A Theory of Objects

Luca Cardelli

joint work with Martín Abadi

Digital Equipment Corporation
Systems Research Center

Sydney '97

Outline

- Topic: a foundation for object-oriented languages based on object calculi.
  ~ Interesting object-oriented features.
  ~ Modeling of those features.
- Plan:

  1) Class-Based Languages

  2) Object-Based Languages

  3) Subtyping -- Advanced Features

  4) A Language with Subtyping

  5) Matching -- Advanced Features

  6) A Language with Matching

Object-Oriented Features

Easy Language Features

The early days

- Integers and floats (occasionally, also booleans and voids).
- Monomorphic arrays (Fortran).
- Monomorphic trees (Lisp).

The days of structured programming

- Product types (records in Pascal, structs in C).
- Union types (variant records in Pascal, unions in C).
- Function/procedure types (often with various restrictions).
- Recursive types (typically via pointers).

End of the easy part

- Languages with rich user-definable types (Pascal, Algol68).

Hard Language Features

Four major innovations

- Objects and Subtyping (Simula 67).
- Abstract types (CLU).
- Polymorphism (ML).
- Modules (Modula 2).

Despite much progress, nobody really knows yet how to combine all these ingredients into coherent language designs.

Confusion

These four innovations are partially overlapping and certainly interact in interesting ways. It is not clear which ones should be taken as more prominent. E.g.:

- Object-oriented languages have tried to incorporate type abstraction, polymorphism, and modularization all at once. As a result, o-o languages are (generally) a mess. Much effort has been dedicated to separating these notions back again.
- Claims have been made (at least initially) that objects can be subsumed by either higher-order functions and polymorphism (ML camp), by data abstraction (CLU camp), or by modularization (ADA camp). But later, subtyping features were adopted: ML => ML2000, CLU =>Theta, ADA => ADA'95.
- One hard fact is that full-blown polymorphism can subsume data abstraction. But this kind of polymorphism is more general than, e.g., ML's, and it is not yet clear how to handle it in practice.
- Modules can be used to obtain some form of polymorphism and data abstraction (ADA generics, C++ templates) (Modula 2 opaque types), but not in full generality.

O-O Programming

- Goals
  - ~ Data (state) abstraction.
  - ~ Polymorphism.
  - ~ Code reuse.

- Mechanisms
  - ~ Objects with *self* (packages of data and code).
  - ~ Subtyping and subsumption.
  - ~ Classes and inheritance.

Objects

- Objects and object types
- Objects are packages of data (*instance variables*) and code (*methods*).
- Object types describe the shape of objects.

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

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

where $a : A$ means that the program a has type A. So, *myCell* : *CellType*.

Classes

- Classes are ways of describing and generating collections of objects of some type.

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

var myCell: CellType := new cell;

procedure double(aCell: CellType) is
    aCell.set(2 * aCell.get());
end;
```

Subtypes

- Subtypes can be formed by extension (of interface) from other types.

> **ObjectType** *ReCellType;*
>   **var** *contents*: *Integer*;
>   **var** *backup*: *Integer*;
>   **method** *get*(): *Integer***;**
>   **method** *set*(*n*: *Integer*);
>   **method** *restore*();
> **end**;

ReCellType is a subtype of *CellType*: an object of type *ReCellType* can be used in place of an object of type *CellType*.

Subclasses

- Subclasses are ways of describing classes incrementally, reusing code.

```
subclass reCell of cell for ReCellType 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;
```

(Inherited:
var *contents*
method *get*)

Subtyping and subsumption

- Subtyping relation, $A <: B$

  An object type is a subtype of any object type with fewer components.

  (e.g.: $\mathit{ReCellType} \quad <: \quad \mathit{CellType}$)

- Subsumption rule

  if $a : A$ and $A <: B$ then $a : B$

  (e.g.: $\mathit{myReCell} : \mathit{CellType}$)

- Subclass rule

  The type of the objects generated by a subclass
  is a subtype of the type of the objects generated by a superclass.

  c can be a subclass of d only if $\mathit{cType} <: \mathit{dType}$

  (e.g.: *reCell* can indeed be declared as a subclass of *cell*)

A Touch of Skepticism

- Object-oriented languages have been plagued, possibly more than languages of any other kind, by confusion and unsoundness.
- How do we keep track of the interactions of the numerous object-oriented features?
- How can we be sure that they all make sense, and that their interactions make sense?

Why Objects?

- Who needs object-oriented languages, anyway?
  - ~ Systems may be modeled by other paradigms.
  - ~ Data abstraction can be achieved with plain abstract data types.
  - ~ Reuse can be achieved by parameterization and modularization.
- Still, the object-oriented approach has been uniquely successful:
  - ~ Some of its features are not easy to explain as the union of well-understood concepts.
  - ~ It seems to integrate good design and implementation techniques in an intuitive framework.

Foundations

- 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 procedural languages.
- *A Theory of Objects* develops a calculus of objects, analogous to the λ-calculus but independent.
  - ~ The calculus is entirely based on objects, not on functions.
  - ~ The calculus is useful 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.

Class-Based Languages

- Mainstream object-oriented languages are class-based.
- Some of them are Simula, Smalltalk, C++, Modula-3, and Java.
- Class-based constructs vary significantly across languages.
- We cover only core features.

Basic Characteristics

- In the simplest class-based languages, there is no clear distinction
  - ~ between classes and object types,
  - ~ between subclasses and subtypes.
- Objects are generated from classes.

  We write *InstanceTypeOf*(c) for the type of objects generated from class c.
- The typical operations on objects are:
  - ~ creation,
  - ~ field selection and update,
  - ~ method invocation.
- Class definitions are often incremental: a new class may inherit structure and code from one or multiple existing classes.

Classes and Objects

- Classes are descriptions of objects.

- Example: storage cells.

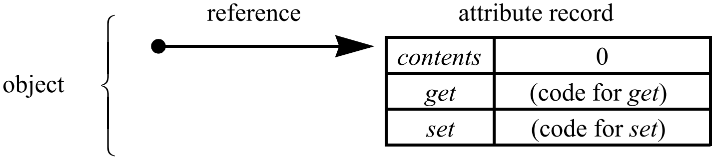
```
class cell is
    var contents: Integer := 0;
    method get(): Integer is
        return self.contents;
    end;
    method set(n: Integer) is
        self.contents := n;
    end;
end;
```

- 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*.

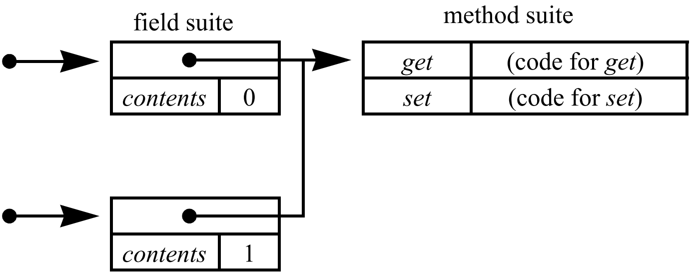
```
var myCell: InstanceTypeOf(cell) := new cell;
```

- Field selection.
- Field update.
- Method invocation.

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

The Method-Suites Storage Model

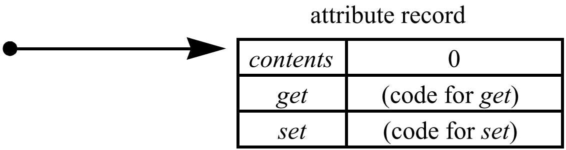
- A more refined storage model for class-based languages.



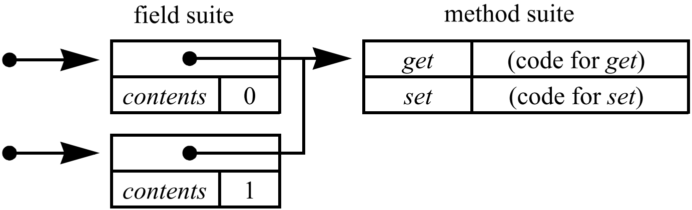
Method suites

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.



Comparison of Storage Models

- 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 $o.m(\ldots)$. 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.
  - ~ Method lookup and field selection look similar ($o.x$ and $o.m(\ldots)$).
  - ~ Features that would distinguish embedding from delegation implementations (e.g., method update) are usually avoided.

  This hides the details of the storage model.
- Self is always the *receiver*: the object that *appears* to contain the method being invoked.

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.

aCell $\bullet\!\longrightarrow$

attribute record

contents	0
get	(code for *get*)
set	(code for *set*)

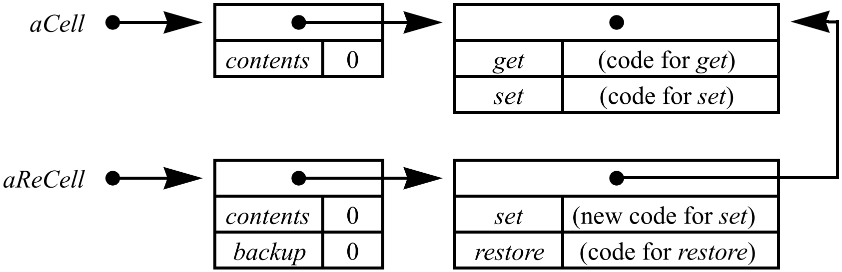
aReCell $\bullet\!\longrightarrow$

attribute record

contents	0
get	(code for *get*)
set	(code for *set*)
backup	0
restore	(code for *restore*)

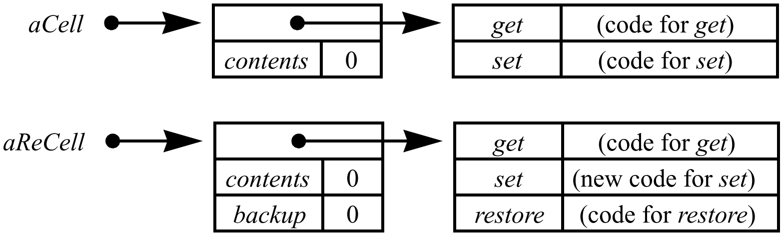
Subclasses and Method Suites

- 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).

Subclasses and Type Compatibility

- Subclasses are not just a mechanism to avoid rewriting definitions.Consider the following code fragments:

  var *myCell*: *InstanceTypeOf*(*cell*) := **new** *cell*;
  var *myReCell*: *InstanceTypeOf*(*reCell*) := **new** *reCell*;
  procedure *f*(*x*: *InstanceTypeOf*(*cell*)) **is** … **end**;

  myCell := *myReCell*;

  f(*myReCell*);

  ~ An instance of *reCell* is assigned to a variable holding instances of *cell*.

  ~ An instance of *reCell* is passed to a procedure *f* that expects instances of *cell*.

- Both code fragments would be illegal in Pascal, since *InstanceTypeOf*(*cell*) and *InstanceTypeOf*(*reCell*) do not match.

Polymorphism

- In object-oriented languages these code fragments are made legal by the following rule, which embodies what is often called (*subtype*) *polymorphism*:

If *c'* is a subclass of *c*, and *o'* is an instance of *c'*, then *o'* is an instance of *c*.

or, from the point of view of the typechecker:

If *c'* is a subclass of *c*, and *o'*: *InstanceTypeOf*(*c'*), then *o'*: *InstanceTypeOf*(*c*).

The Subtype Relation

If c' is a subclass of c, and o': *InstanceTypeOf*(c'),
then o': *InstanceTypeOf*(c).

- We analyze this further, by a reflexive and transitive subtype relation (<:) between *InstanceTypeOf* types.
  - ~ This subtype relation is intended, intuitively, as set inclusion between sets of values.
  - ~ For now we do not define the subtype relation precisely, but we assume that it satisfies two properties:

- If $a : A$, and $A <: B$, then $a : B$.
- *InstanceTypeOf*(c') <: *InstanceTypeOf*(c) if and only if c' is a subclass of c.

The Subtype Relation: Subsumption

If $a : A$, and $A <: B$, then $a : B$.

- This property, called *subsumption*, is the characteristic property of subtype relations.
  - ~ A value of type A can be viewed as a value of a type B.
  - ~ We say that the value is *subsumed* from type A to type B.

Subclassing is Subtyping

InstanceTypeOf(*c'*) <: *InstanceTypeOf*(*c*)
if and only if *c'* is a subclass of *c*.

- This property, which we may call *subclassing-is-subtyping*, is the characteristic of classical class-based languages.
  - ~ Since inheritance is connected with subclassing, we may read this as an inheritance-is-subtyping property.
  - ~ More recent class-based languages adopt a different, inheritance-is-*not*-subtyping approach.

Static versus Dynamic Dispatch

- With the introduction of subsumption, we have to reexamine the meaning of method invocation. For example, given the code:

  procedure *g*(*x*: *InstanceTypeOf*(*cell*)) **is**
      x.*set*(3);
  end;
  g(*myReCell*);

  we should determine what is the meaning of *x*.*set*(3) during the invocation of *g*.

- The declared type of *x* is *InstanceTypeOf*(*cell*), while its value is *myReCell*, which is an instance of *reCell*.

- Since *set* is overridden in *reCell*, there are two possibilities:

Static dispatch:	*x.set*(3) runs the code of *set* from *cell*
Dynamic dispatch:	*x.set*(3) runs the code of *set* from *reCell*

  - ~ Static dispatch is based on the compile-time type information available for *x*.
  - ~ Dynamic dispatch is based on the run-time value of *x*.

- We may say that *InstanceTypeOf*(*reCell*) is the *true type* of *x* during the execution of *g*(*myReCell*), and that the true type determines the choice of method.

- Dynamic dispatch is found in all object-oriented languages, to the point that it can be regarded as one of their defining properties.
- Dynamic dispatch is an important component of object abstraction.
  - Each object knows how to behave autonomously.
  - So the context does not need to examine the object and decide which operation to apply.

- A consequence of dynamic dispatch is that subsumption should have no run-time effect on objects.
  - For example, if subsumption from *InstanceTypeOf*(*reCell*) to *InstanceTypeOf*(*cell*) coerced a *reCell* to a *cell* by cutting *backup* and *restore*, then a dynamically dispatched invocation of *set* would fail.
  - The fact that subsumption has no run-time effect is both good for efficiency and semantically necessary.

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.

Self-Mutation

- Restorable cells with no *backup* field.

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

- The *set* method updates the *restore* method!

```
object reCell: ReCell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is
        let x = self.get();
        self.restore := method () is self.contents := x end;
        self.contents := n;
    end;
    method restore() is self.contents := 0 end;
end;
```

Forms of Mutation

- Method update is an example of a mutation operation. It is simple and statically typable.
- Forms of mutation include:
  - Direct method update (Beta, NewtonScript, Obliq, Kevo, Garnet).
  - Dynamically removing and adding attributes (Self, Act1).
  - Swapping groups of methods (Self, Ellie).

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.

Implicit vs. Explicit Inheritance

- A difference in declaration.
- **Implicit inheritance**: one or more objects are designated as the donors (explicitly!), and their attributes are <u>implicitly</u> inherited.
- **Explicit inheritance**, individual attributes of one or more donors are <u>explicitly</u> designated and inherited.
- **Super** and **override** make sense for implicit inheritance, not for explicit inheritance.

- Intermediate possibility: explicitly designate a named collection of attributes that, however, does not form a whole object. E.g. *mixin* inheritance.
- (We can see implicit and explicit inheritance, as the extreme points of a spectrum.)

Embedding vs. Delegation Inheritance

- A difference in execution.
- **Embedding inheritance**: the attributes inherited from a donor become part of the host (in principle, at least).
- **Delegation inheritance**: the inherited attributes remain part of the donor, and are accessed via an indirection from the host.
- Either way, self is the receiver.
- In embedding, host objects are independent of their donors. In delegation, complex webs of dependencies may be created.

Embedding

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

aCell →

contents	0
get	(code for *get*)
set	(code for *set*)

aReCell →

contents	0
backup	0
get	(new code for *get*)
set	(new code for *set*)
restore	(code for *restore*)

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).

Embedding-Based Inheritance

- Embedding inheritance can be specified explicitly or implicitly.
  - Explicit forms of embedding inheritance can be understood as *reassembling* parts of old objects into new objects.
  - Implicit forms of embedding inheritance can be understood as ways of *concatenating* or extending copies of existing objects with new attributes.

Explicit Embedding Inheritance

- Individual methods and fields of specific objects (donors) are copied into new objects (hosts).
- We write

  embed $o.m(\ldots)$

  to embed the method m of object o into the current object.
- The meaning of **embed** $cell.set(n)$ is to execute the set method of $cell$ with **self** bound to the current self, and not with **self** bound to $cell$ as in a normal invocation $cell.set(n)$.
- Moreover, the code of set is embedded in $reCellExp$.

reCellExp

```
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;

object reCellExp: ReCell is
    var contents: Integer := cell.contents;
    var backup: Integer := 0;
    method get(): Integer is
        return embed cell.get();
    end;
    method set(n: Integer) is
        self.backup := self.contents;
        embed cell.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

- The code for get could be abbreviated to:

```
method get copied from cell;
```

Implicit Embedding Inheritance

- Whole objects (donors) are copied to form new objects (hosts).
- We write

  object *o*: *T* **extends** *o'*

  to designate a donor object *o'* for *o*.
- As a consequence of this declaration, *o* is an object containing a copy of the attributes of *o'*, with independent state.

reCellImp

```
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;

object reCellImp: ReCell extends cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        embed super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

Alternate *reCellImp* via method update

- We could define an equivalent object by a pure extension of *cell* followed by a method update.

```
object reCellImp1: ReCell extends cell is
    var backup: Integer := 0;
    method restore() is self.contents := self.backup end;
end;

reCellImp1.set :=
    method (n: Integer) is
        self.backup := self.contents;
        self.contents := n;
    end;
```

This code works because, with embedding, method update affects only the object to which it is applied. (This is not true for delegation.)

Stand-alone *reCell*

- The definitions of both *reCellImp* and *reCellExp* can be seen as convenient abbreviations:

```
object reCell: ReCell is
    var contents: Integer := 0;
    var backup: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is
        self.backup := self.contents;
        self.contents := n;
    end;
    method restore() is self.contents := self.backup end;
end;
```

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, in such a way that an object can be seen as an extension of another.
- Note: similar to hierarchical method suites.

Delegation and Self

- A crucial aspect of delegation inheritance is the interaction of donor links with the binding of **self**.
- On an invocation of a method called *m*, the code for *m* may be found only in the donor cell. But the occurrences of **self** within the code of *m* refer to the original receiver, not to the donor.
- Therefore, delegation is not redirected invocation.

Implicit Delegation Inheritance (Traditional Delegation)

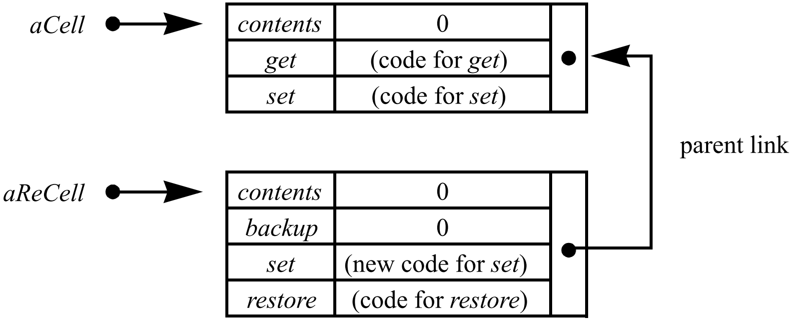
- Whole objects (donors/parents) are shared to from new objects (hosts/children).

- We write

  object *o*: *T* **child of** *o'*

  to designate a parent object *o'* for *o*.

- As a consequence of this declaration, *o* is an object containing a single *parent link* to *o'*, with parent state shared among children. Parent links are followed in the search for attributes.



(Single-parent) Delegation

reCellImp

- A first attempt.

```
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;

object reCellImp’: ReCell child of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        delegate super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

- This is almost identical to the code of *reCellImp* for embedding.
- But for delegation, this definition is wrong: the *contents* field is shared by all the children.

- A proper definition must include a local copy of the *contents* field, overriding the *contents* field of the parent.

```
object reCellImp: ReCell child of cell is
    override contents: Integer := cell.contents;
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        delegate super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

- On an invocation of *reCellImp*.*get*(), the *get* method is found only in the parent cell, but the occurrences of **self** within the code of *get* refer to the original receiver, *reCellImp*, and not to the parent, *cell*.
- Hence the result of *get*() is, as desired, the integer stored in the *contents* field of *reCellImp*, not the one in the parent cell.

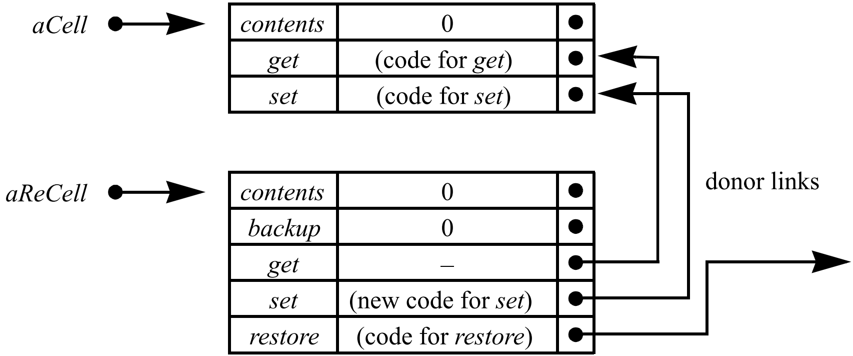
Explicit Delegation Inheritance

- Individual methods and fields of specific objects (donors) are linked into new objects (hosts).
- We write

  delegate $o.m(\ldots)$

  to execute the m method of o with **self** bound to the current self (not to o).
- The difference between **delegate** and **embed** is that the former obtains the method from the donor at the time of method invocation, while the latter obtains it earlier, at the time of object creation.

.



(An example of) Delegation

reCellExp

```
object reCellExp: ReCell is
    var contents: Integer := cell.contents;
    var backup: Integer := 0;
    method get(): Integer is return delegate cell.get() end;
    method set(n: Integer) is
        self.backup := self.contents;
        delegate cell.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

- Explicit delegation provides a clean way of delegating operations to multiple objects. It provides a clean semantics for multiple donors.

Dynamic Inheritance

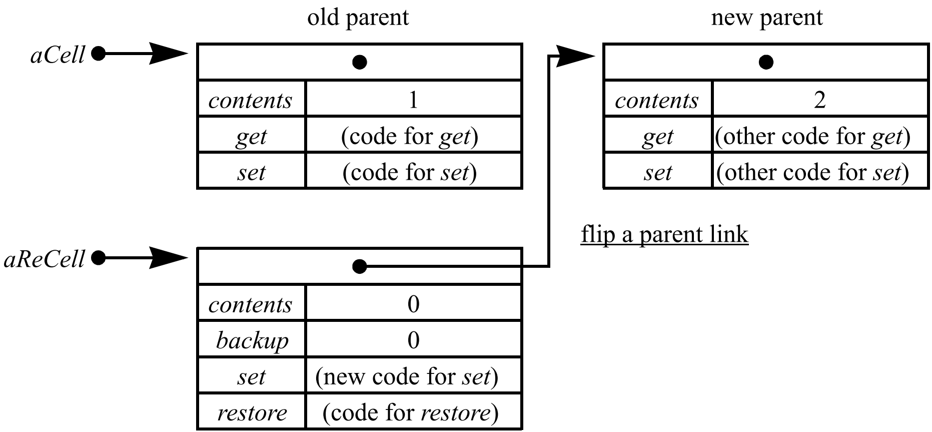
- Inheritance is called *static* when inherited attributes are fixed for all time.
- It is *dynamic* when the collection of inherited attributes can be updated dynamically (replaced, increased, decreased).

Mode Switching

- Although dynamic inheritance is in general a dangerous feature, it enables rather elegant and disciplined programming techniques.

- In particular, *mode-switching* is the special case of dynamic inheritance where a collection of (inherited) attributes is *swapped* with a similar collection of attributes. (This is even typable.)

Delegation-Style Mode Switching



Reparenting

Embedding-Style Mode Switching

<u>flip a set of attributes</u>

old object

contents	0
backup	0
get	(code for *get*)
set	(code for *set*)
restore	(code for *restore*)

new object

contents	0
backup	0
get	(other code for *get*)
set	(other code for *set*)
restore	(code for *restore*)

Method Update

Embedding vs. Delegation Summary

- In embedding inheritance, a freshly created host object contains copies of donor attributes.
- Access to the inherited donor attributes is no different than access to original attributes, and is quick.
- Storage use may be comparatively large, unless optimizations are used.

- In delegation inheritance, a host object contains links to external donor objects.
- During method invocation, the attribute-lookup procedure must preserve the binding of **self** to the original receiver, even while following the donor links.
  - ~ This results in more complicated implementation and formal modeling of method lookup.
  - ~ It creates couplings between objects that may not be desirable in certain (e.g. distributed) situations.

- In class-based languages the embedding and delegation models are normally (mostly) equivalent.
- In object-based languages they are distinguishable.
  - In delegation, donors may contain fields, which may be updated; the changes are seen by the inheriting hosts.
  - Similarly, the methods of a donor may be updated, and again the changes are seen by the inheriting hosts.

- It is often permitted to replace a donor link with another one in an object; then all the inheritors of that object may change behavior.
- Cloning is still taken to perform shallow copies of objects, without copying the corresponding donors. Thus, all clones of an object come to share its donors and therefore the mutable fields and methods of the donors.

- Thus, embedding and delegation are two fundamentally distinct ways of achieving inheritance with prototypes.
- Interesting languages exist that explore both possibilities.

Advantages of Delegation

- Space efficiency by sharing.
- Convenience in performing dynamic, pervasive changes to all inheritors of an object.
- Well suited for integrated languages/environments.

Advantages of Embedding

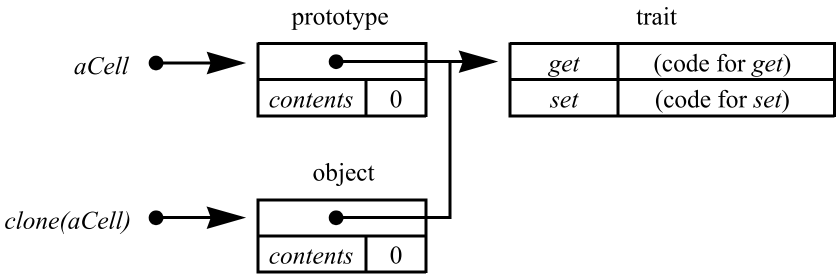
- Delegation can be criticized because it creates dynamic webs of dependencies that lead to fragile systems. Embedding is not affected by this problem since objects remain autonomous.
- In embedding-based languages such as Kevo and Omega, pervasive changes are achieved even without donor hierarchies.
- Space efficiency, while essential, is best achieved behind the scenes of the implementation.
  - ~ Even delegation-based languages optimize cloning operations by transparently sharing structures; the same techniques can be used to optimize space in embedding-based languages.

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.

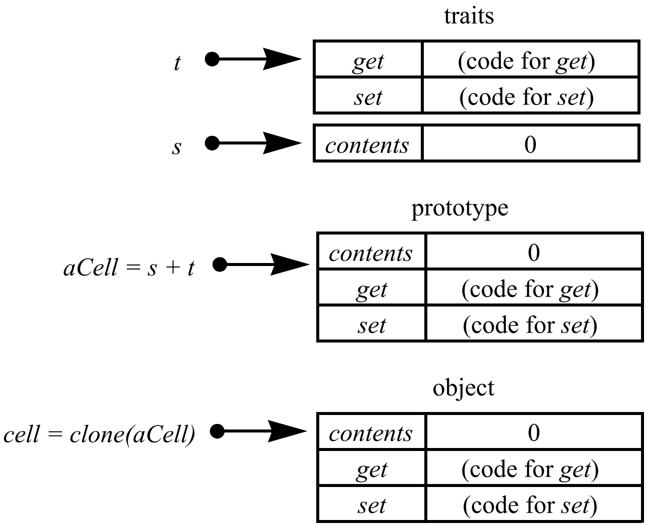
Different Kinds of Objects

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



Traits

Embedding-Style Traits



Traits

Traits are not Prototypes

- In the spirit of classless languages, traits and prototypes are still ordinary objects. But there are distinctions:
  - ~ Traits are intended only as the shared parents of normal objects: they should not be used directly or cloned.
  - ~ Prototypes are intended only as object (and prototype) generators via cloning: they should not be used directly or modified.
  - ~ Normal objects are intended only to be used and to carry local state: they should rely on traits for their methods.
- These distinctions may be methodological or enforced: some operations on traits and prototypes may be forbidden to protect them from accidental damage.

Trait Treason

- 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.

Object Constructions vs. Class Implementations

- The traits-prototypes partition in delegation-based languages looks exactly like an implementation technique for classes.
- A similar traits-prototypes partition in embedding-based languages corresponds to a different implementation technique for classes that trades space for access speed.
- Class-based notions and techniques are not totally banned in object-based languages. Rather, they resurface naturally.

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.

Future Directions

- I look forward to the continued development of typed object-based languages.
  - ~ The notion of object type arise more naturally in object-based languages.
  - ~ Traits, method update, and mode switching are typable (general reparenting is not easily typable).
- No need for dichotomy: object-based and class-based features can be merged within a single language, based on the common object-based semantics (Beta, O–1, O–2, O–3).

- Embedding-based languages seem to be a natural fit for distributed-objects situations. E.g. COM vs. CORBA.
  - Objects are self-contained and are therefore *localized*.
  - For this reason, Obliq was designed as an embedding-based language.

