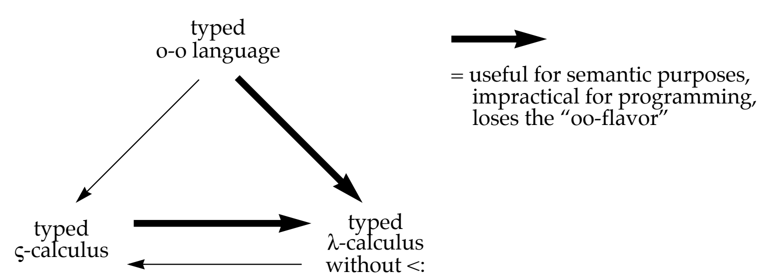
- Give insights into the nature of object-oriented computation.
- Objects = records of functions.



Type-Preserving Translations

- Give insights into the nature of object-oriented typing and subsumption/coercion.
- Object types = recursive records-of-functions types.

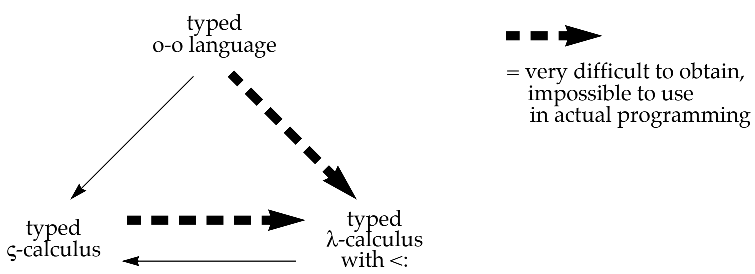
$$[l_i{:}B_i{}^{i\in 1..n}] \quad \triangleq \quad \mu(X)\langle l_i{:}X \rightarrow B_i{}^{i\in 1..n}\rangle$$



Subtype-Preserving Translations

- Give insights into the nature of subtyping for objects.
- Object types = recursive bounded existential types (!!).

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



CONCLUSIONS

- Foundations
  - ~ Subtype-preserving translations of object calculi into λ-calculi are hard.
  - ~ In contrast, subtype-preserving translations of λ-calculi into object-calculi can be easily obtained.
  - ~ In this sense, object calculi are a more convenient foundation for object-oriented programming than λ-calculi.

- Language design
  - ~ Object calculi are a good basis for designing rich object-oriented type systems (including polymorphism, Self types, etc.).
  - ~ Object-oriented languages can be shown sound by fairly direct translations into object calculi.

- Other developments
  - ~ Second-order object types for Self types.
  - ~ Higher-order object types for matching.
- Potential future areas
  - ~ Typed ς-calculi should be a good simple foundation for studying object-oriented specification and verification.
  - ~ They should also give us a formal platform for studying object-oriented concurrent languages (as opposed to “ordinary” concurrent languages).

References

- http://www.research.digital.com/SRC/personal/Luca_Cardelli/TheoryOfObjects.html
- M.Abadi, L.Cardelli: **A Theory of Objects**. Springer, 1996.