Advanced Subtyping Issues

Covariance

- The type $A\times B$ is the type of pairs with left component of type A and right component of type B. The operations $fst(c)$ and $snd(c)$ extract the left and right components, respectively, of an element c of type $A\times B$.
- We say that $\times$ is a *covariant* operator (in both arguments), because $A\times B$ varies in the same sense as A or B:

  $A\times B <: A'\times B'$ provided that $A<:A'$ and $B<:B'$

  We can justify this property as follows:

Argument for the covariance of $A\times B$

A pair $\langle a,b\rangle$ with left component a of type A and right component b of type B, has type $A\times B$. If $A<:A'$ and $B<:B'$, then by subsumption we have $a:A'$ and $b:B'$, so that $\langle a,b\rangle$ has also type $A'\times B'$. Therefore, any pair of type $A\times B$ has also type $A'\times B'$ whenever $A<:A'$ and $B<:B'$. In other words, the inclusion $A\times B <: A'\times B'$ between product types is valid whenever $A<:A'$ and $B<:B'$.

Contravariance

- The type $A \rightarrow B$ is the type of functions with argument type A and result type B.
- We say that $\rightarrow$ is a *contravariant* operator in its left argument, because $A \rightarrow B$ varies in the opposite sense as A; the right argument is instead covariant:

$$A \rightarrow B <: A' \rightarrow B' \text{ provided that } A' <: A \text{ and } B <: B'$$

Argument for the co/contravariance of $A \rightarrow B$

If $B <: B'$, then a function f of type $A \rightarrow B$ produces results of type B' by subsumption. If $A' <: A$, then f accepts also arguments of type A', since these have type A by subsumption. Therefore, every function of type $A \rightarrow B$ has also type $A' \rightarrow B'$ whenever $A' <: A$ and $B <: B'$. In other words, the inclusion $A \rightarrow B <: A' \rightarrow B'$ between function types is valid whenever $A' <: A$ and $B <: B'$.

- In the case of functions of multiple arguments, for example of type $(A_1 \times A_2) \rightarrow B$, we have contravariance in both A_1 and A_2. This is because product, which is covariant in both of its arguments, is found in a contravariant context.

Invariance

- Consider pairs whose components can be updated; we indicate their type by $A※B$. Given $p:A※B$, $a:A$, and $b:B$, we have operations $getLft(p):A$ and $getRht(p):B$ that extract components, and operations $setLft(p,a)$ and $setRht(p,b)$ that destructively update components.
- The operator ※ does not enjoy any covariance or contravariance properties:

$$A※B <: A'※B' \text{ provided that } A=A' \text{ and } B=B'$$

  We say that ※ is an *invariant* operator (in both of its arguments).

Argument for the invariance of $A※B$

If $A<:A'$ and $B<:B'$, can we covariantly allow $A※B <: A'※B'$? If we adopt this inclusion, then from $p:A※B$ we obtain $p:A'※B'$, and we can perform $setLft(p,a')$, for any $a':A'$. After that, $getLft(p)$ might return an element of type A' that is not an element of type A. Hence, the inclusion $A※B <: A'※B'$ is not sound.

Conversely, if $A''<:A$ and $B''<:B$, can we contravariantly allow $A※B <: A''※B''$? From $p:A※B$ we now obtain $p:A''※B''$, and we can incorrectly deduce that $getLft(p):A''$. Hence, the inclusion $A※B <: A''※B''$ is not sound either.

Method Specialization

In the simplest approach to overriding, an overriding method must have the same type as the overridden method.

- This condition can be relaxed to allow *method specialization*:

  An overriding method may adopt different argument and result types, specialized for the subclass.

- We still do not allow overriding and specialization of field types.

  Fields are updatable, like the components of the type $A \divideontimes B$, and therefore their types must be invariant.

Suppose we use different argument and result types, *A'* and *B'*, when overriding *m*:

> **class** *c* **is**
>
>   **method** $m(x: A): B$ **is** … **end**;
>
>   **method** $m_1(x_1: A_1): B_1$ **is** … **end**;
>
> **end**;
>
> **subclass** *c'* **of** *c* **is**
>
>   **override** $m(x: A'): B'$ **is** … **end**;
>
> **end**;

- We are constrained by subsumption between *InstanceTypeOf*(*c'*) and *InstanceTypeOf*(*c*).

Specialization on Override

```
class c is
    method m(x: A): B is … end;
    method m1(x1: A1): B1 is … end;
end;

subclass c' of c is
    override m(x: A'): B' is … end;
end;
```

- When *o'* of *InstanceTypeOf*(*c'*) is subsumed into *InstanceTypeOf*(*c*) and *o'*.*m*(*a*) is invoked, the argument may have static type *A* and the result must have static type *B*.
- Therefore, it is sufficient to require that $B' <: B$ (covariantly) and that $A <: A'$ (contravariantly).
- This is called *method specialization on override*. The result type *B* is specialized to *B'*, while the parameter type *A* is generalized to *A'*.

Specialization on Inheritance

There is another form of method specialization that happens implicitly by inheritance.

- The occurrences of **self** in the methods of *c* can be considered of type *InstanceTypeOf*(*c*).
- When the methods of *c* are inherited by *c'*, the same occurrences of **self** can similarly be considered of type *InstanceTypeOf*(*c'*).
- Thus, the type of **self** is silently specialized on inheritance (covariantly!).

The Variance Controversy

It is controversial whether the types of arguments of methods should vary covariantly or contravariantly from classes to subclasses.

- The properties of the operators $\times$, $\rightarrow$, and ※ follow inevitably from our assumptions.

  The variance properties of method types follow inevitably by a similar analysis.

- We cannot take method argument types to vary covariantly, unless we change the meaning of covariance, subtyping, or subsumption.

  With our definitions, covariance of method argument types is unsound: if left unchecked, it may lead to unpredictable behavior.

- Eiffel still favors covariance of method arguments. Unsound behavior is supposed to be caught by global flow analysis.

- Covariance can be soundly adopted for multiple dispatch, but using a different set of type operators.

Self Type Specialization

Class definitions are often recursive, in the sense that the definition of a class *c* may contain occurrences of *InstanceTypeOf*(*c*).

For example, we could have a class *c* containing a method *m* with result type *InstanceTypeOf*(*c*):

> **class** *c* **is**
>
>   **var** *x*: *Integer* := 0;
>
>   **method** *m*(): *InstanceTypeOf*(*c*) **is** … **self** … **end**;
>
> **end**;
>
> **subclass** *c'* **of** *c* **is**
>
>   **var** *y*: *Integer* := 0;
>
> **end**;

On inheritance, recursive types are, by default, preserved exactly, just like other types.

- For instance, for *o'* of class *c'*, we have that *o'.m*() has type *InstanceTypeOf*(*c*) and not, for example, *InstanceTypeOf*(*c'*).
- In general, adopting *InstanceTypeOf*(*c'*) as the result type for the inherited method *m* in *c'* is unsound, because *m* may construct and return an instance of *c* that is not an instance of *c'*.

Suppose, though, that *m* returns **self**, perhaps after field updates.

- Then it would be sound to give the inherited method the result type *InstanceTypeOf*(*c'*).
- With this more precise typing, we avoid later uses of **typecase**.
- Limiting the result type to *InstanceTypeOf*(*c*) constitutes an unwarranted loss of information.

The Type Self

This argument leads to the notion of *Self types*.

- The keyword **Self** represents the type of **self**.
- Instead of giving the result type *InstanceTypeOf*(c) to m, we write:

class *c* **is**
    var *x*: *Integer* := 0;
    method *m*(): **Self is** … **self** … **end**;
end;

- The typing of the code of m relies on the assumptions that **Self** is a subtype of *InstanceTypeOf*(c), and that **self** has type **Self**.
- When *c'* is declared as a subclass of c, the result type of m is still taken to be **Self**.

  Thus **Self** is automatically specialized on subclassing.

Variance of the Type Self

- There are no drawbacks to extending classical class-based languages with **Self** as the result type of methods.
  - ~ We can even allow **Self** as the type of fields.
  - ~ These extensions prevent loss of type information at no cost other than keeping track of the type of **self**.

  (See Eiffel and Sather.)

- A natural next step is to allow **Self** in contravariant (argument) positions.
  - ~ This is what Eiffel set out to do (with some trouble).
  - ~ The proper handling of **Self** in contravariant positions is a new development in class-based languages.

Inheritance, Subclassing, Subtyping

One central characteristic of classical class-based languages is the strict correlation between inheritance, subclassing, and subtyping.

- A great economy of concepts and syntax is achieved by identifying these three relations.
- But here are situations in which inheritance, subclassing, and subtyping conflict.

  Opportunities for code reuse are then limited.

Therefore, there has been an effort to separate these relations.

- The separation of subclassing and subtyping is now common.
- Other separations are more tentative.

Object Types

In the original formulation of classes (in Simula, for example), the type description of objects is mixed with their implementation.

This conflicts with separating specifications from implementations.

Separation between specifications and implementations can be achieved by introducing types for objects.

- Object types are independent of specific classes.
- Object types list attributes and their types,

  but not their implementations.
- They are suitable to appear in interfaces, and to be implemented separately and in more than one way.

  (In Java, object types are in fact called interfaces.)

Recall the classes *cell* and *reCell*:

```
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;

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;
```

We introduce two object types *Cell* and *ReCell* that correspond to these classes.

- We write them as separate types (but we could introduce syntax to avoid repeating common components).

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

ObjectType *ReCell* **is**
    var *contents*: *Integer*;
    var *backup*: *Integer*;
    method *get*(): *Integer*;
    method *set*(*n*: *Integer*);
    method *restore*();
end;

- We may still use *ObjectTypeOf*(*cell*) as a meta-notation for the object type *Cell*.
  - ~ This type can be mechanically extracted from class *cell*.
  - ~ Therefore, we may write either o: *ObjectTypeOf*(*cell*) or o: *Cell*.
- The main property we expect of *ObjectTypeOf* is that:

$$\textbf{new } c : ObjectTypeOf(c) \qquad\qquad \text{for any class } c$$

- Different classes *cell* and $cell_1$ may happen to produce the same object type *Cell*, equal to *ObjectTypeOf*(*cell*) and *ObjectTypeOf*($cell_1$).
- Therefore, objects having type *Cell* are required only to satisfy a certain protocol, independently of attribute implementation.

Subtyping without Subclassing

When object types are independent of classes, we must provide an independent definition of subtyping.

- There are several choices at this point:
  - whether subtyping is determined by type structure or by type names in declarations,
  - in the former case, what parts of the structure of types matter.
- We will use a particularly simple form of structural subtyping.

  We assume, for two object types O and O', that:

  $$O' <: O \qquad \text{if } O' \text{ has all the components that } O \text{ has}$$

  where a component of an object type is the name of a field or a method and its associated type.

  So, for example, $ReCell <: Cell$.

Multiple Subtyping

With this definition of subtyping, object types naturally support multiple subtyping, because components are assumed unordered.

For example, consider the object type:

> **ObjectType** *ReInteger* **is**
>   **var** *contents*: *Integer*;
>   **var** *backup*: *Integer*;
>   **method** *restore*();
> **end**;

Then we have both *ReCell* <: *Cell* and *ReCell* <: *ReInteger*.

Subclassing Implies Subtyping (Still)

With the new definition of subtyping we have:

> If *c'* is a subclass of *c* then *ObjectTypeOf*(*c'*) <: *ObjectTypeOf*(*c*).

- This holds simply because a subclass can only add new attributes to a class, and because we require that overriding methods preserve the existing method types.
- Therefore, we have partially decoupled subclassing from subtyping, since subtyping does not imply subclassing. Subclassing still implies subtyping, so all the previous uses of subsumption are still allowed. But, since subsumption is based on subtyping and not subclassing, we now have even more freedom in subsumption.
- In conclusion, the notion of subclassing-is-subtyping can be weakened to subclassing-implies-subtyping without loss of expressiveness, and with a gain in separation between interfaces and implementations.

Subclassing without Subtyping

- We have seen how the partial decoupling of subtyping from subclassing increases the opportunities for subsumption.
- Another approach has emerged that increases the potential for inheritance by further separating subtyping from subclassing. This approach abandons completely the notion that subclassing implies subtyping, and is known under the name *inheritance-is-not-subtyping*.
- It is largely motivated by the desire to handle contravariant (argument) occurrences of **Self** so as to allow inheritance of methods with arguments of type **Self**; these methods arise naturally in realistic examples.
- The price paid for this added flexibility in inheritance is decreased flexibility in subsumption. When **Self** is used liberally in contravariant positions, subclasses do not necessarily induce subtypes.

- Consider two types *Max* and *MinMax* for integers enriched with *min* and *max* methods. Each of these types is defined recursively:

> **ObjectType** *Max* **is**
>
>   **var** *n*: *Integer*;
>
>   **method** *max*(*other*: *Max*): *Max*;
>
> **end**;
>
> **ObjectType** *MinMax* **is**
>
>   **var** *n*: *Integer*;
>
>   **method** *max*(*other*: *MinMax*): *MinMax*;
>
>   **method** *min*(*other*: *MinMax*): *MinMax*;
>
> **end**;

- Consider also two classes:

```
class maxClass is
    var n: Integer := 0;
    method max(other: Self): Self is
        if self.n>other.n then return self else return other end;
    end;
end;

subclass minMaxClass of maxClass is
    method min(other: Self): Self is
        if self.n<other.n then return self else return other end;
    end;
end;
```

- The methods *min* and *max* are called *binary* because they operate on two objects: **self** and *other*; the type of *other* is given by a contravariant occurrence of **Self**. Notice that the method *max*, which has an argument of type **Self**, is inherited from *maxClass* to *minMaxClass*.

- Intuitively the type *Max* corresponds to the class *maxClass*, and *MinMax* to *minMaxClass*. To make this correspondence more precise, we must define the meaning of *ObjectTypeOf* for classes containing occurrences of **Self**, so as to obtain *ObjectType-Of*(*maxClass*) = *Max* and *ObjectTypeOf*(*minMaxClass*) = *MinMax*.
- For these equations to hold, we map the use of **Self** in a class to the use of recursion in an object type. We also implicitly specialize **Self** for inherited methods; for example, we map the use of **Self** in the inherited method *max* to *MinMax*. In short, we obtain that any instance of *maxClass* has type *Max*, and any instance of *minMaxClass* has type *MinMax*.

- Although *minMaxClass* is a subclass of *maxClass*, *MinMax* cannot be a subtype of *Max*. Consider the class:

> **subclass** *minMaxClass'* **of** *minMaxClass* **is**
>   **override** *max*(*other*: **Self**): **Self is**
>   **if** *other*.*min*(**self**)=*other* **then return self else return** *other* **end**;
>   **end**;
> **end**;

- For any instance *mm'* of *minMaxClass'* we have *mm'*:*MinMax*. If *MinMax* were a subtype of *Max*, then we would have also *mm'*:*Max*, and *mm'*.*max*(*m*) would be allowed for any *m* of type *Max*. Since *m* may not have a *min* attribute, the overridden *max* method of *mm'* may break. Therefore:

  MinMax <: *Max* does not hold

- Thus, subclasses with contravariant occurrences of **Self** do not always induce subtypes.

Type Parameters

- Type parameterization is a general technique for reusing the same piece of code at different types. It is becoming common in modern object-oriented languages, partially independently of object-oriented features.
- In conjunction with subtyping, type parameterization can be used to remedy some typing difficulties due to contravariance, for example in method specialization.
- Consider the following object types, where *Vegetables* <: *Food* (but not vice versa):

> **ObjectType** *Person* **is**
>
> > …
> >
> > **method** *eat*(*food*: *Food*);
>
> **end**;
>
> **ObjectType** *Vegetarian* **is**
>
> > …
> >
> > **method** *eat*(*food*: *Vegetables*);
>
> **end**;

- The intention is that a vegetarian is a person, so we would expect $Vegetarian <: Person$.
- However, this inclusion cannot hold because of the contravariance on the argument of the *eat* method. If we erroneously assume $Vegetarian <: Person$, then a vegetarian can be subsumed into *Person*, and can be made to eat meat.
- We can obtain some legal subsumptions between vegetarians and persons by converting the corresponding object types into type operators parameterized on the type of *food*:

ObjectOperator *PersonEating*[$F <: Food$] **is**

…

method *eat*(*food*: *F*);

end;

ObjectOperator *VegetarianEating*[$F <: Vegetables$] **is**

…

method *eat*(*food*: *F*);

end;

- The mechanism used here is called *bounded type parameterization*. The variable F is a type parameter, which can be instantiated with a type. A bound like $F <: \mathit{Vegetables}$ limits the possible instantiations of F to subtypes of *Vegetables*.
- So, $\mathit{VegetarianEating}[\mathit{Vegetables}]$ is a type; in contrast, $\mathit{VegetarianEating}[\mathit{Food}]$ is not well-formed. The type $\mathit{VegetarianEating}[\mathit{Vegetables}]$ is an instance of *VegetarianEating*, and is equal to the type *Vegetarian*.
- We have that:

$$\text{for all } F <: \mathit{Vegetables},\ \ \mathit{VegetarianEating}[F] <: \mathit{PersonEating}[F]$$

  because, for any $F <: \mathit{Vegetables}$, the two instances are included by the usual rules for subtyping.
- In particular, we obtain:

$$\mathit{Vegetarian} = \mathit{VegetarianEating}[\mathit{Vegetables}] <: \mathit{PersonEating}[\mathit{Vegetables}].$$

  This inclusion can be useful for subsumption: it asserts, correctly, that a vegetarian is a person that eats only vegetables.

- Related to bounded type parameters are *bounded abstract types* (also called *partially abstract types*). Bounded abstract types offer a different solution to the problem of making *Vegetarian* subtype of *Person*.
- We redefine our object types by adding the *F* parameter, subtype of *Food*, as one of the attributes:

```
ObjectType Person is
    type F <: Food;
    …
    var lunch: F;
    method eat( food: F);
end;

ObjectType Vegetarian is
    type F <: Vegetables;
    …
    var lunch: F;
    method eat( food: F);
end;
```

- The meaning of the type component *F*<:*Food* in *Person* is that, given a person, we know that it can eat some *Food*, but we do not know exactly of what kind. The *lunch* attribute provides some food that a person can eat.
- We can build an object of type *Person* by choosing a specific subtype of *Food*, for example *F*=*Dessert*, picking a dessert for the *lunch* field, and implementing a method with parameter of type *Dessert*. We have that the resulting object is a *Person*, by forgetting the specific *F* that we chose for its implementation.
- Now the inclusion *Vegetarian* <: *Person* holds. A vegetarian subsumed into *Person* can be safely fed the lunch it carries with it, because originally the vegetarian was constructed with *F*<:*Vegetables*.
- A limitation of this approach is that a person can be fed only the food it carries with it as a component of type *F*, and not some food obtained independently.

Object Protocols

- Even when subclasses do not induce subtypes, we can find a relation between the type induced by a class and the type induced by one of its subclasses. It just so happens that, unlike subtyping, this relation does not enjoy the subsumption property. We now examine this new relation between object types.
- We cannot usefully quantify over the subtypes of *Max* because of the failure of subtyping. A parametric definition like:

  ObjectOperator $P[M <: Max]$ **is** … **end**;

- is not very useful; we could instantiate P by writing $P[Max]$, but $P[MinMax]$ would not be well-formed.

- Still, any object that supports the *MinMax* protocol, in an intuitive sense, supports also the *Max* protocol. There seems to be an opportunity for some kind of *subprotocol* relation that may allow useful parameterization. In order to find this subprotocol relation, we introduce two type operators, *MaxProtocol* and *MinMaxProtocol*:

```
ObjectOperator MaxProtocol[X] is
    var n:Integer;
    method max(other: X): X;
end;

ObjectOperator MinMaxProtocol[X] is
    var n:Integer;
    method max(other: X): X;
    method min(other: X): X;
end;
```

- Generalizing from this example, we can always pass uniformly from a recursive type *T* to an operator *T-Protocol* by abstracting over the recursive occurrences of *T*. The operator *T-Protocol* is a function on types; taking the fixpoint of *T-Protocol* yields back *T*.
- We find two formal relationships between *Max* and *MinMax*. First, *MinMax* is a post-fixpoint of *MaxProtocol*, that is:

$$MinMax <: MaxProtocol[MinMax]$$

- Second, let $\prec:$ denote the higher-order subtype relation between type operators:

$$P \prec: P' \quad \text{iff} \quad P[T] <: P'[T] \quad \text{for all types } T$$

- Then, the protocols of *Max* and *MinMax* satisfy:

$$MinMaxProtocol \prec: MaxProtocol$$

- Either of these two relationships can be taken as our hypothesized notion of subprotocol:

$$S \text{ subprotocol } T \quad \text{if} \quad S <: \textit{T-Protocol}[S]$$

$$\text{or}$$

$$S \text{ subprotocol } T \quad \text{if} \quad \textit{S-Protocol} \prec: \textit{T-Protocol}$$

- The second relationship expresses a bit more directly the fact that there exists a subprotocol relation, and that this is in fact a relation between operators, not between types.
- Whenever we have some property common to several types, we may think of parameterizing over these types. So we may adopt one of the following forms of parameterization:

ObjectOperator $P_1[X <: MaxProtocol[X]]$ **is** … **end**;
ObjectOperator $P_2[P \prec: MaxProtocol]$ **is** … **end**;

- Then we can instantiate P_1 to $P_1[MinMax]$, and P_2 to $P_2[MinMaxProtocol]$.

- These two forms of parameterization seem to be equally expressive in practice. The first one is called *F-bounded parameterization*. The second form is *higher-order bounded parameterization*, defined via pointwise subtyping of type operators.
- Instead of working with type operators, a programming language supporting subprotocols may conveniently define a *matching* relation (denoted by <#) directly over types. The properties of the matching relation are designed to correspond to the definition of subprotocol. Depending on the choice of subprotocol relation, we have:

$S <\# T$ if $S <: T\text{-}Protocol[S]$ (F-bounded interpretation)

or

$S <\# T$ if $S\text{-}Protocol \prec: T\text{-}Protocol$ (higher-order interpretation)

- With either definition we have *MinMax* <# *Max*.

- Matching does not enjoy a subsumption property (that is, $S <\# T$ and $s : S$ do not imply that $s : T$); however, matching is useful for parameterizing over all the types that match a given one:

ObjectOperator $P_3[X <\# Max]$ **is** … **end**;

- The instantiation $P_3[MinMax]$ is legal.
- In summary, even in the presence of contravariant occurrences of **Self**, and in absence of subtyping, there can be inheritance of binary methods like *max*. Unfortunately, subsumption is lost in this context, and quantification over subtypes is no longer very useful. These disadvantages are partially compensated by the existence of a subprotocol relation, and by the ability to parameterize with respect to this relation.

Type Information, Lost and Found

Although subsumption has no run-time effect, it has the effect of reducing static knowledge of the true type of an object.

- Imagine a root class with no attributes, such that all classes are subclasses of the root class.

  Any object can be viewed, by subsumption, as a member of the root class. Then it is a useless object with no attributes.

- When an object is subsumed from *InstanceTypeOf*(*reCell*) to *InstanceTypeOf*(*cell*), we lose direct access to its *backup* field.

  However, the field *backup* is still used, through self, by the body of the overriding method *set*.

- So, attributes forgotten by subsumption can still be used thanks to dynamic dispatch.

In purist object-oriented methodology, dynamic dispatch is the only mechanism for accessing attributes forgotten by subsumption.

- This position is often taken on abstraction grounds: no knowledge should be obtainable about objects except through their methods.
- In the purist approach, subsumption provides a simple and effective mechanism for hiding private attributes.

  When we create a *reCell* and give it to a client as a *cell*, we can be sure that the client cannot directly affect the *backup* field.

Typecase

Most languages, however, provide some way of inspecting the type of an object and thus of regaining access to its forgotten attributes.

- A procedure with parameter x of type *InstanceTypeOf*(*cell*) could contain the following code.

```
typecase x
    when rc: InstanceTypeOf(reCell) do … rc.restore() … ;
    when c: InstanceTypeOf(cell) do … c.set(3) … ;
end;
```

- The **typecase** statement binds x to c or to rc depending on the true (run-time) type of x.
- Previously inaccessible attributes can now be used in the rc branch.

The **typecase** mechanism is useful, but it is considered impure for several methodological reasons (and also for theoretical ones).

- It violates the object abstraction, revealing information that may be regarded as private.
- It renders programs more fragile by introducing a form of dynamic failure when none of the branches apply.
- It makes code less extensible: when adding a subclass one may have to revisit and extend the **typecase** statements in existing code.
  - This is a bad property, in particular because the source code of commercial libraries may not be available.
  - In the purist framework, the addition of a new subclass does not require recoding of existing classes.

Although **typecase** may be ultimately an unavoidable feature, its methodological drawbacks require that it be used prudently.

The desire to reduce the uses of **typecase** has shaped much of the type structure of object-oriented languages.

- In particular, **typecase** on self is necessary for emulating objects in conventional languages by records of procedures.

  In contrast, the standard typing of methods in object-oriented languages avoids this need for **typecase**.

- More sophisticated typings of methods are aimed at avoiding **typecase** also on method results and on method arguments.

O-O SUMMARY

- Class-based: various implementation techniques based on embedding and/or delegation. Self is the receiver.
- Object-based: various language mechanisms based on embedding and/or delegation. Self is the receiver.
- Object-based can emulate class-based. (By traits, or by otherwise reproducing the implementations techniques of class-based languages.)

One Step Further

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

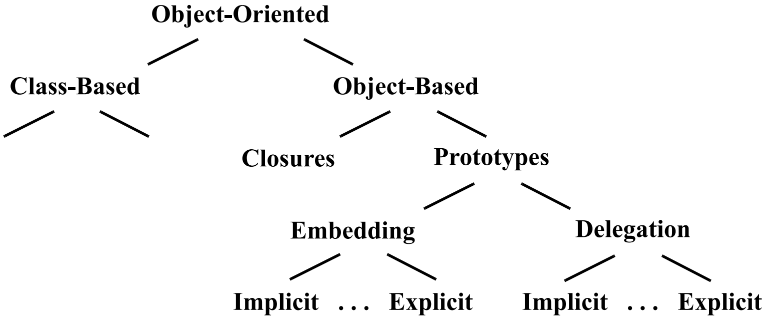
Our Approach to Modeling

- 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.)

Foundations

- Objects can emulate classes (by traits) and procedures (by "stack frame objects").
- *Everything* can indeed be an object.

A Taxonomy


Object-Oriented
Class-Based
Object-Based
Closures
Prototypes
Embedding
Delegation
Implicit . . . Explicit
Implicit . . . Explicit

TYPE SYSTEMS

(transparencies by Martín Abadi,
largely based on the paper
"Type Systems" by Luca Cardelli)

Types

A program variable can assume a range of values during the execution of a program.

An upper bound of such a range is called a **type** of the variable.

- ~ For example, a variable x of type *Boolean* is supposed to assume only boolean values during every run of a program.
- ~ If x has type *Boolean*, then the boolean expression $not(x)$ has a sensible meaning in every run of the program.

Typed and Untyped Languages

Languages that do not restrict the range of variables are called **untyped languages**.

- Operations may be applied to inappropriate arguments: the result may be a fixed value, a fault, an exception, or an unspecified effect.
- The pure λ-calculus is an extreme case of an untyped language where no fault ever occurs.

Languages where variables can be given (nontrivial) types are called **typed languages**.

- A type system is that component of a typed language that keeps track of the types of variables and other program expressions.
- A language is typed by virtue of the existence of a type system for it, whether or not types actually appear in the syntax of programs.
- Typed languages are **explicitly typed** if types are part of the syntax, and **implicitly typed** otherwise.

Properties of Type Systems

Types have pragmatic characteristics that distinguish them from other kinds of program annotations.

- They are more precise than comments.
- They are more easily mechanizable than formal specifications.

Some expected properties of type systems are:

- Types should be checkable, algorithmically.
- Type rules should be transparent: it should be possible to predict whether a program will typecheck, or to see why it does not.

Type Soundness

One important purpose of a type system is to prevent the occurrence of execution errors during the running of a program.

When this property holds for all of the program runs that can be expressed within a language, the language is **type sound**.

- A fair amount of careful analysis is required to avoid false claims of type soundness.
- Even informal knowledge of the principles of type systems helps.
- A formal presentation of a type system permits a formal proof, and also provides an independent specification for a typechecker.

Caveats

- These categories are somewhat simplistic: being typed, or being explicitly typed, can be seen as a matter of degree.
- We will ignore some kinds of type information, for example:
  - untraced and traced (used for garbage collection),
  - static and dynamic (used in partial evaluation),
  - unclassified, secret, and top secret (used for confidentiality),
  - untrusted and trusted (used for integrity),
  - ....
- Even the notion of execution error is difficult to make precise in a simple, general manner.

Advantages of Typed Languages

The use of types in programming has several practical benefits:

- *Economy of execution*
  - In the earliest high-level languages (e.g., FORTRAN), type information was introduced for generating reasonable code for numeric computations.
  - In ML, accurate type information eliminates the need for *nil*-checking on pointer dereferencing.

  Accurate type information at compile time leads to the application of the appropriate operations at run time without expensive tests.
- *Economy of small-scale development*
  - When a type system is well designed, typechecking can capture a large fraction of routine programming errors.
  - The errors that occur are easier to debug, because large classes of other errors have been ruled out.
  - Experienced programmers can adopt a style that causes some logical errors to be detected by a typechecker.

- *Economy of maintenance*
  - Code written in untyped languages can be maintained only with great difficulty.
  - Even weakly checked unsafe languages are superior to safe but untyped languages.
- *Economy of large-scale development*
  - Teams of programmers can negotiate interfaces, then proceed separately to implement the corresponding pieces of code.
  - Dependencies between pieces of code are minimized, and code can be locally rearranged without fear of global effects.

These benefits can be achieved with informal specifications for interfaces, but typechecking helps.

Execution Errors in More Detail

There are two kinds of execution errors:

- **trapped errors** cause the computation to stop immediately, e.g.,
  - ~ division by zero,
  - ~ accessing an illegal address,
- **untrapped errors** may go unnoticed (for a while), e.g.,
  - ~ accessing data past the end of an array,
  - ~ jumping to an address outside the instruction stream.

A program fragment is **safe** if it does not cause untrapped errors.

Languages where all program fragments are safe are **safe languages**.

Safety

	Typed	Untyped
Safe	ML	LISP (classic)
Unsafe	C	Assembler

Good Behavior

For any given language, we may designate a subset of the possible execution errors as **forbidden errors**.

The forbidden errors should include all of the untrapped errors, plus a subset of the trapped errors.

A program fragment that does not cause forbidden errors has **good behavior** (or is well behaved).

A language where all of the (legal) programs have good behavior is called **strongly checked**.

- No untrapped errors occur.
- None of the forbidden trapped errors occur.
- Other trapped errors may occur; it is the programmer's responsibility to avoid them.

Checking Good Behavior

Untyped languages may enforce good behavior by run time checks.

Typed languages (like ML and Pascal) can enforce good behavior by performing static checks to prevent some programs from running.

- These languages are **statically checked**.
- The checking process is **typechecking**.
- The algorithm that performs this check is the typechecker.
- A program that passes the typechecker is said to be **well typed**.

Typed languages may also perform some dynamic checks.

Some languages take advantage of their static type structures to perform dynamic type tests (cf. Java's InstanceOf).

In reality, certain statically checked languages do not ensure safety.

These languages are **weakly checked** (or **weakly typed**): some unsafe operations are detected statically and some are not detected.

- Pascal is unsafe only when untagged variant types and function parameters are used.
- C has many unsafe and widely used features, such as pointer arithmetic and casting.
- Modula-3 supports unsafe features, but only in modules that are explicitly marked as unsafe.

Type Equivalence

Most type systems include a relation of type equivalence.

Are X and Y equivalent?

$$\mathit{type}\ X = \mathit{Real}$$
$$\mathit{type}\ Y = \mathit{Real}$$

- When they fail to match by virtue of being distinct type names, we have *by-name equivalence*.
- When they match by virtue of being associated with similar types, we have *structural equivalence*.

- Most compilers use a combination of by-name and structural equivalence (sometimes without a satisfactory specification).
- Structural equivalence has several advantages:
  - It can be defined easily, without strange special cases.
  - It easily allows “anonymous” types.
  - It works well with data sent over a network, with persistent data.
  - It works well with program sources developed in pieces.
- Pure structural equivalence can be limited through branded types.

  type X = Real brand Temperature
  type Y = Real brand Speed

When Types Do Not Match: Coercions?

Many languages do not give up when a type mismatch occurs.

- Instead, they apply a coercion in the offending program.
- Sometimes the coercion happens at run time, with significant cost.

Languages vary in their use of coercions.

- For languages with lots of basic types (such as COBOL) frequent coercions are a necessity.
- Many languages allow coercions at least for numeric types.

Silent coercions have advantages and disadvantages:

- They free the programmer from tedious conversions.
- Typechecking becomes harder to predict, and can turn simple typos into serious mistakes.
- If a coercion does any allocation, then data structures may not be shared as intended.

The Language of Type Systems

A type system specifies the type rules of a programming language independently of particular typechecking algorithms.

This is analogous to describing a syntax by a formal grammar, independently of particular parsing algorithms (and as important!).

Type systems are formulated in terms of assertions called *judgments*.

A typical judgment has the form:

$$\Gamma \vdash \Im$$

Here Γ is a *static typing environment*; for example, an ordered list of distinct variables and their types, of the form $\emptyset, x_1{:}A_1, ..., x_n{:}A_n$.

The form of the *assertion* $\Im$ varies from judgment to judgment, but all the free variables of $\Im$ must be declared in Γ.

The Typing Judgment

The *typing judgment*, which asserts that a term M has a type A with respect to a static typing environment for the free variables of M.

It has the form:

$\Gamma \vdash M : A$ (M has type A in Γ)

Examples:

$\phi \vdash \mathit{true} : \mathit{Bool}$

$\phi, x{:}\mathit{Nat} \vdash x{+}1 : \mathit{Nat}$

Type Rules

Type rules are rules for deriving judgments.

A typical type rule has the form:

(Rule name) (Annotations)

$$\frac{\Gamma_1 \vdash \mathfrak{I}_1 \quad \ldots \quad \Gamma_n \vdash \mathfrak{I}_n \quad \text{(Annotations)}}{\Gamma \vdash \mathfrak{I}}$$

Examples:

(Val n) ($n = 0, 1, \ldots$)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : Nat}$$

(Val +)

$$\frac{\Gamma \vdash M : Nat \qquad \Gamma \vdash N : Nat}{\Gamma \vdash M{+}N : Nat}$$

(Env $\emptyset$)

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

Type Derivations

A *derivation* in a given type system is a tree of judgments

- with leaves at the top and a root at the bottom,
- where each judgment is obtained from the ones immediately above it by some rule of the system.

A valid judgment is one that can be obtained as the root of a derivation in a given type system.

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash \diamond}\ \text{by (Env }\varnothing)}{\varnothing \vdash 1 : \mathit{Nat}}\ \text{by (Val } n) \qquad \dfrac{\dfrac{}{\varnothing \vdash \diamond}\ \text{by (Env }\varnothing)}{\varnothing \vdash 2 : \mathit{Nat}}\ \text{by (Val } n)}{\varnothing \vdash 1+2 : \mathit{Nat}}\ \text{by (Val +)}$$

Well Typing and Type Soundness

In a given type system, a term M is *well typed* for an environment Γ, if there is a type A such that $\Gamma \vdash M : A$ is a valid judgment.

The discovery of a derivation (and hence of a type) for a term is called the *type inference problem*. This problem can be very hard.

We can check the internal consistency of a type system by proving a *type soundness theorem.*

- For denotational semantics, we expect that if $\emptyset \vdash M : A$ is valid, then $[\![M]\!] \in [\![A]\!]$ holds.
- For operational semantics, we expect that if $\emptyset \vdash M : A$ and M reduces to M' then $\emptyset \vdash M' : A$.

This theorem shows that well typing corresponds to a semantic notion of good behavior.

First-Order Type Systems

In this context, first-order means lacking type parameterization and type abstraction (like Pascal, unlike ML).

Languages with higher-order functions can be first-order.

The type systems of most common languages are first-order.

A common mathematical example of a first-order language is the first-order typed λ-calculus, called system F_1.

The main changes from the untyped λ-calculus are:

- the addition of type annotations for bound variables (as in $\lambda x{:}A.x$),
- the addition of basic types (such as *Bool* and *Nat*),
- the addition of types for functions, of the form $A \rightarrow B$,
- the requirement that programs typecheck.

Syntax of F_1

A,B ::=		types	
K	$K \in Basic$		basic types
$A \rightarrow B$			function types
M,N ::=		terms	
x			variable
$\lambda x{:}A.M$			function abstraction
$M\ N$			function application

Judgments for F_1

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash M : A$	M is a well-formed term of type A in Γ

Rules for $\mathbf{F_1}$

(Env $\emptyset$)

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

(Env x)

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

(Type Const)

$$\frac{\Gamma \vdash \diamond \qquad K \in Basic}{\Gamma \vdash K}$$

(Type Arrow)

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \rightarrow B}$$

(Val x)

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

(Val Fun)

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \rightarrow B}$$

(Val Appl)

$$\frac{\Gamma \vdash M : A \rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

A derivation in F_1

$$
\begin{array}{ccccccc}
\cline{1-1}\cline{3-3}\cline{5-5}\cline{7-7}
\emptyset \vdash \diamond & \text{by (Env } \emptyset) & \emptyset \vdash \diamond & \text{by (Env } \emptyset) & \emptyset \vdash \diamond & \text{by (Env } \emptyset) & \emptyset \vdash \diamond \\
\cline{1-1}\cline{3-3}\cline{5-5}\cline{7-7}
\emptyset \vdash K & \text{by (Type Const)} & \emptyset \vdash K & \text{by (Type Const)} & \emptyset \vdash K & \text{by (Type Const)} & \emptyset \vdash K \\
\cline{1-3}\cline{5-7}
\multicolumn{3}{c}{\emptyset \vdash K \to K} & \text{by (Type Arrow)} & \multicolumn{3}{c}{\emptyset \vdash K \to K} \\
\cline{1-3}\cline{5-7}
\multicolumn{3}{c}{\emptyset, y{:}K \to K \vdash \diamond} & \text{by (Env } x) & \multicolumn{3}{c}{\emptyset, y{:}K \to K \vdash \diamond} \\
\cline{1-3}\cline{5-7}
\multicolumn{3}{c}{\emptyset, y{:}K \to K \vdash K} & \text{by (Type Const)} & \multicolumn{3}{c}{\emptyset, y{:}K \to K \vdash K} \\
\cline{1-3}\cline{5-7}
\multicolumn{3}{c}{\emptyset, y{:}K \to K, z{:}K \vdash \diamond} & \text{by (Env } x) & \multicolumn{3}{c}{\emptyset, y{:}K \to K, z{:}K \vdash \diamond} \\
\cline{1-3}\cline{5-7}
\multicolumn{3}{c}{\emptyset, y{:}K \to K, z{:}K \vdash y : K \to K} & \text{by (Val } x) & \multicolumn{3}{c}{\emptyset, y{:}K \to K, z{:}K \vdash z : K} \\
\hline
\multicolumn{7}{c}{\emptyset, y{:}K \to K, z{:}K \vdash y(z) : K} \\
\hline
\multicolumn{7}{c}{\emptyset, y{:}K \to K \vdash \lambda z{:}K.y(z) : K \to K}
\end{array}
$$

where the last two steps are by (Val Appl) and (Val Fun).

Study of a First-Order Type System

- F_1 allows some programming with higher-order functions.

  For example, the Church numerals:

$$\lambda x{:}K{\to}K.\ \lambda y{:}K.\ y$$

$$\lambda x{:}K{\to}K.\ \lambda y{:}K.\ x(y)$$

$$\lambda x{:}K{\to}K.\ \lambda y{:}K.\ x(x(y))$$

$$\lambda x{:}K{\to}K.\ \lambda y{:}K.\ x(x(x(y)))$$

$$\ldots$$

  all typecheck with type $(K{\to}K){\to}(K{\to}K)$

- Some untyped terms cannot be annotated so that they typecheck in F_1:

$$\lambda x{:}?.\ x(x)$$

- We can prove type soundness theorems for F_1.

  In particular:

$$\text{If } \varnothing \vdash M : A \text{ and } M \to^{l} N \text{ then } \varnothing \vdash N : A.$$

  ~ Here $\to^{l}$ is the trivial extension of the call-by-name operational semantics of the untyped λ-calculus to F_1.

  ~ This is called a *subject reduction theorem*.

  ~ This theorem implies, for example, that if $\varnothing \vdash M : A$ then M does not evaluate to $\lambda x{:}K.\, x(...)$, where a non-function is being applied.

  ~ For richer type systems, subject reduction theorems can be hard.

Basic Types: Unit

We add a set of rules for each of several new type constructions, following a fairly regular pattern.

We begin with some basic data types:

- the type *Unit*, whose only value is the constant *unit*,
- the type *Bool*, whose values are *true* and *false*,
- the type *Nat*, whose values are the natural numbers.

Unit Type

(Type Unit)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Unit}}$$

(Val Unit)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{unit} : \mathit{Unit}}$$

The *Unit* type is often used as a filler for uninteresting arguments and results. (It corresponds to *Void* or *Null* in some languages.)

Basic Types: Booleans

Bool Type

(Type Bool)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Bool}}$$

(Val True)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{true} : \mathit{Bool}}$$

(Val False)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{false} : \mathit{Bool}}$$

(Val Cond)
$$\frac{\Gamma \vdash M : \mathit{Bool} \qquad \Gamma \vdash N_1 : A \qquad \Gamma \vdash N_2 : A}{\Gamma \vdash (\mathit{if}_A\ M\ \mathit{then}\ N_1\ \mathit{else}\ N_2) : A}$$

if_A gives a hint to the typechecker that the result type should be A, and that types inferred for N_1 and N_2 should be compared with A.

It is normally the task of a typechecker to synthesize A and similar type information.

Basic Types: Natural Numbers

Nat Type

(Type Nat)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Nat}$$

(Val Zero)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash 0 : Nat}$$

(Val Succ)
$$\frac{\Gamma \vdash M : Nat}{\Gamma \vdash succ\ M : Nat}$$

(Val Pred)
$$\frac{\Gamma \vdash M : Nat}{\Gamma \vdash pred\ M : Nat}$$

(Val IsZero)
$$\frac{\Gamma \vdash M : Nat}{\Gamma \vdash isZero\ M : Bool}$$

Structured Types: Products

A product type $A_1 \times A_2$ is the type of pairs of values with first component of type A_1 and second component of type A_2.

These components can be extracted with the projections *first* and *second*, respectively.

Product Types

(Type Product)

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}$$

(Val Pair)

$$\frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2}$$

(Val First)

$$\frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \mathit{first}\ M : A_1}$$

(Val Second)

$$\frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \mathit{second}\ M : A_2}$$

Instead of the projections, we can use a *with* statement.

- The *with* statement decomposes a pair M and binds its components to two separate variables x_1 and x_2 in the scope N.
- The *with* notation is related to pattern matching in ML, and also to Pascal's *with* statement.

Product Types (Cont.)

(Val With)

$$\frac{\Gamma \vdash M : A_1{\times}A_2 \qquad \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash N : B}{\Gamma \vdash (\mathit{with}\ (x_1{:}A_1, x_2{:}A_2) := M\ \mathit{do}\ N) : B}$$

Structured Types: Unions

An element of a union type A_1+A_2 is an element of A_1 tagged with a *left* token (created by *inLeft*), or an element of A_2 tagged with a *right* token (created by *inRight*).

The tags can be tested by *isLeft* and *isRight*, and the values extracted with *asLeft* and *asRight*.

Union Types

(Type Union)
$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1+A_2}$$

(Val inLeft)
$$\frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash \mathit{inLeft}_{A_2}\, M_1 : A_1+A_2}$$

(Val inRight)
$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \mathit{inRight}_{A_1}\, M_2 : A_1+A_2}$$

(Val isLeft)
$$\frac{\Gamma \vdash M : A_1+A_2}{\Gamma \vdash \mathit{isLeft}\ M : \mathit{Bool}}$$

(Val isRight)
$$\frac{\Gamma \vdash M : A_1+A_2}{\Gamma \vdash \mathit{isRight}\ M : \mathit{Bool}}$$

(Val asLeft)
$$\frac{\Gamma \vdash M : A_1+A_2}{\Gamma \vdash \mathit{asLeft}\ M : A_1}$$

(Val asRight)
$$\frac{\Gamma \vdash M : A_1+A_2}{\Gamma \vdash \mathit{asRight}\ M : A_2}$$

The use of *asLeft* (and *asRight*) can give rise to errors:

- Any result of *asLeft* must have type A_1.
- When *asLeft* is mistakenly applied to a right-tagged value, a trapped error or exception is produced.

The *case* construct can replace *isLeft*, *isRight*, *asLeft*, *asRight*, and the related trapped errors.

It also eliminates any dependence on the *Bool* type.

Union Types (Cont.)

(Val Case)

$$\frac{\Gamma \vdash M : A_1{+}A_2 \qquad \Gamma, x_1{:}A_1 \vdash N_1 : B \qquad \Gamma, x_2{:}A_2 \vdash N_2 : B}{\Gamma \vdash (case_B\ M\ of\ x_1{:}A_1\ then\ N_1 \mid x_2{:}A_2\ then\ N_2) : B}$$

The *case* construct executes one of two branches depending on the tag of M, with the untagged contents of M bound to x_1 or x_2 in the scope of N_1 or N_2, respectively.

Structured Types: Records

A record type is a named collection of types, with a value-level operation for extracting components by name.

We ignore the order of the record components (and identify expressions that differ only in this order).

Record Types

(Type Record) (l_i distinct)

$$\frac{\Gamma \vdash A_1 \quad \ldots \quad \Gamma \vdash A_n}{\Gamma \vdash \mathit{Record}(l_1{:}A_1, \ldots, l_n{:}A_n)}$$

(Val Record) (l_i distinct)

$$\frac{\Gamma \vdash M_1 : A_1 \quad \ldots \quad \Gamma \vdash M_n : A_n}{\Gamma \vdash \mathit{record}(l_1{=}M_1, \ldots, l_n{=}M_n) : \mathit{Record}(l_1{:}A_1, \ldots, l_n{:}A_n)}$$

(Val Record Select)

$$\frac{\Gamma \vdash M : \mathit{Record}(l_1{:}A_1, \ldots, l_n{:}A_n) \qquad j \in 1..n}{\Gamma \vdash M.l_j : A_j}$$

(Val Record With)

$$\frac{\Gamma \vdash M : \mathit{Record}(l_1{:}A_1, \ldots, l_n{:}A_n) \qquad \Gamma, x_1{:}A_1, \ldots, x_n{:}A_n \vdash N : B}{\Gamma \vdash (\mathit{with}\ (l_1{=}x_1{:}A_1, \ldots, l_n{=}x_n{:}A_n) := M\ \mathit{do}\ N) : B}$$

Product types $A_1 \times A_2$ can be defined as $\mathit{Record}(\mathit{first}{:}A_1, \mathit{second}{:}A_2)$.

Structured Types: Variants

Variant types

(Type Variant) (l_i distinct)

$$\frac{\Gamma \vdash A_1 \quad ... \quad \Gamma \vdash A_n}{\Gamma \vdash \mathit{Variant}(l_1{:}A_1, ..., l_n{:}A_n)}$$

(Val Variant) (l_i distinct)

$$\frac{\Gamma \vdash A_1 \quad ... \quad \Gamma \vdash A_n \qquad \Gamma \vdash M_j : A_j \qquad j \in 1..n}{\Gamma \vdash \mathit{variant}_{(l_1:A_1,\, ...,\, l_n:A_n)}(l_j{=}M_j) : \mathit{Variant}(l_1{:}A_1, ..., l_n{:}A_n)}$$

(Val Variant Is)

$$\frac{\Gamma \vdash M : \mathit{Variant}(l_1{:}A_1, ..., l_n{:}A_n) \qquad j \in 1..n}{\Gamma \vdash M \ \mathit{is}\ l_j : \mathit{Bool}}$$

(Val Variant As)

$$\frac{\Gamma \vdash M : \mathit{Variant}(l_1{:}A_1, ..., l_n{:}A_n) \qquad j \in 1..n}{\Gamma \vdash M \ \mathit{as}\ l_j : A_j}$$

(Val Variant Case)

$$\frac{\Gamma \vdash M : \mathit{Variant}(l_1{:}A_1, ..., l_n{:}A_n) \qquad \Gamma, x_1{:}A_1 \vdash N_1 : B \quad ... \quad \Gamma, x_n{:}A_n \vdash N_n : B}{\Gamma \vdash (\mathit{case}_B\ M\ \mathit{of}\ l_1{=}x_1{:}A_1\ \mathit{then}\ N_1 \mid ... \mid l_n{=}x_n{:}A_n\ \mathit{then}\ N_n) : B}$$

Enumeration Types

Enumeration types, such as $\{red, green, blue\}$, can be defined as $Variant(red{:}Unit, green{:}Unit, blue{:}Unit)$.

Other First-Order Types

See L. Cardelli's paper for a discussion of reference types and arrays.

Recursive Types

Instead of declaring recursive types, as in:

$$\textit{type } X \quad = \quad \textit{Unit} + (\textit{Nat} \times X)$$

we use recursive types of the form $\mu X.A$, like:

$$\mu X.(\textit{Unit} + (\textit{Nat} \times X))$$

Here X is a type variable. Intuitively, $\mu X.A$ is the solution to recursive to the equation X=A where X may occur in A. So for example:

$$\mu X.(\textit{Unit} + (\textit{Nat} \times X)) \quad = \quad \textit{Unit} + (\textit{Nat} \times \mu X.(\textit{Unit} + (\textit{Nat} \times X)))$$

This notation postpones the need for type declarations.

Anyway, with structural equivalence, we would want to say that the type X declared by X=A "is" $\mu X.A$.

For writing rules for recursive types, we enrich environments with type variables.

Recursive Types

(Env X)
$$\frac{\Gamma \vdash \diamond \qquad X \notin dom(\Gamma)}{\Gamma, X \vdash \diamond}$$

(Type Rec)
$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \mu X.A}$$

(Val Fold)
$$\frac{\Gamma \vdash M : A[\mu X.A/X]}{\Gamma \vdash \mathit{fold}_{\mu X.A}\, M : \mu X.A}$$

(Val Unfold)
$$\frac{\Gamma \vdash M : \mu X.A}{\Gamma \vdash \mathit{unfold}_{\mu X.A}\, M : A[\mu X.A/X]}$$

The operations *unfold* and *fold* are explicit coercions that map between a recursive type $\mu X.A$ and its unfolding $A[\mu X.A/X]$.

These coercions do not have any run time effect. They are usually omitted from the syntax of programming languages.

List Types

$$List_A \triangleq \mu X.Unit+(A\times X)$$

$$nil_A : List_A \triangleq fold(inLeft\ unit)$$

$$cons_A : A\to List_A\to List_A \triangleq \lambda hd{:}A.\ \lambda tl{:}List_A.\ fold(inRight\ \langle hd,tl\rangle)$$

$$listCase_{A,B} : List_A\to B\to(A\times List_A\to B)\to B \triangleq$$
$$\quad \lambda l{:}List_A.\ \lambda n{:}B.\ \lambda c{:}A\times List_A\to B.$$
$$\qquad case\ (unfold\ l)\ of\ unit{:}Unit\ then\ n \mid p{:}A\times List_A\ then\ c\ p$$

Value-Level Recursion

Value-level recursion can be typechecked using recursive types.

Encoding of Divergence and Recursion via Recursive Types

$$\bot_A : A \quad \triangleq \quad (\lambda x{:}B.\ (\mathit{unfold}_B\ x)\ x)\ (\mathit{fold}_B\ (\lambda x{:}B.\ (\mathit{unfold}_B\ x)\ x))$$

$$\mathbf{Y}_A : (A \to A) \to A \quad \triangleq$$
$$\lambda f{:}A \to A.\ (\lambda x{:}B.\ f\ ((\mathit{unfold}_B\ x)\ x))\ (\mathit{fold}_B\ (\lambda x{:}B.\ f\ ((\mathit{unfold}_B\ x)\ x)))$$

where $B \equiv \mu X.\ X \to A$, for an arbitrary A

Untyped Programming via Recursive Types

Encoding the Untyped λ-calculus via Recursive Types

$V \triangleq \mu X.X \to X$ the type of untyped λ-terms

$\langle\!\langle x \rangle\!\rangle \triangleq x$ translation $\langle\!\langle - \rangle\!\rangle$ from untyped λ-terms to V elements

$\langle\!\langle \lambda x.M \rangle\!\rangle \triangleq \mathit{fold}_V\, (\lambda x{:}V.\langle\!\langle M \rangle\!\rangle)$

$\langle\!\langle M\, N \rangle\!\rangle \triangleq (\mathit{unfold}_V\, \langle\!\langle M \rangle\!\rangle)\, \langle\!\langle N \rangle\!\rangle$

A Type System for an Imperative Language

Syntax of the imperative language

A ::=	types
$Bool$	boolean type
Nat	natural numbers type
$Proc$	procedure type
D ::=	declarations
proc $I = C$	procedure declaration
var $I : A = E$	variable declaration
C ::=	commands
$I := E$	assignment
$C_1; C_2$	sequential composition
begin D in C end	block
call I	procedure call
while E do C end	while loop

$E ::=$	expressions
I	identifier
N	numeral
$E_1 + E_2$	sum of two numbers
E_1 not= E_2	inequality of two numbers

Judgments for the imperative language

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash C$	C is a well-formed command in Γ
$\Gamma \vdash E : A$	E is a well-formed expression of type A in Γ
$\Gamma \vdash D \therefore S$	D is a well-formed declaration of signature S in Γ

Type rules for the imperative language

(Env $\emptyset$)

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

(Env I)

$$\frac{\Gamma \vdash A \qquad I \notin dom(\Gamma)}{\Gamma, I{:}A \vdash \diamond}$$

(Type Bool)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Bool}$$

(Type Nat)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Nat}$$

(Type Proc)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Proc}$$

(Decl Proc)

$$\frac{\Gamma \vdash C}{\Gamma \vdash (\text{proc } I = C) \therefore (I : Proc)}$$

(Decl Var)

$$\frac{\Gamma \vdash E : A \qquad A \in \{Bool, Nat\}}{\Gamma \vdash (\text{var } I : A = E) \therefore (I : A)}$$

(Comm Assign)

$$\frac{\Gamma \vdash I : A \qquad \Gamma \vdash E : A}{\Gamma \vdash I := E}$$

(Comm Sequence)

$$\frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1 \,;\, C_2}$$

(Comm Block)

$$\frac{\Gamma \vdash D \therefore (I : A) \qquad \Gamma, I{:}A \vdash C}{\Gamma \vdash \text{begin } D \text{ in } C \text{ end}}$$

(Comm Call)

$$\frac{\Gamma \vdash I : \mathit{Proc}}{\Gamma \vdash \text{call } I}$$

(Comm While)

$$\frac{\Gamma \vdash E : \mathit{Bool} \qquad \Gamma \vdash C}{\Gamma \vdash \text{while } E \text{ do } C \text{ end}}$$

(Expr Identifier)

$$\frac{\Gamma_1, I{:}A, \Gamma_2 \vdash \diamond}{\Gamma_1, I{:}A, \Gamma_2 \vdash I : A}$$

(Expr Numeral)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash N : \mathit{Nat}}$$

(Expr Plus)

$$\frac{\Gamma \vdash E_1 : \mathit{Nat} \qquad \Gamma \vdash E_2 : \mathit{Nat}}{\Gamma \vdash E_1 + E_2 : \mathit{Nat}}$$

(Expr NotEq)

$$\frac{\Gamma \vdash E_1 : \mathit{Nat} \qquad \Gamma \vdash E_2 : \mathit{Nat}}{\Gamma \vdash E_1 \text{ not= } E_2 : \mathit{Bool}}$$

Type Inference

Type inference is the problem of finding a type A for a term M in a given typing environment Γ, so that $\Gamma \vdash M : A$, if any such type exists.

- In systems with abundant type annotations, the type inference problem amounts to little more than checking the annotations.
- The problem is not always trivial but, as in the case of F_1, simple typechecking algorithms may exist.

A harder problem, called *type reconstruction*, consists in starting with an untyped program M, and finding an environment Γ, a type-annotated version M' of M, and a type A such that $\Gamma \vdash M' : A$.

A Type Inference Algorithm for $\mathbf{F_1}$

$Type(\Gamma, M)$ takes an environment Γ and a term M and produces the unique type of M, if M has a type.

The instruction *fail* causes a global failure of the algorithm.

We assume that the initial environment parameter Γ is well formed.

Type inference algorithm for $\mathbf{F_1}$

$Type(\Gamma, x) \quad \triangleq$

$\quad$ *if* $x{:}A \in \Gamma$ *for some* A *then* A *else fail*

$Type(\Gamma, \lambda x{:}A.M) \quad \triangleq$

$\quad$ *if* $x \notin dom(\Gamma)$ *then* $A \rightarrow Type((\Gamma, x{:}A), M)$ *else restart after renaming*

$Type(\Gamma, M\, N) \quad \triangleq$

$\quad$ *if* $Type(\Gamma, M) \equiv Type(\Gamma, N) \rightarrow B$ *for some* B *then* B *else fail*

For example:

$$
\begin{aligned}
& \mathit{Type}((\emptyset, y{:}K{\to}K), \lambda z{:}K.y(z)) \\
& = K{\to}\mathit{Type}((\emptyset, y{:}K{\to}K, z{:}K), y(z)) \\
& = K{\to}(\mathit{if}\ \mathit{Type}((\emptyset, y{:}K{\to}K, z{:}K), y) \equiv \\
& \qquad\qquad \mathit{Type}((\emptyset, y{:}K{\to}K, z{:}K), z){\to}B\ \mathit{for\ some}\ B \\
& \qquad\qquad \mathit{then}\ B\ \mathit{else\ fail}) \\
& = K{\to}(\mathit{if}\ K{\to}K \equiv K{\to}B\ \mathit{for\ some}\ B\ \mathit{then}\ B\ \mathit{else\ fail}) \\
& = K{\to}K
\end{aligned}
$$

The algorithm for type inference in F_1 is fundamental.

- It can be extended in straightforward ways to all of the first-order type structures studied earlier.
- It is the basis of the typechecking algorithms of Pascal and similar procedural languages.
- It can be extended (though not always easily) to handle subtyping and higher-order constructs.

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

Higher-order object calculi

Calculus:	**Ob**	$\mathbf{Ob}_{\mu}$	$\mathbf{Ob}_{<:}$	$\mathbf{Ob}_{<:\mu}$	ς**Ob**	**S**	$\mathbf{S}_{\forall}$	*nn*	$\boldsymbol{Ob}_{\omega<:\mu}$
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,
- (some) axiomatic semantics (due to M. Abadi and R. Leino),
- type systems,
- type inference algorithms (due to J. Palsberg, F. Henglein),
- 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.

Just Objects, No Classes

- Language analysis:

  Class-based languages → Object-based languages → Object calculi

- Language synthesis:

  Object calculi → Object-based languages → Class-based languages

Embedding and Delegation

- 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.)

AN UNTYPED OBJECT CALCULUS

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:

$$
\begin{aligned}
o \;\triangleq\;& \\
& [l = \varsigma(x)\, [\,], \\
& \;\; m = \varsigma(x)\, x.l]
\end{aligned}
$$

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

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

$$
\begin{aligned}
\mathit{cell} \;\triangleq\;& \\
& [\mathit{contents} = \varsigma(x)\, 0, \\
& \;\; \mathit{set} = \varsigma(x)\, \lambda(n)\, x.\mathit{contents} \Leftarrow \varsigma(y)\, n]
\end{aligned}
$$

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

An Untyped Object Calculus: Reduction

- The notation $b \rightarrow c$ 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 $o \equiv [l_i = \varsigma(x_i) b_i \; ^{i \in 1..n}]$ $\qquad$ (l_i distinct)

$$
\begin{array}{lcll}
o.l_j & \rightarrow & b_j\{\!\{x_j \leftarrow o\}\!\} & (j \in 1..n) \\
o.l_j \Leftarrow \varsigma(y)b & \rightarrow & [l_j = \varsigma(y)b,\ l_i = \varsigma(x_i)b_i \; ^{i \in (1..n)-\{j\}}] & (j \in 1..n)
\end{array}
$$

In addition, if $a \rightarrow b$ then $C[a] \rightarrow C[b]$ 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	$o''.l \rightarrow (o''.l \Leftarrow \varsigma(x)x) \rightarrow o'$		

Static Scoping and Substitution, in Detail

Object scoping

$$
\begin{array}{lcl}
FV(\varsigma(y)b) & \triangleq & FV(b)-\{y\} \\
FV(x) & \triangleq & \{x\} \\
FV([l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]) & \triangleq & \cup^{i\in 1..n}\ FV(\varsigma(x_i)b_i) \\
FV(a.l) & \triangleq & FV(a) \\
FV(a.l{\Leftarrow}\varsigma(y)b) & \triangleq & FV(a) \cup FV(\varsigma(y)b)
\end{array}
$$

Object substitution

$$
\begin{array}{lcl}
(\varsigma(y)b)\{\!\{x{\leftarrow}c\}\!\} & \triangleq & \varsigma(y')(b\{\!\{y{\leftarrow}y'\}\!\}\{\!\{x{\leftarrow}c\}\!\}) \\
 & & \qquad \text{for} \quad y' \notin FV(\varsigma(y)b) \cup FV(c) \cup \{x\} \\
x\{\!\{x{\leftarrow}c\}\!\} & \triangleq & c \\
y\{\!\{x{\leftarrow}c\}\!\} & \triangleq & y \qquad\qquad \text{for } y \neq x \\
[l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]\{\!\{x{\leftarrow}c\}\!\} & \triangleq & [l_i{=}(\varsigma(x_i)b_i)\{\!\{x{\leftarrow}c\}\!\}\ ^{i\in 1..n}] \\
(a.l)\{\!\{x{\leftarrow}c\}\!\} & \triangleq & (a\{\!\{x{\leftarrow}c\}\!\}).l \\
(a.l{\Leftarrow}\varsigma(y)b)\{\!\{x{\leftarrow}c\}\!\} & \triangleq & (a\{\!\{x{\leftarrow}c\}\!\}).l{\Leftarrow}((\varsigma(y)b)\{\!\{x{\leftarrow}c\}\!\})
\end{array}
$$

Notation

- A *closed term* is a term without free variables.
- We write $b\{x\}$ to highlight that x may occur free in b.
- We write $b\{\!\{c\}\!\}$, instead of $b\{\!\{x \leftarrow c\}\!\}$, when $b\{x\}$ is present in the same context.
- We identify $\varsigma(x)b$ with $\varsigma(y)(b\{\!\{x \leftarrow y\}\!\})$, for all y not occurring free in b.

  (For example, we view $\varsigma(x)x$ and $\varsigma(y)y$ as the same method.)
- We identify any two objects that differ only in the order of their components.

  (For example, $[l_1 = \varsigma(x_1)b_1, l_2 = \varsigma(x_2)b_2]$ and $[l_2 = \varsigma(x_2)b_2, l_1 = \varsigma(x_1)b_1]$ are the same object for us.)

Expressiveness

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

$$[\ldots, l{=}b, \ldots] \;\triangleq\; [\ldots, l{=}\varsigma(y)b, \ldots] \qquad \text{for an unused } y$$

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

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,
  - ~ recursive definitions,
  - ~ 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 $\quad cell \triangleq$

$\qquad [contents = 0,$

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

Then $\quad cell.set(3)$

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

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

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

$\qquad\qquad .contents{:=}3$

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

and $\quad cell.set(3).contents$

$\qquad \rightarrow \quad ...$

$\qquad \rightarrow \quad 3$

A Cell with an Accessor

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

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

A Cell with Undo

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

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

Geometric Points

$$
\begin{aligned}
&\mathit{origin}_1 \triangleq \\
&\quad [x = 0, \\
&\quad\ \mathit{mv\_x} = \varsigma(s)\ \lambda(dx)\ s.x := s.x{+}dx] \\
&\mathit{origin}_2 \triangleq \\
&\quad [x = 0, y = 0, \\
&\quad\ \mathit{mv\_x} = \varsigma(s)\ \lambda(dx)\ s.x := s.x{+}dx, \\
&\quad\ \mathit{mv\_y} = \varsigma(s)\ \lambda(dy)\ s.y := s.y{+}dy]
\end{aligned}
$$

For example, we can define $\mathit{unit}_2 \triangleq \mathit{origin}_2.\mathit{mv\_x}(1).\mathit{mv\_y}(1)$, and then we can compute $\mathit{unit}_2.x = 1$.

Intuitively, all operations possible on origin_1 are possible on origin_2.

Hence we would like to obtain a type system where a point like origin_2 can be accepted in any context expecting a point like origin_1.

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:

$$
\begin{aligned}
& cond(true, false, true) \equiv ((true.then{:=}false).else{:=}true).if \\
& \quad \twoheadrightarrow ([if = \varsigma(x)\, x.then,\ then = false,\ else = \varsigma(x)\, x.else].else{:=}true).if \\
& \quad \twoheadrightarrow [if = \varsigma(x)\, x.then,\ then = false,\ else = true].if \\
& \quad \twoheadrightarrow [if = \varsigma(x)\, x.then,\ then = false,\ else = true].then \\
& \quad \twoheadrightarrow false
\end{aligned}
$$

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

- 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:

$$zero \qquad\qquad \equiv \quad [case = \lambda(z)\,\lambda(s)\,z,\;\; succ = \ldots\,]$$
$$one \quad \triangleq \quad zero.succ \quad \equiv \quad [case = \lambda(z)\,\lambda(s)\,s(zero),\;\; succ = \ldots\,]$$
$$pred \quad \triangleq \quad \lambda(n)\,n.case(zero)(\lambda(p)p)$$

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

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

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

The translation validates the β rule:

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

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

For example:

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

The translation has typed and imperative variants.

Functions as Objects, with Defaults

- $\lambda(x{=}c)b\{x\}$ is a function with a single parameter x with default c.
- $f(a)$ is a normal application of f to a, and $f()$ is an application of f to its default.

  For example, $(\lambda(x{=}c)x)() = c$ and $(\lambda(x{=}c)x)(a) = a$.

Translation of default parameters

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

$$\langle\!\langle b(a) \rangle\!\rangle \quad \triangleq \quad \langle\!\langle b \rangle\!\rangle \bullet \langle\!\langle a \rangle\!\rangle \qquad\qquad \text{where } p \bullet q \quad \triangleq \quad (p.arg{:=}q).val$$

$$\langle\!\langle b() \rangle\!\rangle \quad \triangleq \quad \langle\!\langle b \rangle\!\rangle.val$$

Recursion

The encoding of functions as objects yields fixpoint combinators for the object calculus.

However, more direct techniques for recursion exist.

In particular, we can define $\mu(x)b$ as follows:

$$\langle\!\langle \mu(x)b\{x\} \rangle\!\rangle \triangleq$$
$$[rec{=}\varsigma(x)\langle\!\langle b\{x\}\rangle\!\rangle\{\!\{x{\leftarrow}x.rec\}\!\}].rec$$

and obtain the unfolding property $\mu(x)b\{x\} = b\{\!\{\mu(x)b\{x\}\}\!\}$:

$$
\begin{aligned}
&\langle\!\langle \mu(x)b\{x\} \rangle\!\rangle \\
&\quad\equiv\ [rec{=}\varsigma(x)\langle\!\langle b\{x\}\rangle\!\rangle\{\!\{x{\leftarrow}x.rec\}\!\}].rec \\
&\quad=\ \langle\!\langle b\{x\}\rangle\!\rangle\{\!\{x{\leftarrow}x.rec\}\!\}\{\!\{x{\leftarrow}[rec{=}\varsigma(x)\langle\!\langle b\{x\}\rangle\!\rangle\{\!\{x{\leftarrow}x.rec\}\!\}]\}\!\} \\
&\quad\equiv\ \langle\!\langle b\{x\}\rangle\!\rangle\{\!\{x{\leftarrow}[rec{=}\varsigma(x)\langle\!\langle b\{x\}\rangle\!\rangle\{\!\{x{\leftarrow}x.rec\}\!\}].rec\}\!\} \\
&\quad\equiv\ \langle\!\langle b\{x\}\rangle\!\rangle\{\!\{x{\leftarrow}\langle\!\langle \mu(x)b\{x\}\rangle\!\rangle\}\!\} \\
&\quad\equiv\ \langle\!\langle b\{\!\{\mu(x)b\{x\}\}\!\}\rangle\!\rangle
\end{aligned}
$$

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:

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

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

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

A Class for Cells

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

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

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

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

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

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 $\leadsto$.
- We write $\vdash a \leadsto v$ 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 $v \equiv [l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]$)

$$\frac{}{\vdash v \leadsto v}$$

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

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

(Red Update)

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

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 $\vdash a \leadsto v$ and $\vdash a \leadsto v'$, then $v \equiv v'$.

The rules for $\leadsto$ 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 $a \twoheadrightarrow v$.

$\square$

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 $\vdash a \rightsquigarrow v'$.

$\square$

This theorem was proved by Melliès.

An Interpreter

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

The algorithm takes a closed term and, if it converges, produces a result or the token *wrong*, which represents a computation error.

- *Outcome*(c) is the outcome of running the algorithm on input c, assuming the algorithm terminates.
- The algorithm implements the operational semantics in the sense that $\vdash c \leadsto v$ if and only if *Outcome*(c) $\equiv v$ and v is not *wrong*.
- The algorithm either diverges or terminates with a result or with *wrong*, but it does not get stuck.

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

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

$Outcome(a.l_j) \quad \triangleq$

let $o = Outcome(a)$

in if o is of the form $[l_i{=}\varsigma(x_i)b_i\{x_i\}\ ^{i\in 1..n}]$ with $j\in 1..n$

then $Outcome(b_j\{\!\{o\}\!\})$

else $wrong$

$Outcome(a.l_j \Leftarrow \varsigma(x)b) \quad \triangleq$

let $o = Outcome(a)$

in if o is of the form $[l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]$ with $j\in 1..n$

then $[l_j{=}\varsigma(x)b,\ l_i{=}\varsigma(x_i)b_i\ ^{i\in (1..n)-\{j\}}]$

else $wrong$

Objects and Imperative Features

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.

Order of Evaluation

We adopt the following order of evaluation:

- The ς binders suspend evaluation in object creation.
- The method update $a \Leftarrow \varsigma(y)b$ evaluates a, but the ς binder suspends the evaluation of the new method body b.
- The method invocation $a.l$ triggers the evaluation of a (and of further expressions).
- The cloning $clone(a)$ triggers the evaluation of a.
- $let\ x = a\ in\ b$ evaluates a then b.

With the introduction of side-effects, order of evaluation affects not just termination but also output.

Fields, Revisited

Fields need not be primitive in functional calculi, because there we can regard a field as a method that does not use its self parameter.

So we could try, again:

field:	$[\ldots, l{=}b, \ldots] \triangleq [\ldots, l{=}\varsigma(x)b, \ldots]$	for $x \notin FV(b)$
field selection:	$o.l \triangleq o.l$	
field update:	$o.l{:=}b \triangleq o.l \Leftarrow \varsigma(x)b$	for $x \notin FV(b)$

In both field definition and field update, the implicit $\varsigma(x)$ binder suspends evaluation of the field until selection.

- This semantics is inefficient, because at every access the suspended fields are reevaluated.
- This semantics is inadequate, because at every access the side-effects of suspended fields are repeated.

Fields via Let

So we consider an alternative definition for fields, based on *let*.

The *let* construct gives us a way of controlling execution flow:

An object with fields:

$$[l_i{=}b_i\ ^{i\in 1..n},\ l_j{=}\varsigma(x_j)b_j\ ^{j\in n+1..n+m}] \quad \triangleq$$

$$\mathit{let}\ y_1{=}b_1\ \mathit{in}\ \ldots\ \mathit{let}\ y_n{=}b_n\ \mathit{in}\ [l_i{=}\varsigma(y_0)y_i\ ^{i\in 1..n},\ l_j{=}\varsigma(x_j)b_j\ ^{j\in n+1..n+m}]$$

for $y_i \notin FV(b_k\ ^{k\in 1..n+m})$, y_i distinct, $i\in 0..n$

A field selection:

$$a.l \quad \triangleq \quad a.l$$

A field update:

$$a.l{:=}b \quad \triangleq \quad \mathit{let}\ y_1{=}a\ \mathit{in}\ \mathit{let}\ y_2{=}b\ \mathit{in}\ y_1.l \Leftarrow \varsigma(y_0)y_2$$

for $y_i \notin FV(b)$, y_i distinct, $i\in 0..2$

Let via Fields

Conversely, *let* and sequencing can be defined using fields:

$$let\ x{=}a\ in\ b\{x\} \quad \triangleq \quad [def{=}a,\ val{=}\varsigma(x)b\{\!\{x.def\}\!\}].val$$
$$a;\ b \quad \triangleq \quad [fst{=}a,\ snd{=}b].snd$$

Thus we have a choice between a calculus with fields, field selection, and field update, and one with *let* (**imp**ς).

- These calculi are inter-translatable.
- We adopt **imp**ς as primitive because it is more economical and it enables easier comparisons with our other calculi.

A Cell with Undo (Revisited)

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

- 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:

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

(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.

$$
\begin{array}{l}
\textit{let sieve} = \\
\quad [m = \varsigma(s)\ \lambda(n) \\
\qquad\qquad \textit{let sieve'} = \textit{clone}(s) \\
\qquad\qquad \textit{in} \quad s.\textit{prime} := n; \\
\qquad\qquad\qquad s.\textit{next} := \textit{sieve'}; \\
\qquad\qquad\qquad s.m \Leftarrow \varsigma(s')\ \lambda(n') \\
\qquad\qquad\qquad\qquad \textit{case}\ (n'\ \textit{mod}\ n) \\
\qquad\qquad\qquad\qquad \textit{when}\ 0\ \textit{do}\ [], \\
\qquad\qquad\qquad\qquad \textit{when}\ p{+}1\ \textit{do}\ \textit{sieve'}.m(n'); \\
\qquad\qquad\qquad [], \\
\quad \textit{prime} = \varsigma(x)\ x.\textit{prime}, \\
\quad \textit{next} = \varsigma(x)\ x.\textit{next}];
\end{array}
$$

- 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

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

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:

$$[l_i{=}\iota_i \ ^{i\in 1..n}]$$

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:

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

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)

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

(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)

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

(Red Update)

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

(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)

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

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.

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.

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

- The next one is a divergent reduction.

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

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

  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, $\sigma_0, \sigma_1, \ldots$:

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

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

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

- 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.

$\varnothing \bullet \varnothing \vdash [\,] \leadsto [\,] \bullet \varnothing$ by (Red Object)

STUCK

$\varnothing \bullet \varnothing \vdash [\,].l \leadsto ? \bullet ?$ (Red Select)

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

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

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

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

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.

Classes

The treatment classes carries over, with some twists.

Consider a class:

$$\begin{array}{l} let\ c = \\ \quad [new=\varsigma(z)[l_i=\varsigma(s)z.l_i(s)\ ^{i\in 1..n}], \\ \quad\ l_i=\varsigma(z)\lambda(s)b_i\ ^{i\in 1..n}]; \end{array}$$

The class c evaluates to a set of locations $[new=\iota_0,\ l_i=\iota_i\ ^{i\in 1..n}]$ pointing to closures for *new* and the pre-methods l_i.

- When $c.new$ is invoked, a set of locations is allocated for the new object, containing closures for its methods.
- These closures contain the code $\varsigma(s)z.l_i(s)$, where z is bound to $[new=\iota_0,\ l_i=\iota_i\ ^{i\in 1..n}]$.
- When a method of the new object is invoked, the corresponding pre-method is fetched from the class and applied to the object.

Subclasses

Consider a subclass:

$$
\begin{aligned}
&\mathit{let}\ c' = \\
&\qquad [\mathit{new}=\varsigma(z)[l_i=\varsigma(s)z.l_i(s)\ ^{i\in 1..n+m}], \\
&\qquad\ l_j=\varsigma(z)c.l_j\ ^{j\in 1..n}, \\
&\qquad\ l_k=\varsigma(z)\lambda(s)b_k\ ^{k\in n+1..n+m}];
\end{aligned}
$$

When the pre-method l_j is inherited from c to c', the evaluation of $c.l_j$ is suspended by $\varsigma(z)$.

- Therefore, whenever the method l_j is invoked on an instance of c', the pre-method l_j is fetched from c.
- The binders $\varsigma(z)$ suspend evaluation and achieve this dynamic lookup of pre-methods inherited from c.
- When $c'.\mathit{new}$ is invoked, the methods of the new object refer to c' and, indirectly, to c.

Global Change

Suppose that, after c and c' have been created, and after instances of c and c' have been allocated, we replace the pre-method l_1 of c.

- The instances of c reflect the change, because each method invocation goes back to c to fetch the pre-method.
- The instances of c' also reflect the change, via the indirection through c'.
- So the default effect of replacing a pre-method in a class is to modify the behavior of all instances of the class and of classes that inherited the pre-method.
- This default is inhibited by independent updates to objects and to inheriting classes.

Our definition of classes is designed for this global-change effect.

Or No Global Change

If one is not interested in global change, one can optimize the definition and remove some of the run-time indirections.

- In particular, we can replace the proper method $l_j = \varsigma(z) c.l_j$ in the subclass c' with a field $l_j = c.l_j$.

  Then a change to $c.l_j$ after the definition of c' will not affect c' or instances of c'.

- Similarly, we can make $c.new$ evaluate the pre-methods of c, so that a change to c will not affect existing instances.

Combining these techniques, we obtain the following eager variants *e* and *e'* of *c* and *c'*:

$$
\begin{aligned}
&let\ e = \\
&\quad [new=\varsigma(z)let\ w_1=z.l_1\ in\ \ldots\ let\ w_n=z.l_n\ in\ [l_i=\varsigma(s)w_i(s)\ ^{i\in 1..n}], \\
&\quad\ l_i=\varsigma(z)\lambda(s)b_i\ ^{i\in 1..n}];
\end{aligned}
$$

$$
\begin{aligned}
&let\ e' = \\
&\quad [new=\varsigma(z)let\ w_1=z.l_1\ in\ \ldots\ let\ w_{n+m}=z.l_{n+m}\ in \\
&\qquad [l_i=\varsigma(s)w_i(s)\ ^{i\in 1..n+m}], \\
&\quad\ l_j=e.l_j\ ^{j\in 1..n}, \\
&\quad\ l_k=\varsigma(z)\lambda(s)b_k\ ^{k\in n+1..n+m}];
\end{aligned}
$$

Imperative Examples of Classes

We define classes cp_1 and cp_2 for one-dimensional and two-dimensional points:

$$
\begin{array}{l}
let\ cp_1 = \\
\quad [new = \varsigma(z)[\ldots], \\
\quad\ x = \varsigma(z)\ \lambda(s)\ 0, \\
\quad\ mv\_x = \varsigma(z)\ \lambda(s)\ \lambda(dx)\ s.x := s.x{+}dx]; \\
let\ cp_2 = \\
\quad [new = \varsigma(z)[\ldots], \\
\quad\ x = \varsigma(z)\ cp_1.x, \\
\quad\ y = \varsigma(z)\ \lambda(s)\ 0, \\
\quad\ mv\_x = \varsigma(z)\ cp_1.mv\_x, \\
\quad\ mv\_y = \varsigma(z)\ \lambda(s)\ \lambda(dy)\ s.y := s.y{+}dy]
\end{array}
$$

We define points p_1 and p_2 by generating them from cp_1 and cp_2:

$$\mathit{let}\ p_1 = cp_1.\mathit{new};$$
$$\mathit{let}\ p_2 = cp_2.\mathit{new};$$

We change the mv_x pre-method of cp_1 so that it does not set the x coordinate of a point to a negative number:

$$cp_1.mv\_x \Leftarrow \varsigma(z)\ \lambda(s)\ \lambda(dx)\ s.x := max(s.x+dx, 0)$$

- The update is seen by p_1 because p_1 was generated from cp_1.
- The update is seen also by p_2 because p_2 was generated from cp_2 which inherited mv_x from cp_1:

$$p_1.mv\_x(-3).x \quad = \quad 0$$
$$p_2.mv\_x(-3).x \quad = \quad 0$$

A First-Order Type System for Objects

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.

For example, $[]$ and $[l_1 : [], l_2 : []]$ are object types.

Subtyping

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

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

For example, we have:

$$[l_1 : [], l_2 : []] <: [l_1 : []] <: []$$

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 Type System

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 ø)

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

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

$$\frac{}{\vdash v \leadsto v}$$

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

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

(Red Update)

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

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.

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

$$\begin{aligned}
&true_A : Bool_A \quad \triangleq \\
&\qquad [if = \varsigma(x{:}Bool_A)\ x.then, \\
&\qquad\ then = \varsigma(x{:}Bool_A)\ x.then, \\
&\qquad\ else = \varsigma(x{:}Bool_A)\ x.else]
\end{aligned}$$

$$\begin{aligned}
&false_A : Bool_A \quad \triangleq \\
&\qquad [if = \varsigma(x{:}Bool_A)\ x.else, \ldots]
\end{aligned}$$

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:

$$\begin{aligned} &if_A\ b\ then\ c\ else\ d : A \quad \triangleq \\ &\qquad ((b.then \Leftarrow \varsigma(x{:}Bool_A)c).else \Leftarrow \varsigma(x{:}Bool_A)\ d).if \end{aligned}$$

Moreover, we get some subtypings, for example:

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

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*

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

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 $\vdash a \leadsto v$
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 (MinOb$_{1<:}$ typings are Ob$_{1<:}$ typings)

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

□

Proposition (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 (MinOb$_{1<:}$ has smaller types than 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 (Ob$_{1<:}$ has minimum types)

In **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 **MinOb$_{1<:}$** for some B such that $E \vdash B <: A$.

Hence, $E \vdash a : B$ is also derivable in **Ob$_{1<:}$**.

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

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

$\square$

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:

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

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 $\varnothing \vdash a : A$ but not using $\varnothing \vdash a : A'$.

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 see Palsberg's work.)

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 $\vdash c \leadsto v$.

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

Proof

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

Case (Red Object)

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

Case (Red Select)

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

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

This must have come from an application of (Val Select)

- with assumption $\varnothing \vdash a : A$ where A has the form $[l_j{:}B_j, \ldots]$, and
- with conclusion $\varnothing \vdash a.l_j : B_j$,

followed by a number of subsumption steps implying $\varnothing \vdash B_j <: C$ by transitivity.

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

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

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

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

Case (Red Update)

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

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.

Type Soundness

The subject reduction theorem does not rule out that the execution of a well-typed program may not terminate or may get stuck.

We can prove that the latter is in fact not possible:

- If the reduction does not diverge, then it produces a result of the correct type without getting stuck.
- This absence of stuck states is often called *type soundness*.

In order to formulate a type soundness result, we reconsider the function *Outcome*:

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

$$Outcome(a.l_j) \quad \triangleq$$
$$\quad \text{let } o = Outcome(a)$$
$$\quad \text{in} \quad \text{if } o \text{ is of the form } [l_i{=}\varsigma(x_i{:}A_i)b_i\{x_i\}^{\ i\in 1..n}] \text{ with } j\in 1..n$$
$$\qquad\quad \text{then } Outcome(b_j\{\!\{o\}\!\})$$
$$\qquad\quad \text{else } wrong$$

$$Outcome(a.l_j \Leftarrow \varsigma(x{:}A)b) \quad \triangleq$$
$$\quad \text{let } o = Outcome(a)$$
$$\quad \text{in} \quad \text{if } o \text{ is of the form } [l_i{=}\varsigma(x_i{:}A_i)b_i^{\ i\in 1..n}] \text{ with } j\in 1..n$$
$$\qquad\quad \text{then } [l_j{=}\varsigma(x{:}A_j)b,\ l_i{=}\varsigma(x_i{:}A_i)b_i^{\ i\in (1..n)-\{j\}}]$$
$$\qquad\quad \text{else } wrong$$

If $\mathit{Outcome}(c)$ is defined, then it is either *wrong* or a result.

We obtain:

Theorem (Reductions cannot go wrong)

If $\emptyset \vdash c : C$ and $\mathit{Outcome}(c)$ is defined, then $\emptyset \vdash \mathit{Outcome}(c) : C$, hence $\mathit{Outcome}(c)$ ¦ wrong.

The proof is by induction on the execution of $\mathit{Outcome}(c)$, and is very similar to the proof of subject reduction.

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!

Unsoundness of Method Extraction

Let us imagine an operation for extracting a method from an object.

It may seem natural to give the following rules for this operation:

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

$$\frac{E \vdash a : A \qquad j\in 1..n}{E \vdash a{\cdot}l_j : A{\rightarrow}B_j}$$

(Red Extract)

$$\frac{\vdash a \rightsquigarrow [l_i{=}\varsigma(x_i{:}A_i)b_i{}^{i\in 1..n}] \qquad j\in 1..n}{\vdash a{\cdot}l_j \rightsquigarrow \lambda(x_j{:}A_j)b_j}$$

These rules amount to interpreting an object type $A \equiv [l_i{:}B_i{}^{i\in 1..n}]$ as a recursively defined record-of-functions type $A \equiv \langle l_i{:}A{\rightarrow}B_i{}^{i\in 1..n}\rangle$.

Method extraction is unsound, as the following example shows:

$P \triangleq [x{:}Int,\ f{:}Int]$	
$p \triangleq [x{=}1,\ f{=}1]$	$p : P$ by (Val Object)
$Q \triangleq [x,y{:}Int,\ f{:}Int]$	$Q <: P$ by (Sub Object)
$a \triangleq [x{=}1,\ y{=}1,\ f{=}\varsigma(s{:}Q)s.x+s.y]$	$a : Q$ by (Val Object),
	so $a : P$ by subsumption
$b \triangleq a{\cdot}f$	$b : P \rightarrow Int$ by (Val Extract),
	and $b(p) : Int$

But $b(p)$ must yield an execution error since p lacks a y method.

Hence method extraction is incompatible with subtyping, at least without much further complication or strong restrictions.

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.

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

- 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

$$
\begin{aligned}
&Class(Cell) \triangleq \\
&\quad [new : Cell, \\
&\quad\ \ contents : Cell \to Nat, \\
&\quad\ \ set : Cell \to Nat \to []]
\end{aligned}
$$

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

$Class(GCell)$ <: $Class(Cell)$ 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:

$$a \neq b : A$$

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).

Variance Annotations

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 $B' <: B$	

We get *depth subtyping* as well as *width subtyping*.

Subtyping Rules with Variance Annotations

We use an auxiliary judgment: $E \vdash \upsilon_i\, B_i <: \upsilon_i'\, B_i'$

(Sub Object)

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

(Sub Invariant)

$$\frac{E \vdash B}{E \vdash {}^{o} B <: {}^{o} B}$$

(Sub Covariant)

$$\frac{E \vdash B <: B' \quad \upsilon \in \{{}^{o},{}^{+}\}}{E \vdash \upsilon\, B <: {}^{+} B'}$$

(Sub Contravariant)

$$\frac{E \vdash B' <: B \quad \upsilon \in \{{}^{o},{}^{-}\}}{E \vdash \upsilon\, B <: {}^{-} B'}$$

- (Sub Invariant) An invariant component type on the right requires an identical one on the left.
- (Sub Covariant) A covariant component type on the right can be a supertype of a corresponding component type on the left, either covariant or invariant.
- (Sub Contravariant) A contravariant component type on the right can be a subtype of a corresponding component type on the left, either contravariant or invariant.

Typing Rules with Variance Annotations

The typing rules are easy modifications of the previous ones.

They enforce the read/write restrictions:

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

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

(Val Select)

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

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

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

The rule (Val Object) is unchanged, since we add annotations only to object types, not to objects.

Protection by Subtyping

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

For example:

$$\begin{array}{ll}
\text{Let} & GCell \triangleq [contents : Nat, set : Nat \to [], get : Nat] \\
& PGCell \triangleq [set : Nat \to [], get : Nat] \\
& ProtectedGCell \triangleq [set^+ : Nat \to [], get^+ : Nat] \\
& gcell : GCell \\
\text{then} & GCell <: PGCell <: ProtectedGCell \\
\text{so} & gcell : ProtectedGCell.
\end{array}$$

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

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

Protection for Classes

Using subtyping, we can provide protection for classes.

We may associate two separate interfaces with a class type:

- The first interface is the collection of methods that are available in instances.
- The second interface is the collection of methods that can be inherited in subclasses.

For an object type $A \equiv [l_i{:}B_i{}^{i \in I}]$ with methods $l_i{}^{i \in I}$ we consider:

- a restricted *instance interface*, determined by a set $Ins \subseteq I$, and
- a restricted *subclass interface*, determined by a set $Sub \subseteq I$.

For an object type $A \equiv [l_i{:}B_i{}^{i\in I}]$, and Ins, $Sub \subseteq I$, we define:

$$Class(A)_{Ins,Sub} \;\triangleq\; [new^{+}{:}[l_i{:}B_i{}^{i\in Ins}],\ l_i{:}A\rightarrow B_i{}^{i\in Sub}]$$

$Class(A)$ <: $Class(A)_{Ins,Sub}$ holds, so we get protection by subsumption.

Particular values of Ins and Sub correspond to common situations.

$c : Class(A)_{\emptyset,Sub}$	is an abstract class based on A
$c : Class(A)_{Ins,\emptyset}$	is a leaf class based on A
$c : Class(A)_{I,I}$	is a concrete class based on A
$c : Class(A)_{Pub,Pub}$	has public methods $l_i{}^{i\in Pub}$ and private methods $l_i{}^{i\in I-Pub}$
$c : Class(A)_{Pub,Pub\cup Pro}$	has public methods $l_i{}^{i\in Pub}$, protected methods $l_i{}^{i\in Pro}$, and private methods $l_i{}^{i\in I-Pub\cup Pro}$

Class Types for Cells (with Protection)

$$ProtectedGCell \triangleq [set^{+} : Nat \to [], get^{+} : Nat]$$

$$\begin{aligned} &Class...(GCell) \triangleq \\ &\quad [new^{+} : ProtectedGCell, \\ &\quad\ set : GCell \to Nat \to [], \\ &\quad\ get : GCell \to Nat] \end{aligned}$$

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

$$Class(GCell) <: Class...(GCell)$$

(This is a variant on the general scheme.)

Encoding Function Types

An invariant translation of function types:

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

A covariant/contravariant translation, using annotations:

$$\langle\!\langle A \to B \rangle\!\rangle \quad \triangleq \quad [arg^{-} : \langle\!\langle A \rangle\!\rangle, val^{+} : \langle\!\langle B \rangle\!\rangle]$$

A covariant/contravariant translation, using quantifiers:

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

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

Recursive Object Types

Recursive Types

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

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

Formally, we write instead:

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

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

There are at least two ways of formalizing this intuitive idea:

If $a : A$ and $A = B$ then $a : B$.

and:

If $a : \mu(X)A\{X\}$ then $unfold(a) : A\{\!\{\mu(X)A\{X\}\}\!\}$.

If $a : A\{\!\{\mu(X)A\{X\}\}\!\}$ then $fold(\mu(X)A\{X\}, a) : \mu(X)A\{X\}$.

Officially, we adopt the second way (but often omit *fold* and *unfold*.)

Typing Examples with Recursive Types

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

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

The type *Cell* is a recursive type.

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

Similarly, we can typecheck the calculator, using the type:

$$Calc \triangleq$$
$$\mu(X)[arg, acc : Real, enter : Real \to X, add, sub : X, equals : Real]$$

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 $\mu(X)A\{X\} <: \mu(X)B\{X\}$.

Subtyping Examples with Recursive Types

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

$$
\begin{aligned}
Cell &\triangleq [contents : Nat, set : Nat \to Cell] \\
GCell &\triangleq [contents : Nat, set : Nat \to GCell, get : Nat]
\end{aligned}
$$

Assume $X <: Y$.

We **cannot** derive:

$$
\begin{gathered}
[contents : Nat, set : Nat \to X, get : Nat] \\
<: \\
[contents : Nat, set : Nat \to Y]
\end{gathered}
$$

So we cannot obtain that *GCell* is 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*:

$$\begin{aligned} Cell &\triangleq [contents : Nat, set : Nat \rightarrow Cell] \\ GCell &\triangleq [contents : Nat, set : Nat \rightarrow Cell, get : Nat] \end{aligned}$$

  ~ 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:

$$
\begin{aligned}
Cell &\triangleq [contents : Nat, set : Nat \to Self] \\
GCell &\triangleq [contents : Nat, set : Nat \to Self, get : Nat]
\end{aligned}
$$

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

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

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

  ~ 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.

TYPECASE

A Typecase Construct

Adding a **typecase** construct is one way of incorporating dynamic typing in a statically typed language.

There are several variants of this construct; we will study only one.

Our **typecase** construct evaluates a term to a result, and branches on the type of the result. We write:

$$typecase\ a \mid (x{:}A)d_1 \mid d_2$$

- If a yields a result of type A, then $(typecase\ a|(x{:}A)d_1|d_2)$ returns d_1 with x replaced by this result.
- Otherwise, $(typecase\ a|(x{:}A)d_1|d_2)$ returns d_2.

E.g.,

typecase gcell.set(3)
| (*x*:*GCell*) *x.get*
| *x.contents*

typecase gcell.set(3)
| (*x*:*GCell*) *x.get*
| 0

typecase gcell.set(3)
| (*x*:*GCell*) *x.get*
| ... (some error code or exception)

Typecase: Operational Semantics

- In programming languages that include *typecase*, the type of a value is represented using a tag attached to the value.

  typecase relies on this tag to perform run-time type discrimination.

- In contrast, in our operational semantics, *typecase* performs run-time type discrimination by constructing a typing derivation.

Operational semantics for *typecase*

(Red Typecase Match)

$$\frac{\vdash a \leadsto v' \qquad \emptyset \vdash v' : A \qquad \vdash d_1\{\!\{v'\}\!\} \leadsto v}{\vdash typecase\ a|(x{:}A)d_1\{x\}|d_2 \leadsto v}$$

(Red Typecase Else)

$$\frac{\vdash a \leadsto v' \qquad \emptyset \nvdash v' : A \qquad \vdash d_2 \leadsto v}{\vdash typecase\ a|(x{:}A)d_1|d_2 \leadsto v}$$

~ Our rules do not clearly suggest an efficient implementation.

~ They have the advantage of being simple and general.

Typecase: Typing

Typing rule for *typecase*

(Val Typecase)

$$\frac{E \vdash a : A' \qquad E, x{:}A \vdash d_1 : D \qquad E \vdash d_2 : D}{E \vdash \textit{typecase}\ a|(x{:}A)d_1|d_2 : D}$$

The first hypothesis says that a is well-typed; the precise type (A') is irrelevant.

The body of the first branch, d_1, is typed under the assumption that x has type A.

The two branches have the same type, D, which is also the type of the whole *typecase* expression.

Although *typecase* permits dynamic typing, the static typing rules remain consistent.

It is straightforward to extend our subject reduction proof to *typecase*.

Typecase: Discussion

typecase may seem simple, but it is considered problematic (both methodologically and theoretically):

~ It violates the object abstraction, revealing information that may be regarded as private.

~ It renders programs more fragile by introducing a form of dynamic failure when none of the branches apply.

~ It makes code less extensible: when adding another type one may have to revisit the **typecase** statements in existing code.

~ It violates uniformity (parametricity) principles.

Although **typecase** may be ultimately an unavoidable feature, its drawbacks require that it be used prudently.

The desire to reduce the uses of **typecase** has shaped much of the type structure of object-oriented languages.

THE LANGUAGE O–1

Synthesis of a Language

- O–1 is a language built out of constructs from object calculi.
  - ~ The main purpose of O–1 is to help us assess the contributions of object calculi.
  - ~ In addition, O–1 embodies a few intriguing language-design ideas.
  - ~ We have studied more advanced languages that include Self types and parametric polymorphism.

Some 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. (To compensate for, e.g., lack of Self types.)
- Separation of interfaces from implementations.
- Separation of inheritance from subtyping.

Some Non-Features of O–1

- No public/private/protected/abstract, etc.,
- No cloning,
- No basic types, such as integers,
- No arrays and other data structures,
- No procedures,
- No concurrency.

Syntax of Types

Syntax of O–1 types

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

- Roughly, we may think **Object** = μ.
  But the fold/unfold coercions do not appear in the syntax of O–1.
- Usually, $^+$ variance is for methods, and ^{o} variance is for fields.

Syntax of Programs

Syntax of O–1 terms

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

Comments

- Superclass attributes are inherited “automatically”.
  (No copying premethods by hand as in the encodings of classes.)

- Inheritance “by hand” still possible by class selection $c\hat{\ }l(a)$.

- Classes are first-class values.

- Parametric classes can be written as functions that return classes.

Language Fragments

- 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).
  The result would be a little object-based language.

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** $(x{:}A)$ … $c\hat{\ }l(x)$ … **end**

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

  object$(x{:}A)$ … $l{=}c\hat{\ }l(x)$ … **end**

N.B.: conversely, **subclass** could be defined from **class** and $c\hat{\ }l$.

Examples: Types and Classes

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

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

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

- $CPoint <: Point$

- 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)

$$
\begin{array}{l}
\mathit{pointClass} : \mathbf{Class}(\mathit{Point}) \quad \triangleq \\
\qquad \mathbf{class\ with}\ (\mathit{self}\colon \mathit{Point}) \\
\qquad\qquad x = 0, \\
\qquad\qquad \mathit{eq} = \mathbf{fun}(\mathit{other}\colon \mathit{Point})\ \mathit{self}.x = \mathit{other}.x\ \mathbf{end}, \\
\qquad\qquad \mathit{mv} = \mathbf{fun}(\mathit{dx}\colon \mathit{Int})\ \mathit{self}.x := \mathit{self}.x{+}\mathit{dx}\ \mathbf{end} \\
\qquad \mathbf{end}
\end{array}
$$

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*.

Creating Objects

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

$$cPoint : CPoint \triangleq \textbf{new}\ cPointClass$$

- But points of the same type can also be created independently:

$cPoint' : CPoint \triangleq$

object(*self*: *CPoint*)

x = 0,

c = *red*,

eq = **fun**(*other*: *Point*) *other.eq*(*self*) **end**,

mv = *cPointClass*^*mv*(*self*)

end

Using Objects

- 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:

$$
\begin{array}{l}
\mathit{movedColor} : \mathit{Color} \quad \triangleq \\
\qquad \textbf{typecase}\ \mathit{cPoint}.\mathit{mv}(1) \\
\qquad \textbf{when}\ (\mathit{cp}\colon \mathit{CPoint})\ \mathit{cp}.\mathit{c} \\
\qquad \textbf{else}\ \mathit{black} \\
\qquad \textbf{end}
\end{array}
$$

Alternative Types

- A stronger type of color points that would preserve type information on move is:

$$\begin{aligned}&CPoint2 \quad \triangleq \\ &\qquad \mathbf{Object}(X)[x\colon Int,\ c\colon Color,\ eq^{+}\colon Point{\rightarrow}Bool,\ mv^{+}\colon Int{\rightarrow}X]\end{aligned}$$

  CPoint2 <: *Point*, by the read-only annotation on *mv*.

- To define a subclass for *CPoint2*, one must override *mv*. Subtyping does not imply inheritability!

- The new code for *mv* may be just **super**.*mv* followed by a typecase.

$cPointClass2 : \mathbf{Class}(CPoint2) \triangleq$

```
cPointClass2 : Class(CPoint2)  ≜
    subclass of pointClass: Class(Point)
    with (self: CPoint2)
        c = black
    override
        eq = fun(other: Point) ... end,
        mv = fun(dx: Int)
                typecase super.mv(dx)
                when (res: CPoint2) res
                else … (error)
                end
            end
    end
```

- But typecase is no longer needed after a color point is moved:

$$\mathit{cPoint2} : \mathit{CPoint2} \triangleq \textbf{new}\ \mathit{cPoint2Class}$$

$$\mathit{movedColor2} : \mathit{Color} \triangleq \mathit{cPoint2.mv}(1).\mathit{c}$$

- By switching from *CPoint* to *CPoint2* we have shifted typecase from the code that uses color points to the code that creates them.
- This shift may be attractive, for example because it may help in localizing the use of typecase.

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)

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

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

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

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

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

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

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

(Val New)

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

(Val Root)

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

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

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

(Val Typecase)

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

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

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

- A is the object type for the subclass.
- A' is the object type for the superclass.
- Ovr is the set of indices of overridden methods.
- $E \vdash A <: A'$ The “class rule”, means that:
  “type generated by subclass <: type generated by superclass”
  Allows “method specialiazation” $l_i^+{:}B_i <: l_i^+{:}B_i'$ for $i \in Ovr$
- $E \vdash B_i'\{\!\{A'\}\!\} <: B_i\{\!\{A\}\!\}$ Toghether with $A <: A'$ requires type invariance for an inherited method. If this condition does not hold, the method must be overridden.
- $E, x{:}A \vdash b_i : B_i\{\!\{A\}\!\}$ Checking the bodies of overridden and additional methods.

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 $\mathbf{Object}(X)[l_i\upsilon_i{:}B_i\ ^{i\in 1..n}]$ 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 $\mathbf{Class}(\mathbf{Object}(X)[l_i\upsilon_i{:}B_i\{X\}\ ^{i\in 1..n}])$ 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 Programs

- Officially, the translation is guided by the type structure.
- Most of the clauses are straightforward.
- A class is mapped to an object with a collection of pre-methods plus a *new* method.
- **new** c is interpreted as an invocation of the *new* method of $\langle\!\langle c \rangle\!\rangle$.

(Simplified) Translation of O–1 terms

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

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

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

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

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

$\langle\!\langle \textbf{new}\ c \rangle\!\rangle \;\triangleq\; \langle\!\langle c \rangle\!\rangle.\mathit{new}$

$\langle\!\langle \textbf{root} \rangle\!\rangle \;\triangleq\; [\mathit{new}=[]]$

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

$\langle\!\langle c\hat{\ }l(a) \rangle\!\rangle \;\triangleq\; \langle\!\langle c \rangle\!\rangle.l(\langle\!\langle a \rangle\!\rangle)$

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

- 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 $c\hat{\ }l(a)$ is interpreted as the extraction and the application of a pre-method.

Soundness

- If E ⊢ J is valid in O–1, then 《E ⊢ J》 is valid in the object calculus.

- The object subtyping rule relies on the following rule for recursive types:

(Sub Rec')

$$\frac{E \vdash \mu(X)A\{X\} \qquad E \vdash \mu(Y)B\{Y\} \qquad E, X{<:}\mu(Y)B\{Y\} \vdash A\{X\} <: B\{\!\{\mu(Y)B\{Y\}\}\!\}}{E \vdash \mu(X)A\{X\} <: \mu(Y)B\{Y\}}$$

- The most interesting case is for **subclass**. We need to check:

《**subclass of** c':**Class**(A') **with**(x:A) l_i=b_i $^{i \in n+1..n+m}$ **override** l_i=b_i $^{i \in Ovr}$ **end**》

: 《**Class**(A)》

That is:

$$
\begin{array}{l}
[new{=}\varsigma(z{:}\langle\!\langle \mathbf{Class}(A)\rangle\!\rangle)[l_i{=}\varsigma(s{:}\langle\!\langle A\rangle\!\rangle)z.l_i(s)\ ^{i\in 1..n+m}],\\
\qquad l_i{=}\langle\!\langle c'\rangle\!\rangle.l_i\ ^{i\in 1..n-Ovr},\\
\qquad l_i{=}\lambda(x{:}\langle\!\langle A\rangle\!\rangle)\langle\!\langle b_i\rangle\!\rangle\ ^{i\in Ovr\cup n+1..n+m}]\\
: [new^+{:}\langle\!\langle A\rangle\!\rangle,\ l_i^+{:}\langle\!\langle A\rangle\!\rangle\to\langle\!\langle B_i\rangle\!\rangle\{\langle\!\langle A\rangle\!\rangle\}\ ^{i\in 1..n}]
\end{array}
$$

- *new* checks by computation.
- $i\in Ovr\cup n+1..n+m$ checks by one the (Val Subclass) hypotheses.
- $i\in 1..n-Ovr$ (inherited methods) checks as follows:

  $\langle\!\langle c'\rangle\!\rangle : \langle\!\langle \mathbf{Class}(A')\rangle\!\rangle$ by hypothesis. Hence:

  $\langle\!\langle c'\rangle\!\rangle.l_i : \langle\!\langle A'\rangle\!\rangle\to\langle\!\langle B_i'\rangle\!\rangle\{\langle\!\langle A'\rangle\!\rangle\}$. Moreover:

  $\langle\!\langle A'\rangle\!\rangle\to\langle\!\langle B_i'\rangle\!\rangle\{\langle\!\langle A'\rangle\!\rangle\} <: \langle\!\langle A\rangle\!\rangle\to\langle\!\langle B_i\rangle\!\rangle\{\langle\!\langle A\rangle\!\rangle\}$ directly from hypotheses. So:

  $\langle\!\langle c'\rangle\!\rangle.l_i : \langle\!\langle A\rangle\!\rangle\to\langle\!\langle B_i\rangle\!\rangle\{\langle\!\langle A\rangle\!\rangle\}$ by subsumption.

Usefulness of the Translation

- The translation validates the typing rules of O–1.
  If $E \vdash J$ is valid in O–1, then $\langle\langle E \vdash J \rangle\rangle$ 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.

Polymorphism

Types of Polymorphism

Polymorphic values have (or can be instantiated to have) more than one type.

In particular, polymorphic functions can be applied to arguments of more than one type.

There are several kinds of polymorphism:

- Ad hoc polymorphism,

  as for the functions + and print.
- Subtype (or inclusion) polymorphism,

  as for operations on objects.
- Parametric polymorphism,

  as for the functions identity and append.

(See Strachey.)

Ad Hoc Polymorphism

Ad hoc polymorphism arises in many forms in practical languages.

- Functions like + and print run different code and behave in fairly different ways depending on the types of their arguments.

  The notations + and print are overloaded.

- Ad hoc polymorphism is not "true" polymorphism in that the overloading is purely syntactic.

  (E.g., it will not enhance the computational power of a language.)

- Overloading is sometimes combined, and confused, with implicit type coercions.
- Because of its ad hoc nature, there is not much general we can say about ad hoc polymorphism (except "be careful").

Subtype (or Inclusion) Polymorphism

Much as with ad hoc polymorphism, the invocation of a method on an object may run different code depending on the type of the object.

However,

- the polymorphism of languages with subtyping is systematic, and
- code that manipulates objects is uniform, although objects of different types may have different memory requirements.

Parametric Polymorphism

Parametric polymorphism is usually considered the cleanest and purest form of polymorphism.

- Parametrically polymorphic code uses no type information.

  This uniformity implies that parametrically polymorphic code often works with an extra level of indirection, or "boxing".

- Parametric polymorphism has a rich theory.

  (See Girard, Reynolds, many others.)

Languages with Parametric Polymorphism

Parametric polymorphism appears in a few languages:

- languages with generic functions,
- CLU,
- ML and its relatives,
- languages for logical proof systems.

Typically, these languages limit parametric polymorphism:

- it is sometimes eliminated at compile time or link time,
- it is usually less general than in theoretical presentations.

Some Advantages and Disadvantages of Parametric Polymorphism

Advantages:

- less code duplication,
- stronger type information,
- in some contexts, more computational power.

Disadvantages:

- run-time cost of extra indirections,
- complexity (for example, in combination with side-effects).

These disadvantages have in part been addressed in recent research.

Expressing Parametric Polymorphism

In the general case, parametric polymorphism can be expressed through universal type quantifiers.

For example:

$\forall(X)\ X \to X$	the type of the identity function
$\forall(X)\ \forall(Y)\ X \times Y \to Y \times X$	the type of a permutation function
$\forall(X)\ List(X) \to List(X)$	the type of the reverse function
$\forall(X)\ X$	a “very small” type
$\forall(X)\ (X \to X) \to (X \to X)$	the type of Church numerals

Writing Polymorphic Values

Type parameterization permits writing polymorphic expressions that have types with universal quantifiers.

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

$id(\forall(X)\, X \to X) : (\forall(X)\, X \to X) \to (\forall(X)\, X \to X)$

$id(\forall(X)\, X \to X)(id) : \forall(X)\, X \to X$

Another example:

$$p : \forall(X)\, \forall(Y)\, X{\times}Y{\rightarrow}Y{\times}X \quad \triangleq \quad \lambda(X)\, \lambda(Y)\, \lambda(u{:}X{\times}Y)\, \langle snd(u), fst(u)\rangle$$

$$p(Int) : \forall(Y)\, Int{\times}Y{\rightarrow}Y{\times}Int$$

$$p(Int)(Bool) : Int{\times}Bool{\rightarrow}Bool{\times}Int$$

$$p(Int)(Bool)(\langle 3, true\rangle) : Bool{\times}Int$$

Writing Polymorphic Values: Definitions

- 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.
- 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 $B\{\!\{A\}\!\}$.

Rules for the Universal Quantifier

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

- (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.
- These rules should be complemented with standard rules for forming environments with type variables.

Semantics of Parametric Polymorphism

Intuitively, we may want to define the meaning of types inductively:

Int	=	the integers
$A{\times}B$	=	the pairs of A's and B's
$A{\rightarrow}B$	=	the functions from A to B
$\forall(X)A$	=	the intersection of A for all possible values of X

but this last clause assumes that we know in advance the set of sets over which X ranges!

- Reynolds first conjectured that this could be made to work, but later proved that substantial restrictions and changes are needed.
- Much research on the semantics of polymorphism followed; e.g.:

Int	=	the integers, plus an undefined value
$A{\times}B$	=	the pairs of A's and B's
$A{\rightarrow}B$	=	the "continuous" functions from A to B
$\forall(X)A$	=	the intersection of A for all "ideals" X

ML-Style Polymorphism

In ML, parametric polymorphism is somewhat restricted:

- Type schemes are distinguished from ordinary, simple types (without quantifiers).

Two-level syntax of types

$A, B ::=$	ordinary types
$\quad X$	type variables
$\quad Int$	base types
$\quad A \to B$	function types
$\quad \ldots$	
$C ::=$	type schemes
$\quad A$	ordinary types
$\quad \forall(X)C$	quantified types

- Type quantification ranges over simple types, so $\lambda(X)b$ cannot be instantiated with a type scheme.

ML polymorphism is said to be predicative.

Predicative polymorphism is simpler semantically, and also has some practical appeal:

- Type instantiations and parameterizations need not be written explicitly in programs, but can be inferred.
- Type inference is decidable, and efficient in practice.

ML-Style Polymorphism and Ref

let r = *ref* [ ] *in*	r is a ref to an empty list: $\forall(X)\ Ref\ List(X)$
r := [1] ;	r is a ref to an integer list: $Ref\ List(Int)$
if head(!r)	but r is used as a ref to a boolean list!
then ...	
else ...	

- ML-style polymorphism often has problematic interactions with imperative features.
- These interactions have led to significant restrictions and to sophisticated type systems.
- In contrast, explicit polymorphism is more easily combined with imperative features.

Bounded Parametric Polymorphism

The requirement for a function to work with an arbitrary parameter X is sometimes relaxed.

- Suppose that we want to write a filter function for a type $List(X)$.

  We would like to assume a boolean test operation on X.

- One solution is to let the filter function take the test as argument.

$$filter : \forall(X)\ (X \to Bool) \to (List(X) \to List(X))$$

- Another solution is to restrict the kind of X that can be passed, through a bound or constraint.

$$filter : \forall(X \text{ } with \text{ } method \text{ } test : Bool)\ (List(X) \to List(X))$$

  This idea leads to various forms of bounded polymorphism (as in CLU, Theta, Haskell, Modula-3).

  There is debate about their relative merits and expressiveness.

A 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<:)

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

(The subtyping relation is no longer decidable!)

Structural Update for Objects

When we combine bounded parametric polymorphism and objects, 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, $\forall(C<:[l:Nat])\ C\rightarrow C$ contains only an identity function and its approximations.
- $\forall(C<:[l:Nat])\ C\rightarrow C$ does not contain $\lambda(C<:[l:Nat])\ \lambda(a:C)\ a.l:=3$.

This difficulty suggests that one should proceed with caution.

Data Abstraction

Much like the universal quantifier gives parametric polymorphism, the existential quantifier gives a form of data abstraction.

The existentially quantified type $\exists(X)B\{X\}$ is the type of the pairs $\langle A,b\rangle$ where A is a type and b is a term of type $B\{\!\{A\}\!\}$.

- The type $\exists(X)B\{X\}$ can be seen as an abstract data type with *interface* $B\{X\}$ and with *representation type* X.
- The pair $\langle A,b\rangle$ describes an element of the abstract data type with representation type A and *implementation* b.

(See Mitchell and Plotkin.)

For example, we may write the type:

$\exists(X)\ (Int \rightarrow X) \times (X \rightarrow Int)$ the type of a "store" for an integer

An implementation of this type might be a pair of a sign bit and a natural number, an integer, or a natural number, for example.

A user of the implementation can access it only through the interface.

A 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:

$$p : \exists(X{<:}Int)X\times(X\to X) \quad \triangleq$$
$$\qquad pack\ X{<:}Int{=}Nat\ with\ \langle 0, succ_{Nat}\rangle : X\times(X\to X)$$

$$a : Int \quad \triangleq$$
$$\qquad open\ p\ as\ X{<:}Int, x{:}X\times(X\to X)\ in\ snd(x)(fst(x)){:}Int$$

and then $a = 1$.

Rules for the Bounded Existential Quantifier

(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<:)

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

(Val Open<:)

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

Objects, Parametric Polymorphism, and Data Abstraction

There have been some languages with reasonable, successful combinations of parametric polymorphism and data abstraction.

Less is known about how to add objects.

- Are objects redundant?
  - ~ Objects provide a kind of polymorphism.
  - ~ Objects provide a kind of data abstraction, too.
- How should objects interact with a module (or package) system?

  (Cf. Modula-3 and Java.)
- Is type inference feasible for languages with both objects and parametric polymorphism?

  There have been many proposals for object-oriented extensions to ML, and some for extensions of Java with parametric polymorphism.

Self Quantifier

Second-Order Calculi

Take a first-order object calculus with subtyping, and add bounded quantifiers:

Bounded universals: (contravariant in the bound)

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

Bounded existentials: (covariant in the bound)

$E \vdash \exists(X<:A)B$	if	$E, X<:A \vdash B$
$E \vdash \exists(X<:A)B <: \exists(X<:A')B'$	if	$E \vdash A <: A'$ and $E, X<:A \vdash B <: B'$

$E \vdash (pack\ X<:A=C, b\{X\}:B\{X\}) : \exists(X<:A)B\{X\}$

if $E \vdash C <: A$ and $E \vdash b\{C\} : B\{C\}$

$E \vdash (open\ c\ as\ X<:A,x:B\ in\ d:D) : D$

if $E \vdash c : \exists(X<:A)B$ and $E \vdash D$ and $E, X<:A, x:B \vdash d : D$

Covariant Components

Suppose we have:

$$\begin{array}{lll} \mathit{Point} & \triangleq & [x,y\colon \mathit{Real}] \\ \mathit{ColorPoint} <: \mathit{Point} & \triangleq & [x,y\colon \mathit{Real}, c\colon \mathit{Color}] \\ \mathit{Circle} & \triangleq & [\mathit{center}\colon \mathit{Point}, \mathit{radius}\colon \mathit{Real}] \\ \mathit{ColorCircle} & \triangleq & [\mathit{center}\colon \mathit{ColorPoint}, \mathit{radius}\colon \mathit{Real}] \end{array}$$

Unfortunately, $\mathit{ColorCircle} \not<: \mathit{Circle}$, because of invariance. Now redefine:

$$\begin{array}{lll} \mathit{Circle} & \triangleq & \exists(X<:\mathit{Point})\ [\mathit{center}\colon X, \mathit{radius}\colon \mathit{Real}] \\ \mathit{ColorCircle} & \triangleq & \exists(X<:\mathit{ColorPoint})\ [\mathit{center}\colon X, \mathit{radius}\colon \mathit{Real}] \end{array}$$

Thus we gain $\mathit{ColorCircle} <: \mathit{Circle}$. But covariance in object types was supposed to be unsound, so we must have lost something.

We have lost the ability (roughly) to update the *center* component, since X is unknown. Therefore covariant components are (roughly) read-only components.

The *center* component can still be extracted out of the abstraction, by subsumption from X to *ColorPoint*.

Contravariant Components

There are techniques to obtain contravariant (write-only) components; but these are more complex. (A write-only component can still be read by its sibling methods.) Here is an overview.

$A \triangleq [l{:}B, ...]$ which we want contravariant in B

is transformed into:

$A' \triangleq ...\ [l_{upd}{:}Y, l{:}B, ...]$ where $Y{<:}(A'{\rightarrow}B){\rightarrow}A'$ and l_{upd} updates l

A' is still invariant in B, but any element of A' can be subsumed into:

$A'' \triangleq ...\ [l_{upd}{:}Y, ...]$ contravariant in B, with $A'{<:}A''$

The appropriate definitions are:

$$A' \triangleq \mu(X)\,\exists(Y{<:}(X{\rightarrow}B){\rightarrow}X)\,[l_{upd}{:}Y, l{:}B, ...]$$
$$A'' \triangleq \mu(X)\,\exists(Y{<:}(X{\rightarrow}B){\rightarrow}X)\,[l_{upd}{:}Y, ...]$$

Then $o.l{\Leftarrow}\varsigma(s{:}A)b$ is simulated by a definable update(o', l_{upd}, $\lambda(s{:}A'')b''$) (*i.e.*, roughly, $o.l_{upd}(\lambda(s{:}A)b)$) for appropriate transformations of $o{:}A$ into $o'{:}A'$ and b into b''.

Variant Product and Function Types

Encodings based on object types alone may be undesirably invariant. Quantifiers can introduce the necessary degree of variance.

Variant product types can be define as:

$$A \times^{\exists\exists} B \quad \triangleq \quad \exists(X{<:}A)\ \exists(Y{<:}B)\ [\mathit{fst}{:}X,\ \mathit{snd}{:}Y]$$

With the property:

$$A \times^{\exists\exists} B <: A' \times^{\exists\exists} B' \qquad \text{if} \quad A <: A' \quad \text{and} \quad B <: B'$$

Similarly, but somewhat more surprisingly, we can obtain variant function types:

$$A \rightarrow^{\forall\exists} B \quad \triangleq \quad \forall(X{<:}A)\ \exists(Y{<:}B)\ [\mathit{arg}{:}X,\ \mathit{val}{:}Y]$$

With the property:

$$A \rightarrow^{\forall\exists} B <: A' \rightarrow^{\forall\exists} B' \qquad \text{if} \quad A' <: A \quad \text{and} \quad B <: B'$$

Translation of the first-order λ-calculus with subtyping:

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

$$\langle\!\langle x_A\rangle\!\rangle_\rho \quad\triangleq\quad \rho(x)$$

$$\langle\!\langle b_{A\to B}(a_A)\rangle\!\rangle_\rho \quad\triangleq$$

$$open\ \langle\!\langle b\rangle\!\rangle_\rho(\langle\!\langle A\rangle\!\rangle)\ as\ Y<:\langle\!\langle B\rangle\!\rangle,\ y{:}[arg{:}\langle\!\langle A\rangle\!\rangle,\ val{:}Y]$$

$$in\ (y.arg \Leftarrow \varsigma(x{:}[arg{:}\langle\!\langle A\rangle\!\rangle,\ val{:}Y])\ \langle\!\langle a\rangle\!\rangle_\rho).val \qquad\qquad \text{for } Y,y,x \notin FV(\langle\!\langle a\rangle\!\rangle_\rho)$$

$$\langle\!\langle \lambda(x{:}A)b_B\rangle\!\rangle_\rho \quad\triangleq$$

$$\begin{aligned}&\lambda(X<:\langle\!\langle A\rangle\!\rangle)\\ &\quad(pack\ Y<:\langle\!\langle B\rangle\!\rangle=\langle\!\langle B\rangle\!\rangle,\\ &\qquad[arg = \varsigma(x{:}[arg{:}X,\ val{:}\langle\!\langle B\rangle\!\rangle])\ x.arg,\\ &\qquad\ val = \varsigma(x{:}[arg{:}X,\ val{:}\langle\!\langle B\rangle\!\rangle])\ \langle\!\langle b\rangle\!\rangle_{\rho\{x\leftarrow x.arg\}}]\\ &\quad :[arg{:}X,\ val{:}Y])\end{aligned}$$

Self Types

Recall that $\mu(X)B$ failed to give some expect subtyping behavior. We are now looking for a different quantifier, $\varsigma(X)B$, with the expected behavior.

$P_1 \triangleq \varsigma(Self)[x{:}Int,\ mv\_x{:}Int\to Self]$ movable 1-D points

$P_2 \triangleq \varsigma(Self)[x,y{:}Int,\ mv\_x,mv\_y{:}Int\to Self]$ movable 2-D points

Let $P_1\{X\} \triangleq [x{:}Int,\ mv\_x{:}Int\to X]$ be the ***X*-unfolding** of P_1

with $P_1\{P_1\} \equiv [x{:}Int,\ mv\_x{:}Int\to P_1]$ the **self-unfolding** of P_1.

Some properties we expect for $\varsigma(X)B$, are:

Subtyping. *E.g.*: $P_2 <: P_1$

Creation (folding) *E.g.*: from $P_1\{P_1\}$ to P_1

Selection (unfolding) *E.g.*: $p_1.mv\_x: Int\to P_1$

Update (refolding)

E.g.: from $p_1{:}P_1$ and a "Self-parametric" method such that for all $Y<:P_1$ and $x{:}P_1\{Y\}$ gives $Int\to Y$, produce a new P_1 with an updated mv_x

The $\varsigma(X)B$ Quantifier

It turns out that Self can be formalized via a general quantifier, *i.e.*, independently of object types. Define:

$$\varsigma(X)B \;\triangleq\; \mu(Y)\,\exists(X<:Y)\,B \quad (Y \text{ not occurring in } B)$$

The intuition is the following. Take $A<:A'$ with A¦A':

Want:		$[l{:}A,\ m{:}C]$	$<:$		$[l{:}A']$	(fails)
Do:	$\exists(X<:A)$	$[l{:}X,\ m{:}C]$	$<:$	$\exists(X<:A')$	$[l{:}X]$	(holds)

Want:		$\mu(Y)\,[l{:}Y,\ m{:}C]$	$<:$		$\mu(Y)\ [l{:}Y]$	(fails)
Do:	$\mu(Y)\,\exists(X<:Y)$	$[l{:}X,\ m{:}C]$	$<:$	$\mu(Y)\,\exists(X<:Y)$	$[l{:}X]$	(holds)

This way we can have, *e.g.* $P_2<:P_1$. We achieve subtyping at the cost of making certain fields covariant and, hence, essentially read-only. This suggests, in particular, that we will have difficulties in updating methods that return self.

(Note)

$\varsigma(X)B$ satisfies the subtyping property:

$$E \vdash \varsigma(X)B <: \varsigma(X)B' \quad \text{if} \quad E,X \vdash B <: B'$$

even though we do not have, in general, $\mu(X)B <: \mu(X)B'$.

$E,X \vdash B <: B'$

$\Rightarrow$ $E,Z,Y<:Z,X<:Y \vdash B <: B'$ by weakening , for fresh Y,Z

$\Rightarrow$ $E,Z,Y<:Z \vdash \exists(X<:Y)B <: \exists(X<:Z)B'$ by (Sub Exists)

$\Rightarrow$ $E \vdash \mu(Y)\exists(X<:Y)B <: \mu(Z)\exists(X<:Z)B'$ by (Sub Rec)

Building Elements of Type $\varsigma(X)B$

Modulo an unfolding, $\varsigma(X)B \equiv \mu(Y)\exists(X{<:}Y)B$ (for Y not in B) is the same as:

$$\exists(X{<:}\varsigma(X)B)B.$$

An element of $\exists(X{<:}\varsigma(X)B)B$ is a pair $\langle C, c\rangle$ consisting of a subtype C of $\varsigma(X)B\{X\}$ and an element c of $B\{C\}$.

We denote by

$$\mathit{wrap}\langle C, c\rangle$$

the injection of the pair $\langle C, c\rangle$ from $\exists(X{<:}\varsigma(X)B)B$ into $\varsigma(X)B$.

For example, suppose we have an element x of type $\varsigma(X)X$. Then, choosing $\varsigma(X)X$ as the required subtype of $\varsigma(X)X$, we obtain $\mathit{wrap}\langle\varsigma(X)X, x\rangle : \varsigma(X)X$. Therefore we can construct:

$$\mu(x)\ \mathit{wrap}\langle\varsigma(X)X, x\rangle\ : \varsigma(X)X$$

The fully explicit version of $\mathit{wrap}\langle C, c\rangle$ is written:

$$\mathit{wrap}(X{<:}\varsigma(X)B{=}C)\ c \qquad \text{(or } \mathit{wrap}(X{=}\varsigma(X)B)\ c \text{ for } C{\equiv}\varsigma(X)B\text{)}$$

and it binds the name X to C in c.

Building a Memory Cell

Suppose we want to build a memory cell $m{:}M$ with a read operation $rd{:}Nat$ and a write operation $wr{:}Nat{\to}M$. We can define:

$$M \quad \triangleq \quad \varsigma(Self)[rd{:}Nat,\ wr{:}Nat{\to}Self]$$

where the *wr* method should use its argument to update the *rd* field. For convenience, we adopt the following abbreviation to unfold a Self quantifier:

$$A\{\!|C|\!\} \quad \triangleq \quad B\{C\} \qquad \text{whenever} \qquad A \equiv \varsigma(X)B\{X\} \ \ \text{and} \ \ C{<:}A$$

For example we have $M\{\!|M|\!\} \equiv [rd{:}Nat,\ wr{:}Nat{\to}M]$.

Then we can define:

$$\begin{aligned} m{:}\ M \quad \triangleq \quad & wrap\langle M, \\ & \quad [rd = 0, \\ & \quad\ \ wr = \varsigma(s{:}M\{\!|M|\!\})\ \lambda(n{:}Nat)\ wrap\langle M,\ s.rd{:=}n\rangle]\rangle \end{aligned}$$

Derived Rules for $\varsigma(X)B$

Formally, we can define an introduction construct ($wrap(Y{<:}A{=}C)b\{Y\}$) and an elimination construct ($use\ c\ as\ Y{<:}A,\ y{:}B\{Y\}\ in\ d{:}D$), for $\varsigma(X)B$, such that:

(Type Self)

$$\frac{E, X{<:}Top \vdash B}{E \vdash \varsigma(X)B}$$

(Sub Self)

$$\frac{E, X{<:}Top \vdash B <: B'}{E \vdash \varsigma(X)B <: \varsigma(X)B'}$$

(Val Wrap) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash C <: A \qquad E \vdash b\{\!\{C\}\!\} : B\{\!\{C\}\!\}}{E \vdash \mathrm{wrap}(Y{<:}A{=}C)b\{Y\} : A}$$

(Val Use) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash c : A \qquad E \vdash D \qquad E, Y{<:}A, y{:}B\{\!\{Y\}\!\} \vdash d : D}{E \vdash use\ c\ as\ Y{<:}A,\ y{:}B\{\!\{Y\}\!\}\ in\ d{:}D : D}$$

(Plus the derived equational theory.)

(Note)

Define, for $A \equiv \varsigma(X)B\{X\}$, $C <: A$, and $b\{C\}:B\{C\}$:

$$wrap(Y <: A = C)\ b\{Y\} \ \triangleq\ fold(A,\ (pack\ Y <: A = C,\ b\{Y\}:B\{Y\}))$$

and, for $c:A$ and $d\{Y,y\}:D$, where Y does not occur in D:

$$(use\ c\ as\ Y <: A,\ y:B\{Y\}\ in\ d\{Y,y\}:D) \ \triangleq$$
$$(open\ unfold(c)\ as\ Y <: A,\ y:B\{Y\}\ in\ d\{Y,y\}:D)$$

The ςOb Calculus

At this point we may extract a **minimal second-order object calculus**. We discard the universal and existential quantifiers, and recursion, and we retain the ς quantifier and the object types:

A,B ::=	a,b ::=
X	x
Top	$[l_i{=}\varsigma(x_i{:}A_i)b_i \;^{i\in 1..n}]$
$[l_i{:}B_i \;^{i\in 1..n}]$	$a.l$
$\varsigma(X)B$	$a.l \Leftarrow \varsigma(x{:}A)b$
	$wrap(X{<:}A{=}B)b$
	$use\ a\ as\ X{<:}A,\ y{:}B\ in\ b{:}D$

ς-Object Types

Now that we have a general formulation of $\varsigma(X)B$, we can go back and consider its application to object types. We consider types of the special form:

$$\varsigma(X^+)[l_i{:}B_i\{X\}^{\,i\in 1..n}] \;\triangleq\; \varsigma(X)[l_i{:}B_i\{X\}^{\,i\in 1..n}] \qquad \text{when the } B_i \text{ are covariant in } X$$

Here, $\varsigma(X^+)[l_i{:}B_i\{X\}^{\,i\in 1..n}]$ are called ς-object types. Our goal is to discover their derived typing rules.

- The covariance requirement is necessary to get selection to work. An example of violation of covariance are “binary methods” such as:

  $$\varsigma(Self)[\ ...,\ eq{:}\ Self \rightarrow Bool,\ ...\ ]$$

  (It turns out that $p.eq$ cannot be given a type, because a contravariant *Self* occurrence is not able to escape the scope of the existential quantifier. A covariant *Self* occurrence can be eliminated by subsumption into the object type.)

- The covariance requirements rules out “nested” *Self* types, because of the invariance of object type components ($\varsigma(Y)\ [l_2{:}\ X]$ is invariant in X):

  $$\varsigma(X)\ [l_1{:}\ \varsigma(Y)\ [l_2{:}\ X]]$$

- These restrictions are common in languages that admit *Self* types.

Derived Rules for ς-Object Types

We have essentially the same rules for subtyping and construction. But now, the generic "use" elimination construct of ς-quantifiers can be specialized to obtain selection and update:

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

$$\frac{E \vdash a : A \qquad j\in 1..n}{E \vdash a_A.l_j : B_j\{\!\{A\}\!\}}$$

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

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

where $wrap(Y{<:}A, x{:}A(Y))b$ is a "Self-parametric" method that must produce for every $Y{<:}A$ and $x{:}A(Y)$ (where x is self) a result of type $B_j\{Y^+\}$, parametrically in Y. In particular, it is unsound for the method to produce a result of type $B_j\{A\}$.

Hence the (already known) notion of Self-parametric methods falls out naturally in this framework, as a condition for a derived rule.

(Note)

Assume $a{:}A$ with $A \equiv \varsigma(X^{+})[l_i{:}B_i\{X\}^{\ i \in 1..n}]$ and $A(X) \equiv [l_i{:}B_i\{X^{+}\}^{\ i \in 1..n}]$, and set, with some overloading of notation:

$a.l_j \ \triangleq$

$\quad (use\ a\ as\ Z{<:}A,\ y{:}A(Z)\ in\ y.l_j\ :\ B_i\{A^{+}\})$

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

$\quad (use\ a\ as\ Z{<:}A,\ y{:}A(Z)\ in\ wrap(Y{<:}A{=}Z)\ (y.l_j \Leftarrow \varsigma(x{:}A(Y))b\{Y,y,x\})\ :\ A)$

The Type of the Object-Oriented Naturals

We can finally give a type for the object-oriented natural numbers:

$$N_{\mathrm{Ob}} \triangleq \varsigma(Self^{+})[succ{:}Self,\ case{:}\forall(Z)Z\to(Self\to Z)\to Z]$$

Note that the covariance restriction is respected.

The zero numeral can then be typed as follows:

$$\begin{aligned}
&zero_{\mathrm{Ob}} : N_{\mathrm{Ob}} \triangleq \\
&\quad wrap(Self{=}N_{\mathrm{Ob}}) \\
&\qquad [\ case = \lambda(Z)\ \lambda(z{:}Z)\ \lambda(f{:}Self\to Z)\ z, \\
&\qquad\ \ succ = \varsigma(n{:}N_{\mathrm{Ob}}\{Self\}) \\
&\qquad\qquad wrap\langle Self,\ n.case := \lambda(Z)\ \lambda(z{:}Z)\ \lambda(f{:}Self\to Z)\ f(wrap\langle Self, n\rangle)\rangle]
\end{aligned}$$

The Type of the Calculator

$$C \triangleq \varsigma(Self^{+})[arg,acc: Real,\ enter: Real \to Self,\ add,sub: Self,\ equals: Real]$$

$$Calc \triangleq \varsigma(Self^{+})[enter: Real \to Self,\ add,sub: Self,\ equals: Real]$$

Then *Calc* <: *C*; we can hide *arg* and *acc* from clients.

$$
\begin{array}{l}
calculator: C \triangleq \\
\quad wrap(Self{=}C) \\
\qquad [arg = 0.0, \\
\qquad\ acc = 0.0, \\
\qquad\ enter = \varsigma(s{:}C\{Self\})\ \lambda(n{:}Real)\ wrap\langle Self,\ s.arg := n\rangle, \\
\qquad\ add = \varsigma(s{:}C\{Self\}) \\
\qquad\quad wrap\langle Self,\ (s.acc := s.equals).equals \Leftarrow \varsigma(s'{:}C\{Self\})\ s'.acc{+}s'.arg\rangle, \\
\qquad\ sub = \varsigma(s{:}C\{Self\}) \\
\qquad\quad wrap\langle Self,\ (s.acc := s.equals).equals \Leftarrow \varsigma(s'{:}C\{Self\})\ s'.acc{-}s'.arg\rangle, \\
\qquad\ equals = \varsigma(s{:}C\{Self\})\ s.arg\ ]
\end{array}
$$

Overriding and Self

If we want to update a method of a ς-object $o{:}A$, the new method must work for any possible $Self{<:}A$, because o might have been initially built as an element of an unknown $B{<:}A$.

This is a tough requirement if the method result involves the Self type, since we do not know the “true Self” of o.

(We have no such problem at object creation time, since the “true Self” is known then. But the same difficulty would likely surface if we were creating objects incrementally, adding one method at a time to extensible objects.)

Consider, for example, the type:

$$A \triangleq \varsigma(Self^{+})[n{:}Int,\ m{:}Self] \qquad \text{with} \quad A\{\!|Self|\!\} \equiv [n{:}Int,\ m{:}Self]$$

According to the rule (Val ςUpdate), an updating method can use in its body the variables $Self{<:}A$, and $x{:}A\{\!|Self|\!\}$, where x is the self of the new method.

Basically, for a method l with result type $B_l\{Self\}$, the update rule requires that we construct a polymorphic function of type:

$$\forall(Self{<:}A)\ A\{\!|Self|\!\}{\to}B_l\{Self\}$$

For n, we have no problem in returning a $B_n\{Self\} \equiv Int$.

But for m, there is no obvious way of producing a $B_m\{Self\} \equiv Self$ from $x{:}A\{\!|Self|\!\}$, except for $x.m$ which loops. And we cannot construct an element of an arbitrary $Self{<:}A$.

Moreover, using $\forall(Self{<:}A)\ A\{\!|Self|\!\}{\to}B\{A\}$, for example, would be unsound.

In conclusion, the (Val ςUpdate) rule, although sufficient for updating simple methods and fields, is not sufficient to allow us to *usefully* update methods that return a value of type *Self*, *after object construction*.

Recoup

We introduce a special method called *recoup* with an associated run-time invariant. Recoup is a method that returns self immediately. The invariant asserts that the result of recoup is its host object. These simple assumptions have surprising consequences.

$$A \quad \triangleq \quad \varsigma(Self^{+})[r{:}Self,\ n{:}Int,\ m{:}Self] \qquad \text{with } A\{Self\} \equiv [r{:}Self,\ n{:}Int,\ m{:}Self]$$

$$a : A \quad \triangleq \quad wrap(Self{<:}A{=}A)\ [r = \varsigma(x{:}A\{Self\})wrap\langle Self,x\rangle,\ \ldots\ ] : A$$

Then, the following update on m typechecks, since $x.r$ has type $Self$:

$$a.m \Leftarrow \varsigma(Self{<:}A,\ x{:}A\{Self\})\ (x.n{:=}3).r{:} \quad A$$

The reduction behavior of this term relies on the recoup invariant. *I.e.*, recoup should be correctly initialized and not subsequently corrupted.

Intuitively, recoup allows us to recover a “parametric self ” $x.r$ which equals the object a but has type $Self{<:}A$ (the “true Self”) and not just type A (the “known Self”).

In general, if A has the form $\varsigma(Self^{+})[r{:}Self, ...]$ then we can write *useful* polymorphic functions of type:

$$\forall(Self{<:}A)\ A\{Self\}\rightarrow Self$$

that are not available without recoup. Such functions are parametric enough to be useful for method update.

In a programming language based on these notions, recoup could be introduced as a "built-in feature", so that the recoup invariant is guaranteed for all objects at all times.

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 $X \neq Y$
$Top\{X^+\}$	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 $X \neq Y$
$Top\{X^-\}$	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

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

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

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

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

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

Cells

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

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

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

$$GCell <: Cell$$

Cells with Undo

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

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

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

$$\begin{aligned}
& uncell : UnCell \triangleq \\
& \quad obj(X{=}UnCell) \\
& \quad [contents = \varsigma(x{:}X)\ 0, \\
& \quad\ set = \varsigma(x{:}X)\ \lambda(n{:}Nat) \\
& \qquad\qquad (x.undo \Leftarrow (Y{<:}X, y{:}Y)\ \varsigma(z{:}Y)\ y) \\
& \qquad\qquad\quad .contents \Leftarrow (Y{<:}X, y{:}Y)\ \varsigma(z{:}Y)\ n, \\
& \quad\ undo = \varsigma(x{:}X)\ x]
\end{aligned}$$

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

Operational Semantics

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

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

$$\frac{}{\vdash v \leadsto v}$$

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

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

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

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

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 $\emptyset$)

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

(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 X<:)

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

(Type Top)

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

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

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

(Sub Refl)

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

(Sub Trans)

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

(Sub Top)

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

(Sub X)

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

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

$$\frac{E \vdash A \qquad E \vdash A' \qquad E, Y{<:}A \vdash \upsilon_i\ B_i\{\!\{Y\}\!\} <: \upsilon_i'\ B_i'\{\!\{Y\}\!\} \qquad \forall i \in 1..n}{E \vdash A <: A'}$$

(Sub Invariant)

$$\frac{E \vdash B}{E \vdash {}^{o}B <: {}^{o}B}$$

(Sub Covariant)

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

(Sub Contravariant)

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

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

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

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

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

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
$\ldots$	(as before)
$\forall(X<:A)B$	bounded universal type
$a,b ::=$	terms
$\ldots$	(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 $\varnothing \vdash a : A$ and $\vdash a \leadsto v$, then $\varnothing \vdash v : A$.

Classes and Self

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

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

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

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

Now pre-methods have polymorphic types.

For example:

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

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

Inheritance and Self

We can now reconsider the inheritance relation between classes.

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

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

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*):

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

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

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:

$$CellEq \triangleq \mu(X)[contents : Nat, set : Nat \to X, eq : X \to Bool]$$

$$CellSEq \triangleq \mu(X)[contents : Nat, set : Nat \to X, sign : Bool, eq : X \to Bool]$$

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 $eq = \varsigma(x)\lambda(y)\ x.contents = y.contents$.

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:

$$\mathit{CellSEqOp}(X) <: \mathit{CellEqOp}(X)$$

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

$$\mathit{CellSEqOp} <: \mathit{CellEqOp} :: \quad \mathit{Ty} \Rightarrow \mathit{Ty}$$

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

$$\mathit{CellEq} \triangleq$$
$$\quad \mu(X)\mathit{CellEqOp}(X)$$
$$\mathit{CellSEq} \triangleq$$
$$\quad \mu(X)\mathit{CellSEqOp}(X)$$

- 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:

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

- 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.)

ENCODING OBJECT CALCULI

Objects vs. Procedures

- Object-oriented programming languages have introduced (or popularized) a number of ideas and techniques.
- In order to avoid premature commitments, so far we have avoided any explicit encoding of objects in terms of other notions.
- 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 subtyping.

  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

- It is natural to try to program objects from records and functions. The self-application semantics is one of the more natural ways of doing this.
- All implementations of standard (single-dispatch) object-oriented languages are based on self-application. In the self-application semantics,
  - ~ methods are functions,
  - ~ objects are records,
  - ~ update is simply record update.
  - ~ On method invocation, the whole object is passed to the method as a parameter.

Untyped self-application interpretation

$[l_i{=}\varsigma(x_i)b_i \;^{i\in 1..n}]$	$\triangleq \;\; \langle l_i{=}\lambda(x_i)b_i \;^{i\in 1..n}\rangle$	(l_i distinct)
$o.l_j$	$\triangleq \;\; o{\cdot}l_j(o)$	$(j\in 1..n)$
$o.l_j{\Leftarrow}\varsigma(y)b$	$\triangleq \;\; o{\cdot}l_j{:=}\lambda(y)b$	$(j\in 1..n)$

The Self-Application Semantics (Typed)

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

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

Self-application interpretation

$A \equiv [l_i{:}B_i \ ^{i\in 1..n}] \quad \triangleq$ $\quad \mu(X)\langle l_i{:}X\rightarrow B_i \ ^{i\in 1..n}\rangle$	(l_i distinct)
$[l_i{=}\varsigma(x_i{:}A)b_i \ ^{i\in 1..n}] \quad \triangleq \quad fold(A,\langle l_i{=}\lambda(x_i{:}A)b_i \ ^{i\in 1..n}\rangle)$	
$o.l_j \quad \triangleq \quad unfold(o)\cdot l_j(o)$	$(j\in 1..n)$
$o.l_j \Leftarrow\varsigma(y{:}A)b \quad \triangleq \quad fold(A,unfold(o)\cdot l_j{:=}\lambda(y{:}A)b)$	$(j\in 1..n)$

- 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 $b_i{'}$)

$o{\scriptstyle\square}f_j \ \triangleq \ o{\cdot}st{\cdot}f_j$ ($j\in 1..m$) (external)

$o{\scriptstyle\square}f_j{:=}b \ \triangleq \ \langle st{=}(o{\cdot}st{\cdot}f_j{:=}b), mt{=}o{\cdot}mt\rangle$ ($j\in 1..m$) (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 $\mu(m)\langle l_i{=}\lambda(s)\ \ldots\ \rangle$

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

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

- 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

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

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

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

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

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

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

$\qquad : C\{X\}$

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

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

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

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

$$CPoint \;\triangleq$$
$$\mu(X)\langle x{:}Int,\ c{:}Color,\ mv{:}Int\rightarrow X\rangle$$

$$cPoint : CPoint \;\triangleq$$
$$\begin{aligned}
&let\ rec\ init(x0{:}Int,\ c0{:}Color) = \\
&\quad \mu(s{:}CPoint)\ fold(CPoint, \\
&\qquad \langle x = x0,\ c = c0, \\
&\qquad\ mv = \lambda(dx{:}Int)\ init(unfold(s)\cdot x{+}dx,\ unfold(s)\cdot c)\rangle) \\
&in\ init(0,\ black)
\end{aligned}$$

- 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 $\lambda(dx{:}Int)\ f(s).x{:=}s.x{+}dx$, where f is of appropriate type, it would not have been clear how to proceed.

The Split-Methods Semantics

Untyped split-method interpretation

$$[l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}] \ \triangleq \qquad (l_i \text{ distinct})$$
$$\quad \mathit{let\ rec\ create}(y_i\ ^{i\in 1..n}) =$$
$$\qquad \langle l_i^{sel}{=}y_i,$$
$$\qquad\ l_i^{upd}{=}\lambda(y_i')\ \mathit{create}(y_j\ ^{j\in 1..i-1}, y_i', y_k\ ^{k\in i+1..n})\ ^{i\in 1..n}\rangle$$
$$\quad \mathit{in\ create}(\lambda(x_i)b_i\ ^{i\in 1..n})$$

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

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

- 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

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

$\quad \mu(Y)\exists(X<:Y)C\{X\}$

$\qquad$ where

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

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

$\quad \mathit{let\ rec\ create}(y_i{:}A\to B_i \;^{i\in 1..n}){:}A =$

$\qquad \mathit{fold}(A,$

$\qquad\quad \mathit{pack}\ X{=}A$

$\qquad\quad \mathit{with}$

$\qquad\qquad \langle r{=}\mathit{create}(y_i \;^{i\in 1..n}),$

$\qquad\qquad\ l_i^{sel}{=}y_i \;^{i\in 1..n},$

$\qquad\qquad\ l_i^{upd}{=}\lambda(y_i'{:}A\to B_i)\mathit{create}(y_j \;^{j\in 1..i-1}, y_i', y_k \;^{k\in i+1..n}) \;^{i\in 1..n}\rangle$

$\qquad\quad : C\{X\})$

$\quad \mathit{in\ create}(\lambda(x_i{:}A)b_i \;^{i\in 1..n})$

$$o_A.l_j \quad \triangleq \qquad\qquad (j \in 1..n)$$
$$open\ unfold(o)\ as\ X<:A,\ p:C\{X\}$$
$$in\ p\cdot l_j^{sel}(p\cdot r) : B_j$$

$$o.l_j \Leftarrow \varsigma(y:A)b \quad \triangleq \qquad\qquad (j \in 1..n)$$
$$open\ unfold(o)\ as\ X<:A,\ p:C\{X\}$$
$$in\ p\cdot l_j^{upd}(\lambda(y:A)b) : A$$

- 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.

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

Imperative self-application interpretation

$A \equiv [f_k{:}B_k \; ^{k\in 1..m} \mid l_i{:}B_i \; ^{i\in 1..n}] \quad \triangleq$ $\qquad$ (f_k, l_i distinct)

$\quad \mu(Y)\exists(X<:Y)C\{X\}$

$\qquad$ with

$\quad C\{X\} \equiv \langle r{:}X, f_k{:}B_k \; ^{k\in 1..m}, l_i{:}X\to B_i \; ^{i\in 1..n}, cl{:}\langle\rangle\to X\rangle$

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

$\quad let\ rec\ create(y_k{:}B_k \; ^{k\in 1..m}, y_i{:}A\to B_i \; ^{i\in 1..n}){:}A =$

$\qquad let\ z{:}C\{\!\{A\}\!\} = \langle r{=}nil(A), f_k{=}y_k \; ^{k\in 1..m}, l_i{=}y_i \; ^{i\in 1..n}, cl{=}nil(\langle\rangle\to A)\rangle$

$\qquad in \quad z{\cdot}r{:=}fold(A, pack\ X<:A{=}A\ with\ z : C\{\!\{X\}\!\});$

$\qquad\qquad z{\cdot}cl{:=}\lambda(x{:}\langle\rangle)create(z{\cdot}f_k \; ^{k\in 1..m}, z{\cdot}l_i \; ^{i\in 1..n});$

$\qquad\qquad z{\cdot}r$

$\quad in\ create(b_k \; ^{k\in 1..m}, \lambda(x_i{:}A)b_i \; ^{i\in 1..n})$

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

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

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

$\quad in\ fold(A, pack\ X'<:X{=}X\ with\ p{\cdot}f_j{:=}b : C\{\!\{X'\}\!\}) : A$

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

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

$\quad in\ p{\cdot}l_j(p{\cdot}r) : B_j$

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

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

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

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

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

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

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

Summary

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.

Matching

- The subtyping relation between object types is the foundation of subclassing and inheritance . . . when it holds.
- Subtyping fails to hold between certain types that arise naturally in object-oriented programming. Typically, recursively defined object types with binary methods.

- F-bounded subtyping was invented to solve this kind of problem.
- A new programming construction, called "matching" has been proposed to solve the same problem, inspired by F-bounded subtyping.

- Matching achieves "covariant subtyping" for Self types. Contravariant subtyping still applies, otherwise.

- We argue that matching is a good idea, but that it should not be based on F-bounded subtyping. We show that a new interpretation of matching, based on higher-order subtyping, has better properties.

When Subtyping Works

- A simple treatment of objects, classes, and inheritance is possible for covariant Self types (only).

Object Types

- Consider two types Inc and IncDec containing an integer field and some methods:

$$\mathrm{Inc} \triangleq \mu(X)[n{:}\mathrm{Int},\ inc^{+}{:}X]$$
$$\mathrm{IncDec} \triangleq \mu(Y)[n{:}\mathrm{Int},\ inc^{+}{:}Y,\ dec^{+}{:}Y]$$

- A typical object of type Inc is:

$$p : \mathrm{Inc} \triangleq$$
$$[n = 0,$$
$$inc = \varsigma(self{:}\ \mathrm{Inc})\ self.n := self.n + 1]$$

Subtyping

- Subtyping (<:) is a reflexive and transitive relation on types, with *subsumption*:

  if $a : A$ and $A <: B$ then $a : B$

- For object types, we have the subtyping rule:

  $[v_i{:}B_i\ ^{i\in I}, m_j^{+}{:}C_j\ ^{j\in J}] \;<:\; [v_i{:}B_i\ ^{i\in I'}, m_j^{+}{:}C_j'\ ^{j\in J'}]$

  if $C_j <: C_j'$ for all $j\in J'$, with $I'\subseteq I$ and $J'\subseteq J$

- For recursive types we have the subtyping rule:

  $\mu(X)A\{X\} <: \mu(Y)B\{Y\}$

  if $X <: Y$ implies $A\{X\} <: B\{Y\}$

- Combining them, we obtain a derived rule for recursive object types:

  $\mu(X)[v_i{:}B_i\ ^{i\in I}, m_j^{+}{:}C_j\{X\}\ ^{j\in J}] \;<:\; \mu(Y)[v_i{:}B_i\ ^{i\in I'}, m_j^{+}{:}C_j'\{Y\}\ ^{j\in J'}]$

  if $X <: Y$ implies $C_j\{X\} <: C_j'\{Y\}$ for all $j\in J'$, with $I'\subseteq I$ and $J'\subseteq J$

  E.g.: IncDec <: Inc

Pre-Methods

- The subtyping relation (e.g. IncDec <: Inc) plays an important role in inheritance.
- Inheritance is obtained by reusing polymorphic code fragments.

$$\text{pre-inc} : \forall(X{<:}\text{Inc})X\to X \quad \triangleq$$
$$\lambda(X{<:}\text{Inc})\ \lambda(\text{self}:X)\ \text{self.n} := \text{self.n}+1 \qquad \text{(using a “structural” rule)}$$

- We call a code fragment such as pre-inc a *pre-method*.
- N.B. it is not enough to have pre-inc : Inc→Inc if we want to inherit this pre-method in IncDec classes. Here polymorphism is essential.
- N.B. the body of pre-inc is typed by means of a *structural* rule for update, which is essential in many examples involving bounded quantification.
- We can specialize pre-inc to implement the method inc of type Inc or IncDec:

$$\text{pre-inc(Inc)} : \text{Inc}\to\text{Inc}$$
$$\text{pre-inc(IncDec)} : \text{IncDec}\to\text{IncDec}$$

- Thus, we have reused pre-inc at different types, without retypechecking its code.

Classes

- Pre-method reuse can be systematized by collecting pre-methods into *classes*.
- A class for an object type A can be described as a collection of pre-methods and initial field values, plus a way of generating new objects of type A.
- In a class for an object type A, the pre-methods are parameterized over all subtypes of A, so that they can be reused (inherited) by any class for any subtype of A.
- Let A be a type of the form $\mu(X)[v_i{:}B_i\ ^{i\in I}, m_j^+{:}C_j\{X\}\ ^{j\in J}]$. As part of a class for A, a pre-method for m_j would have the type $\forall(X<:A)X\rightarrow C_j\{X\}$. For example:

$$
\begin{aligned}
&\mathrm{IncClass} \triangleq \\
&\quad [\mathrm{new}^+{:}\ \mathrm{Inc}, \\
&\quad\ \mathrm{n}{:}\ \mathrm{Int}, \\
&\quad\ \mathrm{inc}{:}\ \forall(X<:\mathrm{Inc})X\rightarrow X]
\end{aligned}
$$

N.B.: $\mathrm{inc}{:}\ \mathrm{Inc}\rightarrow\mathrm{Inc}$
would not allow inheritance

$$
\begin{aligned}
&\mathrm{IncDecClass} \triangleq \\
&\quad [\mathrm{new}^+{:}\ \mathrm{IncDec}, \\
&\quad\ \mathrm{n}{:}\ \mathrm{Int}, \\
&\quad\ \mathrm{inc}{:}\ \forall(X<:\mathrm{IncDec})X\rightarrow X, \\
&\quad\ \mathrm{dec}{:}\ \forall(X<:\mathrm{IncDec})X\rightarrow X]
\end{aligned}
$$

- A typical class of type IncClass reads:

$$
\begin{array}{l}
\mathrm{incClass} : \mathrm{IncClass} \quad \triangleq \\
\quad [\mathrm{new} = \varsigma(\mathrm{classSelf}: \mathrm{IncClass}) \\
\qquad\quad [n = \mathrm{classSelf}.n,\ \mathrm{inc} = \varsigma(\mathrm{self}{:}\mathrm{Inc})\ \mathrm{classSelf}.\mathrm{inc}(\mathrm{Inc})(\mathrm{self})] \\
\quad\ n = 0, \\
\quad\ \mathrm{inc} = \text{pre-inc}]
\end{array}
$$

The code for new is uniform: it assembles all the pre-methods into a new object.

Inheritance

- Inheritance is obtained by extracting a pre-method from a class and reusing it for constructing another class.

  For example, the pre-method pre-inc of type $\forall(X<:Inc)X\to X$ in a class for Inc could be re-used as a pre-method of type $\forall(X<:IncDec)X\to X$ in a class for IncDec:

  incDecClass : IncDecClass $\triangleq$
  [new = ς(classSelf: IncDecClass)[...],
  n = 0,
  inc = incClass.inc,
  dec = ...]

- This example of inheritance requires the subtyping:

  $$\forall(X<:Inc)X\to X \quad <: \quad \forall(X<:IncDec)X\to X$$

  which follows from the subtyping rules for quantified types and function types:

  $\forall(X<:A)B <: \forall(X<:A')B'$ if $A'<:A$ and if $X<:A$ implies $B<:B'$
  $A\to B <: A'\to B'$ if $A' <: A$ and $B <: B'$

Inheritance from Subtyping

- In summary, inheritance from a class for Inc to a class for IncDec is enabled by the subtyping IncDec <: Inc.
- Unfortunately, inheritance is possible and desirable even in situations where such subtypings do not exist. These situations arise with binary methods.

Binary Methods

- Consider a recursive object type Max, with a field n and a binary method max.

  $$\mathrm{Max} \triangleq \mu(X)[n{:}\mathrm{Int},\ max^{+}{:}X\rightarrow X]$$

  Consider also a type MinMax with an additional binary method min:

  $$\mathrm{MinMax} \triangleq \mu(Y)[n{:}\mathrm{Int},\ max^{+}{:}Y\rightarrow Y,\ min^{+}{:}Y\rightarrow Y]$$

- Problem:

  $$\mathrm{MinMax} \not<: \mathrm{Max}$$

  according to the rules we have adopted, since :

  $$Y <: X \quad \not\Rightarrow \quad Y\rightarrow Y <: X\rightarrow X \qquad \text{for } max^{+}$$

  Moreover, it would be unsound to assume MinMax <: Max.

- Hence, the development of classes and inheritance developed for Inc and IncDec falters in presence of binary methods.

Looking for a New Relation

- If subtyping doesn't work, maybe some other relation between types will.
- A possible replacement for subtyping: *matching*.

Matching

- Recently, Bruce *et al.* proposed axiomatizing a relation between recursive object types, called *matching*.
- We write A <# B to mean that A matches B; that is, that A is an “extended version” of B. We expect to have, for example:

  IncDec <# Inc
  MinMax <# Max

- In particular, we may write X <# A, where X is a variable. We may then quantify over all types that match a given one, as follows:

  $\forall(X{<}\#A)B\{X\}$

  We call $\forall(X{<}\#A)B$ *match-bounded quantification*, and say that occurrences of X in B are *match-bound*.

- For recursive object types we have:

$$\mu(X)[v_i{:}B_i{}^{\,i\in I}, m_j{}^{+}{:}C_j\{X\}^{\,j\in J}] \;\;<\# \;\; \mu(X)[v_i{:}B_i{}^{\,i\in I'}, m_j{}^{+}{:}C_j\{X\}^{\,j\in J'}]$$
$$\text{if} \quad I' \subseteq I \quad \text{and} \quad J' \subseteq J$$

- Using match-bounded quantification, we can rewrite the polymorphic function pre-inc in terms of matching rather than subtyping:

  pre-inc : $\forall(X<\#Inc)X\rightarrow X \quad \triangleq$

  $\lambda(X<\#Inc)\ \lambda(self{:}X)\ self.n := self.n+1$

  pre-inc(IncDec) : IncDec$\rightarrow$IncDec

- Similarly, we can write a polymorphic version of the function pre-max:

  pre-max : $\forall(X<\#Max)X\rightarrow X\rightarrow X \quad \triangleq$

  $\lambda(X<\#Max)\ \lambda(self{:}X)\ \lambda(other{:}X)$

  if self.n>other.n then self else other

  pre-max(MinMax) : MinMax$\rightarrow$MinMax$\rightarrow$MinMax (didn't hold with <:)

- Thus, the use of match-bounded quantification enables us to express the polymorphism of both pre-max and pre-inc: contravariant and covariant occurrences of Self are treated uniformly.

Matching and Subsumption

- A subsumption-like property does not hold for matching; A <# B is not quite as good as A <: B. (Fortunately, subsumption was not needed in the examples above.)

$$a : A \quad \text{and} \quad A <\# B \quad \text{need not imply} \quad a : B$$

- Thus, matching cannot completely replace subtyping. For example, forget that IncDec <: Inc and try to get by with IncDec <# Inc. We could not typecheck:

$$\begin{aligned}&inc : Inc \to Inc \;\triangleq\\&\qquad \lambda(x{:}Inc)\; x.n := x.n+1\\&\lambda(x{:}IncDec)\; inc(x)\end{aligned}$$

  We can circumvent this difficulty by turning inc into a polymorphic function of type $\forall(X<\#Inc)X\to X$, but this solution requires foresight, and is cumbersome:

$$\begin{aligned}&pre\text{-}inc : \forall(X<\#Inc)X\to X \;\triangleq\\&\qquad \lambda(X<\#Inc)\;\lambda(x{:}X)\; x.n := x.n+1\\&\lambda(x{:}IncDec)\; pre\text{-}inc(IncDec)(x)\end{aligned}$$

Matching and Classes

- We can now revise our treatment of classes, adapting it for matching.

$$\begin{array}{l} \mathrm{MaxClass} \triangleq \\ \quad [\mathrm{new}^{+}: \mathrm{Max}, \\ \quad\ \mathrm{n}: \mathrm{Int}, \\ \quad\ \mathrm{max}: \forall(X <\# \mathrm{Max}) X \to X \to X] \end{array}$$

$$\begin{array}{l} \mathrm{MinMaxClass} \triangleq \\ \quad [\mathrm{new}^{+}: \mathrm{MinMax}, \\ \quad\ \mathrm{n}: \mathrm{Int}, \\ \quad\ \mathrm{max}: \forall(X <\# \mathrm{MinMax}) X \to X \to X, \\ \quad\ \mathrm{min}: \forall(X <\# \mathrm{MinMax}) X \to X \to X] \end{array}$$

- A typical class of type MaxClass reads:

$$\begin{array}{l} \mathrm{maxClass} : \mathrm{MaxClass} \triangleq \\ \quad [\mathrm{new} = \varsigma(\mathrm{classSelf}: \mathrm{MaxClass}) \\ \qquad\qquad [\mathrm{n} = \mathrm{classSelf.n},\ \ \mathrm{max} = \varsigma(\mathrm{self}{:}\mathrm{Max})\ \mathrm{classSelf.max}(\mathrm{Max})(\mathrm{self})], \\ \quad\ \mathrm{n} = 0, \\ \quad\ \mathrm{max} = \text{pre-max}] \end{array}$$

Matching and Inheritance

- A typical (sub)class of type MinMaxClass reads:

$$
\begin{aligned}
&\mathrm{minMaxClass} : \mathrm{MinMaxClass} \quad \triangleq \\
&\qquad [\mathrm{new} = \varsigma(\mathrm{classSelf}: \mathrm{MinMaxClass})[...], \\
&\qquad\ \ n = 0, \\
&\qquad\ \ \mathrm{max} = \mathrm{maxClass.max}, \\
&\qquad\ \ \mathrm{min} = ...]
\end{aligned}
$$

- The implementation of max is taken from maxClass, that is, it is inherited. The inheritance typechecks assuming that

$$\forall(X{<}\#\mathrm{Max})X{\to}X{\to}X \quad <: \quad \forall(X{<}\#\mathrm{MinMax})X{\to}X{\to}X$$

- Thus, we are still using some subtyping and subsumption as a basis for inheritance.

Advantages of Matching

Matching is attractive

- The fact that MinMax matches Max is reasonably intuitive.
- Matching handles contravariant Self and inheritance of binary methods.
- Matching is meant to be directly axiomatized as a relation between types.The typing rules of a programming language that includes matching can be explained directly.
- Matching is simple from the programmer's point of view, in comparison with more elaborate type-theoretic mechanisms that could be used in its place.

However...

- The notion of matching is ad hoc (e.g., is defined only for object types).
- We still have to figure out the exact typing rules and properties matching.
- The rules for matching vary in subtle but fundamental ways in different languages.
- What principles will allow us to derive the "right" rules for matching?

Applications

- A language based on matching should be given a set of type rules based on the source type system.
- The rules can be proven sound by a judgment-preserving translation into an object-calculus with higher-order subtyping.

Matching as Higher-Order Subtyping

Higher-Order Subtyping

- Subtyping can be extended to operators, in a pointwise manner:

$F \prec: G \qquad$ if, for all X, $\qquad F(X) <: G(X)$

- The property:

$A_{Op} \prec: B_{Op} \qquad\qquad (A_{Op}$ is a suboperator of $B_{Op})$

is seen as a statement that A extends B.

$MinMax_{Op} \;\equiv\; \lambda(X)\, [n{:}Int,\, max^{+}{:}X\to X,\, min^{+}{:}\, X\to X]$
$\prec: \quad \lambda(X)\, [n{:}Int,\, max^{+}{:}X\to X,\, min^{+}{:}\, X\to X] \;\equiv\; Max_{Op}$

- We obtain:

$$
\begin{array}{ll}
Max_{Op} \prec: Max_{Op} & \\
MinMax_{Op} & (\forall X.\quad [n:Int,\ max^{+}:X\to X,\ min^{+}: X\to X] \\
\quad \prec: Max_{Op} & \quad <:\ [n:Int,\ max^{+}:X\to X])
\end{array}
$$

We can parameterize over all type operators X with the property that $X \prec: Max_{Op}$.

$$\forall(X\prec:Max_{Op})B\{X\}$$

We need to be careful about how X is used in B{X}, because X is now a type operator. The idea is to take the fixpoint of X wherever necessary.

$$
\begin{array}{l}
pre\text{-}max : \forall(X\prec:Max_{Op})X^{*}\to X^{*}\to X^{*} \quad \triangleq \\
\quad \lambda(X\prec:Max_{Op})\ \lambda(self:X^{*})\ \lambda(other:X^{*}) \\
\quad\quad \text{if } self.n>other.n \text{ then } self \text{ else } other
\end{array}
$$

$$pre\text{-}max(MinMax_{Op}) : MinMax\to MinMax\to MinMax$$

This typechecks, e.g.:

$$X = X(X^*)$$

$$X \prec: Max_{Op} \Rightarrow X(X^*) <: Max_{Op}(X^*)$$

$$self : X^* \Rightarrow self : X(X^*) \Rightarrow self : Max_{Op}(X^*) \Rightarrow self.n : Int$$

(In this derivation we have used the unfolding property X*=X(X*), but we can do without it by introducing explicit fold/unfold terms.)

The Higher-Order Interpretation

- The central idea of the interpretation is:

$$A <\# B \quad \approx \quad A_{Op} \prec: B_{Op}$$

$$\forall(X<\#A)B\{X\} \quad \approx \quad \forall(X\prec:A_{Op})B\{X^*\} \qquad \text{(not quite)}$$

  We must be more careful about the B{X*} part, because X may occur both in type and operator contexts.

- We handle this problem by two translations for the two kinds of contexts:

$$A <\# B \quad \approx \quad \mathit{Oper}\langle A\rangle \prec: \mathit{Oper}\langle B\rangle$$

$$\forall(X<\#A)B \quad \approx \quad \forall(X\prec:\mathit{Oper}\langle A\rangle)\mathit{Type}\langle B\rangle$$

- The two translations, *Type*$\langle A\rangle$ and *Oper*$\langle A\rangle$, can be summarized as follows.

  For object types of the source language, we set:

$$\mathit{Oper}\langle X\rangle \approx \qquad \text{(assuming that X is match-bound)}$$
$$\quad X$$

$$\mathit{Oper}\langle \mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{X\}\ ^{j\in J}]\rangle \approx$$
$$\quad \lambda(X)[v_i{:}\mathit{Type}\langle B_i\rangle\ ^{i\in I},\ m_j^+{:}\mathit{Type}\langle C_j\{X\}\rangle\ ^{j\in J}]$$

$$\mathit{Type}\langle X\rangle \approx \qquad \text{(when X is match-bound)}$$
$$\quad X^*$$

$$\mathit{Type}\langle \mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{X\}\ ^{j\in J}]\rangle \approx$$
$$\quad \mu(X)[v_i{:}\mathit{Type}\langle B_i\rangle\ ^{i\in I},\ m_j^+{:}\mathit{Type}\langle C_j\{X\}\rangle\ ^{j\in J}]$$

  For other types, we set:

$$\mathit{Type}\langle X\rangle \approx X \qquad \text{(when X is not match-bound)}$$
$$\mathit{Type}\langle A\to B\rangle \approx \mathit{Type}\langle A\rangle\to\mathit{Type}\langle B\rangle$$
$$\mathit{Type}\langle \forall(X<\#A)B\rangle \approx \forall(X<:\mathit{Oper}\langle A\rangle)\mathit{Type}\langle B\rangle$$

- For instance:

$$\mathit{Type}\langle\forall(X{<}\#Max)\ \forall(Y{<}\#X)\ X{\to}Y\rangle \quad\approx$$
$$\forall(X{\prec:}Max_{Op})\ \forall(Y{\prec:}X)\ X^{*}{\to}Y^{*}$$

  This translation is well-defined on type variables, so there are no problems with cascading quantifiers.

- A note about unfolding of recursive types:
  - ~ The higher-order interpretation does not use the unfolding property of recursive types for the *target* language; instead, it uses explicit fold and unfold primitives.
  - ~ On the other hand, the higher-order interpretation is incompatible with the unfolding property of recursive types in the *source* language, because $\mathit{Oper}\langle\mu(X)A\{X\}\rangle$ and $\mathit{Oper}\langle A\{\mu(X)A\{X\}\}\rangle$ are in general different type operators.
  - ~ Technically, the unfolding property of recursive types is not an essential feature and it is the origin of complications; we are fortunate to be able to drop it throughout.

Reflexivity and Transitivity

- Reflexivity is now satisfied by all object types, including variables; for every object type A, we have:

$$A \mathrel{<\#} A \qquad \approx \qquad \mathit{Oper}\langle A\rangle \mathrel{<:} \mathit{Oper}\langle A\rangle$$

  This follows from the reflexivity of $<:$.

- Similarly, transitivity is satisfied by all triples A,B, and C of object types, including variables:

$$\begin{array}{l} A \mathrel{<\#} B \quad \text{and} \quad B \mathrel{<\#} C \quad \text{imply} \quad A \mathrel{<\#} C \quad \approx \\ \qquad \mathit{Oper}\langle A\rangle \mathrel{<:} \mathit{Oper}\langle B\rangle \quad \text{and} \quad \mathit{Oper}\langle B\rangle \mathrel{<:} \mathit{Oper}\langle C\rangle \\ \qquad\qquad \text{imply} \quad \mathit{Oper}\langle A\rangle \mathrel{<:} \mathit{Oper}\langle C\rangle \end{array}$$

  This follows from the transitivity of $<:$.

Matching Self

- With the higher-order interpretation, the relation:

$$A \equiv \mu(Self)[v_i{:}B_i{}^{\,i\in I}, m_j^{+}{:}C_j\{Self\}^{\,j\in J}]$$
$$<\!\#\ \mu(Self)[v_i{:}B_i{}^{\,i\in I}, m_j^{+}{:}C_j'\{Self\}^{\,j\in J'}] \equiv A'$$

holds when the type operators corresponding to A and A' are in the subtyping relation, that is, when:

$$[v_i{:}\mathit{Type}\langle B_i\rangle^{\,i\in I}, m_j^{+}{:}\mathit{Type}\langle C_j\{Self\}\rangle^{\,j\in J}]$$
$$<: [v_i{:}\mathit{Type}\langle B_i\rangle^{\,i\in I}, m_j^{+}{:}\mathit{Type}\langle C_j'\{Self\}\rangle^{\,j\in J'}] \qquad \text{for an arbitrary Self}$$

For this, it suffices that, for every j in J':

$$\mathit{Type}\langle C_j\{Self\}\rangle \ <: \ \mathit{Type}\langle C_j'\{Self\}\rangle$$

Since Self is μ-bound, all the occurrences of Self are translated as Self*. Then, an occurrence of Self* on the left can be matched only by a corresponding occurrence of Self* on the right, since Self is arbitrary. In short,:

Self matches only itself.

.This makes it easy to glance at two object types and tell whether they match

Inheritance and Classes via Higher-Order Subtyping

- Applying our higher-order translation to MaxClass, we obtain:

$$
\begin{aligned}
&\mathrm{MaxClass} \quad \triangleq\\
&\quad [\mathrm{new}^{+}\!: \mathrm{Max},\\
&\quad\ \ \mathrm{n}: \mathrm{Int},\\
&\quad\ \ \mathrm{max}: \forall(X \prec: \mathrm{Max}_{\mathrm{Op}})X^* \to X^* \to X^*]
\end{aligned}
$$

The corresponding translation at the term level produces:

$$
\begin{aligned}
&\mathrm{maxClass} : \mathrm{MaxClass} \quad \triangleq\\
&\quad [\mathrm{new} = \varsigma(\mathrm{classSelf}\colon \mathrm{MaxClass})\\
&\qquad\qquad \mathrm{fold}(\\
&\qquad\qquad\quad [\mathrm{n} = \mathrm{classSelf.n},\\
&\qquad\qquad\quad\ \ \mathrm{max} = \varsigma(\mathrm{self}{:}\mathrm{Max}_{\mathrm{Op}}(\mathrm{Max}))\\
&\qquad\qquad\qquad\qquad \mathrm{classSelf.max}(\mathrm{Max}_{\mathrm{Op}})(\mathrm{fold}(\mathrm{self}))]),\\
&\quad\ \ \mathrm{n} = 0,\\
&\quad\ \ \mathrm{max} = \mathrm{pre\text{-}max}]
\end{aligned}
$$

$$\text{pre-max} : \forall(X \prec: Max_{Op})X^* \to X^* \to X^* \quad \triangleq$$
$$\quad \lambda(X \prec: Max_{Op})\ \lambda(self{:}X^*)\ \lambda(other{:}X^*)$$
$$\quad\quad \text{if } unfold(self).n > unfold(other).n \text{ then } self \text{ else } other$$

It is possible to check that pre-max is well typed.

The instantiations pre-max(Max_{Op}) and pre-max($MinMax_{Op}$) are both legal. Since pre-max has type $\forall(X \prec: Max_{Op})X^* \to X^* \to X^*$, this pre-method can be used as a component of a class of type MaxClass.

Moreover, a higher-order version of the rule for quantifier subtyping yields:

$$\forall(X \prec: Max_{Op})X^* \to X^* \to X^* \quad <: \quad \forall(X \prec: MinMax_{Op})X^* \to X^* \to X^*$$

so pre-max has type $\forall(X \prec: MinMax_{Op})X^* \to X^* \to X^*$ by subsumption, and hence pre-max can be reused as a component of a class of type MinMaxClass.

- Note. We expect following typings:

  if $X<\#Inc$ and $x:X$ then $x.n : Int$

  if $X<\#Inc$ and $x:X$ and $b:Int$ then $x.n{:=}b : X$

The higher-order interpretation induces the following term translations:

  if $X \prec: Inc_{Op}$ and $x:X^*$ then $unfold(x).n : Int$

  if $X \prec: Inc_{Op}$ and $x:X^*$ and $b:Int$ then $fold(unfold(x).n{:=}b) : X^*$

For the first typing, we have $unfold(x):X(X^*)$. Moreover, from $X \prec: Inc_{Op}$ we obtain $X(X^*) <: Inc_{Op}(X^*) = [n:Int, inc:X^*]$. Therefore, $unfold(x):[n:Int, inc:X^*]$, and $unfold(x).n:Int$.

For the second typing, we have again $unfold(x):X(X^*)$ with $X(X^*) <: [n:Int, inc:X^*]$. We then use a typing rule for field update in the target language. This rule says that if $a:A$, $c:C$, and $A <: [v:C,...]$ then $(a.v{:=}c) : A$. In our case, we have $unfold(x):X(X^*)$, $b:Int$, and $X(X^*) <: [n:Int, inc:X^*]$. We obtain $(unfold(x).n{:=}b) : X(X^*)$. Finally, by folding, we obtain $fold(unfold(x).n{:=}b) : X^*$.

Matching as F-bounded Subtyping

Type Operators

- We introduce a theory of type operators that will enable us to express various formal relationships between types. Alternatives interpretations of matching will become available.
- A type operator is a function from types to types.

$\lambda(X)B\{X\}$	maps each type X to a corresponding type $B\{X\}$
$B(A)$	applies the operator B to the type A

$$(\lambda(X)B\{X\})(A) = B\{A\}$$

- Notation for fixpoints:

F^*	abbreviates	$\mu(X)F(X)$	
A_{Op}	abbreviates	$\lambda(X)D\{X\}$	whenever $A \equiv \mu(X)D\{X\}$

- We obtain:

$$\mathrm{Max}_{Op} \equiv \lambda(X)[n{:}Int,\ max^{+}{:}X\to X]$$
$$\mathrm{MinMax}_{Op} \equiv \lambda(Y)[n{:}Int,\ max^{+}{:}Y\to Y,\ min^{+}{:}Y\to Y]$$

- The unfolding property of recursive types yields:

$$\mathrm{Max}_{Op}{*} = \mu(X)\,\mathrm{Max}_{Op}(X) = \mu(X)\,[n{:}Int,\ max^{+}{:}X\to X] = \mathrm{Max}$$
$$\mathrm{Max}_{Op}{*} = \mathrm{Max}_{Op}(\mu(X)\,\mathrm{Max}_{Op}(X)) = \mathrm{Max}_{Op}(\mathrm{Max})$$

- Note that A_{Op} is defined in terms of the syntactic form $\mu(X)D\{X\}$ of A. In particular, the unfolding $D\{A\}$ of A is not necessarily in a form such that $D\{A\}_{Op}$ is defined. Even if $D\{A\}_{Op}$ is defined, it need not equal A_{Op}. For example, consider:

$$D\{X\} \triangleq \mu(Y)\,X\to Y$$
$$A \triangleq \mu(X)\,D\{X\}$$
$$D\{A\} \equiv \mu(Y)\,A\to Y \qquad = A$$
$$A_{Op} \equiv \lambda(X)\,D\{X\}$$
$$D\{A\}_{Op} \equiv \lambda(Y)\,A\to Y \qquad \neq A_{Op}$$

- Thus, we may have two types A and B such that $A = B$ but $A_{Op} \neq B_{Op}$ (when recursive types are taken equal up to unfolding). This is a sign of trouble to come.

F-bounded Subtyping

- F-bounded subtyping was invented to support parameterization in the absence of subtyping.
- The property:

  $A <: B_{Op}(A)$ (A is a pre-fixpoint of B_{Op})

  is seen as a statement that A extends B.
- This view is justified because, for example, a recursive object type A such that $A <: [n:Int, max^+:A \rightarrow A]$ often has the shape $\mu(Y)[n:Int, max^+:Y \rightarrow Y, \ldots]$.

- Both Max and MinMax are pre-fixpoints of Max_{Op}:

$$
\begin{array}{ll}
Max <: Max_{Op}(Max) & (= Max) \\
MinMax & (= [n:Int,\ max^{+}:MinMax \to MinMax,\ min^{+}: \ldots\ ]) \\
\quad <: Max_{Op}(MinMax) & (= [n:Int,\ max^{+}:MinMax \to MinMax])
\end{array}
$$

So, we can parameterize over all types X with the property that X <: $Max_{Op}(X)$.

$$\forall(X<:Max_{Op}(X))B\{X\}$$

This form of parameterization leads to a general typing of pre-max, and permits the inheritance of pre-max:

$$
\begin{array}{l}
\text{pre-max} : \forall(X<:Max_{Op}(X))X\to X\to X \ \triangleq \\
\quad \lambda(X<:Max_{Op}(X))\ \lambda(self:X)\ \lambda(other:X) \\
\quad\quad \text{if } self.n>other.n \text{ then } self \text{ else } other
\end{array}
$$

$$\text{pre-max}(Max) : Max\to Max\to Max$$
$$\text{pre-max}(MinMax) : MinMax\to MinMax\to MinMax$$

The F-bounded Interpretation

- The central idea of the interpretation is:

$$A <\# B \quad\approx\quad A <: B_{Op}(A)$$
$$\forall(X<\#A)B\{X\} \quad\approx\quad \forall(X<:A_{Op}(X))B\{X\}$$

- However, this interpretation is not defined when the right-hand side of <# is a variable, as in the case of cascading quantifiers:

$$\forall(X<\#A)\ \forall(Y<\#X)\ ... \quad\approx\quad ?$$

  Since $\forall(X<:A_{Op}(X))\ \forall(Y<:X_{Op}(Y))$ ... does not make sense the type structure supported by this interpretation is somewhat irregular: type variables are not allowed in places where object types are allowed.

Reflexivity and Transitivity

- We would expect A <# A to hold, e.g. to justifying the instantiation f(A) of a polymorphic function f : ∀(X<#A)B. We have:

  $$A <\# A \qquad \approx \qquad A <: A_{Op}(A)$$

  with $A = A_{Op}(A)$ by the unfolding property of recursive types. However, if A is a type variable X, then X_{Op} is not defined, so $X <: X_{Op}(X)$ does not make sense.
  Hence, reflexivity does not hold in general.

- If A, B, and C are object types of the source language, then we would expect that A <# B and B <# C imply A <# C; this would mean:

  $$A <: B_{Op}(A) \quad \text{and} \quad B <: C_{Op}(B) \quad \text{imply} \quad A <: C_{Op}(A)$$

  As in the case of reflexivity, we run into difficulties with type variables.

Counterexample to Transitivity

- Worse, transitivity fails even for closed types, with the following counterexample:

$$A \triangleq \mu(X)[p^+: X \rightarrow Int,\ q: Int]$$
$$B \triangleq \mu(X)[p^+: X \rightarrow Int]$$
$$C \triangleq \mu(X)[p^+: B \rightarrow Int]$$

We have both A <# B and B <# C, but we do not have A <# C (because $[p^+:A\rightarrow Int,\ q:Int] <: [p^+:B\rightarrow Int]$ fails).

$$\begin{array}{lll} A & = [p^+: A\rightarrow Int,\ q: Int] & <: \\ B_{Op}(A) & = [p^+: A\rightarrow Int] & \end{array}$$

$$\begin{array}{lll} B & = [p^+: B\rightarrow Int] & <: \\ C_{Op}(B) & = [p^+: B\rightarrow Int] & \end{array}$$

$$\begin{array}{lll} A & = [p^+: A\rightarrow Int,\ q: Int] & \not<: \\ C_{Op}(A) & = [p^+: B\rightarrow Int] & \end{array}$$

- We can trace this problem back to the definition of D_{Op}, which depends on the exact syntax of the type D. Because of the syntactic character of that definition, two equal types may behave differently with respect to matching.

  In our example, we have B = C by the unfolding property of recursive types. Despite the equality B = C, we have A <# B but not A <# C !

Matching Self

- According to the F-bounded interpretation, two types that look rather different may match. Consider two types A and A' such that:

$$A \equiv \mu(X)[v_i{:}B_i{}^{i\in I}, m_j^+{:}C_j\{X\}{}^{j\in J}]$$
$$<\# \ \mu(X)[v_i{:}B_i{}^{i\in I}, m_j^+{:}C_j'\{X\}{}^{j\in J'}] \equiv A'$$

This holds when $A <: A'_{Op}(A)$, that is, when $[v_i{:}B_i{}^{i\in I}, m_j^+{:}C_j\{A\}{}^{j\in J}] <: [v_i{:}B_i{}^{i\in I}, m_j^+{:}C_j'\{A\}{}^{j\in J'}]$. It suffices that, for every $j\in J'$:

$$C_j\{A\} <: C_j'\{A\}$$

- For example, we have:

$$\mu(X)[v{:}Int, m^+{:}X] <\# \ \mu(X)[m^+{:}\ [v{:}Int]]$$

The variable X on the left matches the type [v:Int] on the right. Since X is the Self variable, we may say that Self matches not only Self but also other types (here [v:Int]). This treatment of Self is both sound and flexible. On the other hand, it can be difficult for a programmer to see whether two types match.

THE LANGUAGE O–3

Matching in O–3

- Many features of O–3 are familiar: for example, object types, class types, and single inheritance.
- The main new feature is a matching relation, written <#. The matching relation is defined only between object types, and between variables bounded by object types.
  - ~ An object type $A \equiv$ **Object**$(X)[\ldots]$ matches another object type $B \equiv$ **Object**$(X)[\ldots]$ (written A <# B) when all the object components of B are literally present in A, including any occurrence of the common variable X.

Object$(X)[l{:}X\to X,\ m{:}X]$ <# **Object**$(X)[l{:}X\to X]$	Yes
Object$(X)[l{:}X\to X,\ m{:}X]$ <: **Object**$(X)[l{:}X\to X]$	No

- Matching is the basis for inheritance in O–3. That is, if $A <\# B$, then a method of a class for B may be inherited as a method of a class for A.
  - In particular, binary methods can be inherited. For example, a method l of a class for **Object**$(X)[l{:}X\to X]$ can be inherited as a method of a class for **Object**$(X)[l{:}X\to X, m{:}X]$.
  - Matching does not support subsumption: when a has type A and $A <\# B$, it is not sound in general to infer that a has type B.
  - We will have that if A and B are object types and $A <: B$, then $A <\# B$. Moreover, if all occurrences of Self in B are covariant and $A <\# B$, then $A <: B$.

- With the loss of subsumption, it is often necessary to parameterize over all types that match a given type.
  - For example, a function with type $(\mathbf{Object}(X)[l{:}X\to X])\to C$ may have to be rewritten, for flexibility, with type $\mathbf{All}(Y{<}\#\mathbf{Object}(X)[l{:}X\to X])Y\to C$, enabling the application to an object of type $\mathbf{Object}(X)[l{:}X\to X,\ m{:}X]$.
- No subtype relation appears in the syntax of O–3, although subtyping is still used in its type rules.

Syntax of Types

Syntax of O–3 types

$A,B ::=$	types
$\quad X$	type variable
$\quad$ **Top**	maximum type
$\quad$ **Object**$(X)[l_i \upsilon_i{:}B_i\{X\}\ ^{i\in 1..n}]$	object type
$\quad$ **Class**(A)	class type
$\quad$ **All**$(X{<}\#A)B$	match-quantified type

Syntax of Programs

Syntax of O–3 terms

a,b,c ::=	terms
x	variable
object$(x{:}X{=}A)\ l_i{=}b_i\{X,x\}^{\,i\in 1..n}$ **end**	direct object construction
$a.l$	field/method selection
$a.l$:= **method**$(x{:}X{<}\#A)\ b$ **end**	update
new c	object construction from a class
root	root class
subclass of $c{:}C$ **with**$(x{:}X{<}\#A)$	subclass
$l_i{=}b_i\{X,x\}^{\,i\in n+1..n+m}$	additional attributes
override $l_i{=}b_i\{X,x\}^{\,i\in Ovr\subseteq 1..n}$ **end**	overridden attributes
$c\hat{\ }l(A,a)$	class selection
fun$(X{<}\#A)\ b$ **end**	match-polymorphic abstraction
$b(A)$	match-polymorphic instantiation

Abbreviations

Root $\triangleq$

  Class(**Object**(X)[])

class with$(x{:}X{<}\#A)\ l_i{=}b_i\{X,x\}^{\,i\in 1..n}$ **end** $\triangleq$

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

subclass of $c{:}C$ **with**$(x{:}X{<}\#A)$ ... **super**.l ... **end** $\triangleq$

  subclass of $c{:}C$ **with**$(x{:}X{<}\#A)$ ... $c\hat{\ }l(X,x)$ ... **end**

object$(x{:}X{=}A)$ ... l **copied from** c ... **end** $\triangleq$

  object$(x{:}X{=}A)$ ... $l{=}c\hat{\ }l(X,x)$ ... **end**

$a.l := b$ $\triangleq$ where $X,x \notin \mathrm{FV}(b)$ and $a{:}A$,

  $a.l :=$ **method**$(x{:}X{<}\#A)\ b$ **end** with A clear from context

Example: Points

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

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

- These definitions freely use covariant and contravariant occurrences of Self types. The liberal treatment of Self types in O–3 yields *CPoint* <# *Point*.

- In O–1, the same definitions of *Point* and *CPoint* are valid, but they are less satisfactory because *CPoint* <: *Point* fails; therefore the O–1 definitions adopt a different type for *eq*.

- In O–2, contravariant occurrences of Self types are illegal; therefore the O–2 definitions have a different type for *eq*, too.

~ We define two classes *pointClass* and *cPointClass* that correspond to the types *Point* and *CPoint*, respectively:

$$
\begin{array}{l}
\mathit{pointClass} : \mathbf{Class}(\mathit{Point}) \quad \triangleq \\
\qquad \mathbf{class\ with}\ (\mathit{self}\colon X{<}\#\mathit{Point}) \\
\qquad\qquad x = 0, \\
\qquad\qquad \mathit{eq} = \mathbf{fun}(\mathit{other}\colon X)\ \mathit{self}.x = \mathit{other}.x\ \mathbf{end}, \\
\qquad\qquad \mathit{mv} = \mathbf{fun}(\mathit{dx}\colon \mathit{Int})\ \mathit{self}.x := \mathit{self}.x{+}\mathit{dx}\ \mathbf{end} \\
\qquad \mathbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{cPointClass} : \mathbf{Class}(\mathit{CPoint}) \quad \triangleq \\
\qquad \mathbf{subclass\ of}\ \mathit{pointClass}\colon \mathbf{Class}(\mathit{Point}) \\
\qquad \mathbf{with}\ (\mathit{self}\colon X{<}\#\mathit{CPoint}) \\
\qquad\qquad c = \mathit{black} \\
\qquad \mathbf{override} \\
\qquad\qquad \mathit{eq} = \mathbf{fun}(\mathit{other}\colon X)\ \mathbf{super}.\mathit{eq}(\mathit{other})\ \mathit{and}\ \mathit{self}.c = \mathit{other}.c\ \mathbf{end}, \\
\qquad\qquad \mathit{mv} = \mathbf{fun}(\mathit{dx}\colon \mathit{Int})\ \mathbf{super}.\mathit{mv}(\mathit{dx}).c := \mathit{red}\ \mathbf{end} \\
\qquad \mathbf{end}
\end{array}
$$

- The subclass *cPointClass* could have inherited both *mv* and *eq*. However, we chose to override both of these methods in order to adapt them to deal with colors.

- In contrast with the corresponding programs in O–1 and O–2, no uses of typecase are required in this code. The use of typecase is not needed for accessing the color of a point after moving it. (Typecase is needed in O–1 but not in O–2.) Specifically, the overriding code for *mv* does not need a typecase on the result of **super**.*mv*(*dx*) in the definition of *cPointClass*.

~ Other code that moves color points does not need a typecase either:

$$cPoint : CPoint \triangleq \mathbf{new}\ cPointClass$$

$$movedColor : Color \triangleq cPoint.mv(1).c$$

~ Moreover, O–3 allows us to specialize the binary method *eq* as we have done in the definition of *cPointClass* (unlike O–2). This specialization does not require dynamic typing: we can write **super**.*eq*(*other*) without first doing a typecase on *other*.

~ Thus the treatment of points in O–3 circumvents the previous needs for dynamic typing. The price for this is the loss of the subtyping *CPoint* <: *Point*, and hence the loss of subsumption between *CPoint* and *Point*.

Example: Binary Trees

$Bin \triangleq$

$\quad \mathbf{Object}(X)[isLeaf: Bool,\ lft: X,\ rht: X,\ consLft: X \rightarrow X,\ consRht: X \rightarrow X]$

$binClass : \mathbf{Class}(Bin) \triangleq$

$\quad \mathbf{class\ with}(self: X <\# Bin)$

$\qquad isLeaf = true,$

$\qquad lft = self.lft,$

$\qquad rht = self.rht,$

$\qquad consLft = \mathbf{fun}(lft: X)\ ((self.isLeaf := false).lft := lft).\ rht := self\ \mathbf{end},$

$\qquad consRht = \mathbf{fun}(rht: X)\ ((self.isLeaf := false).lft := self).\ rht := rht\ \mathbf{end}$

$\quad \mathbf{end}$

$leaf : Bin \triangleq$

$\quad \mathbf{new}\ binClass$

- The definition of the object type *Bin* is the same one we could have given in O–1, but it would be illegal in O–2 because of the contravariant occurrences of *X*.
- The method bodies rely on some new facts about typing; in particular, if *self* has type *X* and $X <\# Bin$, then *self.lft* and *self.isLeaf*:=*false* have type *X*.

- Let us consider now a type *NatBin* of binary trees with natural number components.

$$\mathit{NatBin} \triangleq \mathbf{Object}(X)[n{:}\ \mathit{Nat},\ \mathit{isLeaf}{:}\ \mathit{Bool},\ \mathit{lft}{:}\ X,\ \mathit{rht}{:}\ X,\ \mathit{consLft}{:}\ X\to X,\ \mathit{consRht}{:}\ X\to X]$$

- We have $\mathit{NatBin} <\# \mathit{Bin}$, although $\mathit{NatBin}\ </:\ \mathit{Bin}$.
- If *b* has type *Bin* and *nb* has type *NatBin*, then *b*.*consLft*(*b*) and *nb*.*consLft*(*nb*) are allowed, but *b*.*consLft*(*nb*) and *nb*.*consLft*(*b*) are not.
- The methods *consLft* and *consRht* can be used as binary operations on any pair of objects whose common type matches *Bin*. O–3 allows inheritance of *consLft* and *consRht*. A class for *NatBin* may inherit *consLft* and *consRht* from *binClass*.

- Because *NatBin* is not a subtype of *Bin*, generic operations must be explicitly parameterized over all types that match *Bin*. For example, we may write:

$$selfCons : \mathbf{All}(X{<}\#Bin)X \rightarrow X \;\triangleq$$
$$\quad \mathbf{fun}(X{<}\#Bin)\ \mathbf{fun}(x{:}\ X)\ x.consLft(x)\ \mathbf{end}\ \mathbf{end}$$

$$selfCons(NatBin)(nb) : NatBin \qquad \text{for } nb : NatBin$$

- Explicit parameterization must be used systematically in order to guarantee future flexibility in usage, especially for object types that contain binary methods.

Example: Cells

- In this version, the proper methods are indicated with variance annotations $^{+}$; the *contents* and *backup* attributes are fields.

$$Cell \triangleq$$
$$\quad \mathbf{Object}(X)[contents: Nat,\ get^{+}: Nat,\ set^{+}: Nat \to X]$$

$$cellClass : \mathbf{Class}(Cell) \triangleq$$

class with(*self*: X<#*Cell*)
    contents = 0,
    get = *self.contents*,
    set = **fun**(*n*: *Nat*) *self.contents* := *n* **end**
end

$ReCell \triangleq$

$\quad \mathbf{Object}(X)[contents\colon Nat,\ get^{+}\colon Nat,\ set^{+}\colon Nat \to X,\ backup\colon Nat,\ restore^{+}\colon X]$

$reCellClass : \mathbf{Class}(ReCell) \triangleq$

$\quad$ **subclass of** *cellClass*:**Class**(*Cell*)

$\quad$ **with**(*self*: *X*<#*ReCell*)

$\qquad backup = 0,$

$\qquad restore = self.contents := self.backup$

$\quad$ **override**

$\qquad$ *set* = **fun**(*n*: *Nat*) *cellClass*^*set*(*X*, *self.backup* := *self.contents*)(*n*) **end**

$\quad$ end

- We can also write a version of *ReCell* that uses method update instead of a *backup* field:

```
ReCell'  ≜
    Object(X)[contents: Nat, get⁺: Nat, set⁺: Nat→X, restore: X]

reCellClass' : Class(ReCell')  ≜
    subclass of cellClass:Class(Cell)
    with(self: X<#ReCell')
        restore = self.contents := 0
    override
        set = fun(n: Nat)
                  let  m = self.contents
                  in   cellClass^set(X,
                           self.restore := method(y: X) y.contents := m end)
                       (n)
                  end
              end
    end
```

- We obtain *ReCell* <: *Cell* and *ReCell'* <: *Cell*, because of the covariance of *X* and the positive variance annotations on the method types of *Cell* where *X* occurs.
- On the other hand, we have also *ReCell* <# *Cell* and *ReCell'* <# *Cell*, and this does not depend on the variance annotations.
- A generic doubling function for all types that match *Cell* can be written as follows:

$$
\begin{array}{l}
\mathit{double} : \mathbf{All}(X \mathord{<}\#\mathit{Cell})\ X \to X \quad \triangleq \\
\qquad \mathbf{fun}(X \mathord{<}\#\mathit{Cell})\ \mathbf{fun}(x{:}\ X)\ x.\mathit{set}(2 * x.\mathit{get})\ \mathbf{end}\ \mathbf{end}
\end{array}
$$

Typing

Judgments

$E \vdash \diamond$	environment E is well formed
$E \vdash A$	A is a well formed type in E
$E \vdash A :: Obj$	A is a well formed object type in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash A <\# B$	A matches B in E
$E \vdash a : A$	a has type A in E

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

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

Type Formation Rules

Types

(Type Obj)

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

(Type X)

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

(Type Top)

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

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

$$\frac{E, X{<}\#A \vdash B_i \qquad \forall i \in 1..n}{E \vdash \mathbf{Class}(A)}$$

(Type All<#)

$$\frac{E, X{<}\#A \vdash B}{E \vdash \mathbf{All}(X{<}\#A)B}$$

Object Types

(Obj X)

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

(Obj Object) (l_i distinct, $\upsilon_i \in \{^{\circ}, ^{-}, ^{+}\}$)

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

- The judgments for types and object types are connected by the (Type Obj) rule.

Subtyping Rules

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)

$$\frac{\begin{array}{c} E \vdash \mathbf{Object}(X)[l_i \upsilon_i : B_i \ {}^{i \in 1..n+m}] \qquad E \vdash \mathbf{Object}(Y)[l_i \upsilon_i' : B_i' \ {}^{i \in 1..n}] \\ E, Y <: \mathbf{Top}, X <: Y \vdash \upsilon_i B_i <: \upsilon_i' B_i' \quad \forall\, i \in 1..n \qquad E, X <: \mathbf{Top} \vdash B_i \quad \forall\, i \in n+1..m \end{array}}{E \vdash \mathbf{Object}(X)[l_i \upsilon_i : B_i \ {}^{i \in 1..n+m}] <: \mathbf{Object}(Y)[l_i \upsilon_i' : B_i' \ {}^{i \in 1..n}]}$$

(Sub All<#)

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

(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)

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

Matching Rules

Matching

(Match Refl)

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

(Match Trans)

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

(Match X)

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

(Match Object) (l_i distinct)

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

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

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

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

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

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

$$\frac{E \vdash a : A' \qquad E \vdash A' <\# A \qquad E, X{<}\#A', x{:}X \vdash b : B_j \qquad \upsilon_j \in \{^{o}, ^{-}\} \qquad j \in 1..n}{E \vdash a.l_j := \mathbf{method}(x{:}X{<}\#A')b\ \mathbf{end} : A'}$$

(Val New)

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

(Val Root)

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

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

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

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

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

(Val Fun<#)

$$\frac{E, X\text{<\#}A \vdash b : B}{E \vdash \mathbf{fun}(X\text{<\#}A)\ b\ \mathbf{end} : \mathbf{All}(X\text{<\#}A)B}$$

(Val Appl<#)

$$\frac{E \vdash b : \mathbf{All}(X\text{<\#}A)B\{X\} \qquad E \vdash A' \text{<\#}\ A}{E \vdash b(A') : B\{\!\{A'\}\!\}}$$

Translation of O–3

- We give a translation into a functional calculus:

Syntax of $Ob_{\omega<:\mu}$

K,L ::=	kinds
Ty	types
$K \Rightarrow L$	operators from K to L
A,B ::=	constructors
X	constructor variable
Top	the biggest constructor at kind Ty
$[l_i \upsilon_i : B_i \ ^{i \in 1..n}]$	object type (l_i distinct, $\upsilon_i \in \{^o, ^-, ^+\}$)
$\forall(X<:A::K)B$	bounded universal type
$\mu(X)A$	recursive type
$\lambda(X::K)B$	operator
$B(A)$	operator application

$a,b ::=$	terms
x	variable
$[l{=}\varsigma(x_i{:}A_i)b_i \ ^{i\in 1..n}]$	object formation (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \varsigma(x{:}A)b$	method update
$\lambda(X{<:}A{::}K)b$	constructor abstraction
$b(A)$	constructor application
$fold(A,a)$	recursive fold
$unfold(a)$	recursive unfold

Translation (Sketch)

- The symbol $\cong$ means "informally translates to", with $\cong_{Ty}$ for translations that yield types, and $\cong_{Op}$ for translations that yield operators.
- We represent the translation of a term a by $\underline{a}$, the type translation of a type A by $\underline{A}$, and its operator translation by $\underline{\underline{A}}$.
- We say that a variable X is subtype-bound when it is introduced as $X{<:}A$ for some A; we say that X is match-bound when it is introduced as $X{<\#}A$ for some A.

Translation summary

$X \;\; \cong_{Op} \;\; X$ (where X is match-bound in the environment)

$\mathbf{Object}(X)[l_i \upsilon_i{:}B_i \;^{i\in 1..n}] \;\; \cong_{Op} \;\; \lambda(X)[l_i \upsilon_i{:}\underline{B_i} \;^{i\in 1..n}]$

$X \;\; \cong_{Ty} \;\; X$ (when X is subtype-bound in the environment)

$X \;\; \cong_{Ty} \;\; X^*$ (when X is match-bound in the environment)

$\mathbf{Top} \;\; \cong_{Ty} \;\; Top$

$\mathbf{Object}(X)[l_i \upsilon_i{:}B_i \;^{i\in 1..n}] \;\; \cong_{Ty} \;\; (\lambda(X)[l_i \upsilon_i{:}\underline{B_i} \;^{i\in 1..n}])^*$

$\mathbf{Class}(A) \;\; \cong_{Ty} \;\; [new^+{:}\underline{A},\, l_i^+{:}\forall(X{\prec:}\underline{\underline{A}})X^*{\to}\underline{B_i} \;^{i\in 1..n}]$
where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i{:}B_i \;^{i\in 1..n}]$

$\mathbf{All}(X{<\#}A)B \;\; \cong_{Ty} \;\; \forall(X{\prec:}\underline{\underline{A}})\underline{B}$

$$x \;\cong\; x$$

$$\textbf{object}(x{:}A)\ l_i{=}b_i\{x\}^{\,i\in 1..n}\ \textbf{end} \;\cong\; fold(\underline{A},[l_i{=}\varsigma(x{:}\underline{\underline{A}}(\underline{A}))\underline{b_i}\{\!\{fold(\underline{A},x)\}\!\}^{\,i\in 1..n}])$$

$$a.l_j \;\cong\; unfold(\underline{a}).l_j$$

$$a.l_j := \textbf{method}(x{:}A')b\{x\}\ \textbf{end} \;\cong$$
$$fold(\underline{A}',unfold(\underline{a}).l_j \Leftarrow \varsigma(x{:}\underline{\underline{A}}'(\underline{A}'))\underline{b}\{\!\{fold(\underline{A}',x)\}\!\})$$

$$\textbf{new}\ c \;\cong\; \underline{c}.new$$

$$\textbf{root} \;\cong\; [new{=}\varsigma(z{:}[new^{+}{:}\mu(X)[]])fold(\mu(X)[],[])]$$

$$\textbf{subclass of}\ c'{:}C'\ \textbf{with}(x{:}X{<}\#A)\ l_i{=}b_i^{\,i\in n+1..n+m}\ \textbf{override}\ l_i{=}b_i^{\,i\in Ovr}\ \textbf{end} \;\cong$$
$$[new{=}\varsigma(z{:}\underline{C})fold(\underline{A},[l_i{=}\varsigma(s{:}\underline{\underline{A}}(\underline{A}))z.l_i(\underline{\underline{A}})(fold(\underline{A},s))^{\,i\in 1..n+m}])$$
$$l_i{=}\varsigma(z{:}\underline{C})\ \underline{c}'.l_i^{\,i\in 1..n-Ovr},$$
$$l_i{=}\varsigma(z{:}\underline{C})\lambda(X{<:}\underline{\underline{A}})\lambda(x{:}X^*)\underline{b_i}^{\,i\in Ovr\cup n+1..n+m}]$$
$$\text{where } C \equiv \textbf{Class}(A)$$

$$c\hat{\ }l_j(A',a) \;\cong\; \underline{c}.l_j(\underline{\underline{A}}')(\underline{a})$$

$$\textbf{fun}(X{<}\#A)b\ \textbf{end} \;\cong\; \lambda(X{<:}\underline{\underline{A}})\underline{b}$$

$$b(A') \;\cong\; \underline{b}(\underline{\underline{A}}')$$

Summary on Matching

- There are situations in programming where one would like to parameterize over all "extensions" of a recursive object type, rather than over all its subtypes.
- Both F-bounded subtyping and higher-order subtyping can be used in explaining the matching relation.

  We have presented two interpretations of matching:

$A <\# B$	$\approx$	$A <: B_{Op}(A)$	(F-bounded interpretation)
$A <\# B$	$\approx$	$A_{Op} \prec: B_{Op}$	(higher-order interpretation)

- Both interpretations can be soundly adopted, but they require different assumptions and yield different rules. The higher-order interpretation validates reflexivity and transitivity.

  Technically, the higher-order interpretation need not assume the equality of recursive types up to unfolding (which seems to be necessary for the F-bounded interpretation). This leads to a simpler underlying theory, especially at higher order.

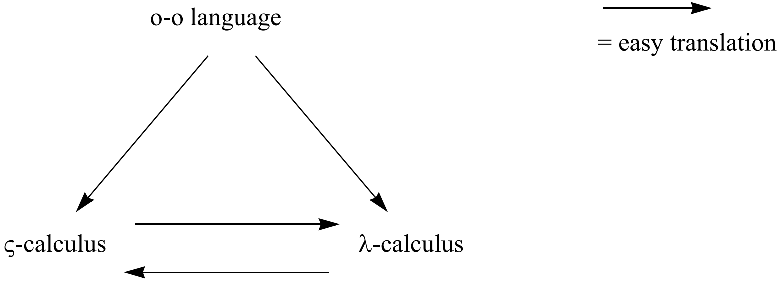
- Thus, we believe that the higher-order interpretation is preferable; it should be a guiding principle for programming languages that attempt to capture the notion of type extension.
- Matching achieves "covariant subtyping" for Self types and inheritance of binary methods at the cost not validating subsumption.
- Subtyping is still useful when subsumption is needed. Moreover, matching is best understood as higher-order subtyping. Therefore, subtyping is still needed as a fundamental concept, even though the syntax of a programming language may rely only on matching.

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.

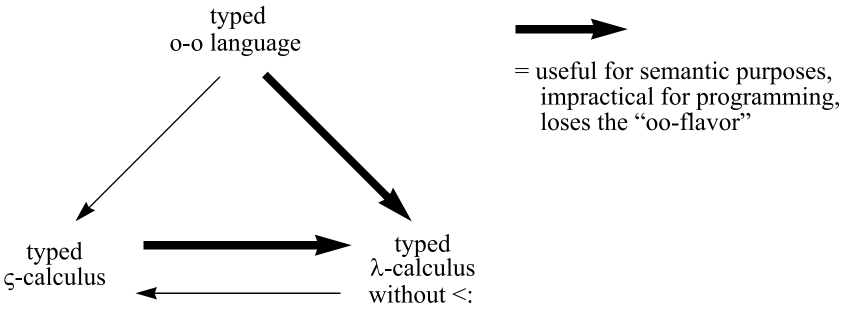


Type-Preserving Translations

- Give insights into the nature of object-oriented typing and subsumption/coercion.

- Object types = recursive records-of-functions types.

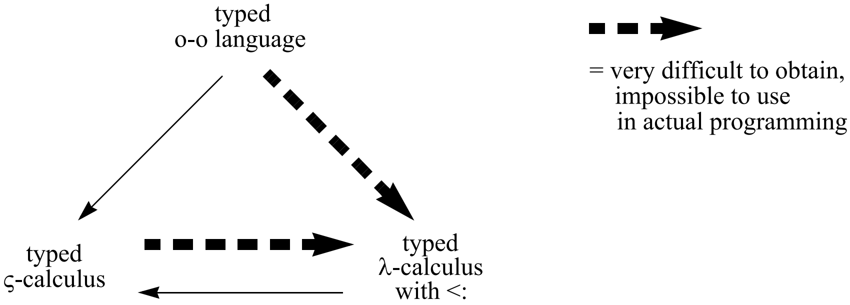
$$[l_i{:}B_i{}^{i \in 1..n}] \quad \triangleq \quad \mu(X)\langle l_i{:}X \rightarrow B_i{}^{i \in 1..n}\rangle$$



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

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.

Foundations

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

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.

EXTRA SLIDES

Unsoundness of Naive Object Subtyping with Binary Methods

$\text{Max} \triangleq \mu(X)[\text{n:Int}, \text{max}^+\text{:}X \rightarrow X]$

$\text{MinMax} \triangleq \mu(Y)[\text{n:Int}, \text{max}^+\text{:}Y \rightarrow Y, \text{min}^+\text{:}Y \rightarrow Y]$

Consider:

$\text{m : Max} \triangleq [\text{n} = 0, \text{max} = \ldots\ ]$

$\text{mm : MinMax} \triangleq$

$\quad [\text{n} = 0, \text{min} = \ldots\ ,$

$\quad \text{max} = \varsigma(\text{s:MinMax})\ \lambda(\text{o:MinMax})$

$\qquad\qquad \text{if o.min(o).n} > \text{s.n then o else s}]$

Assume MinMax <: Max, then:

mm : Max (by subsumption)

mm.max(m) : Max

But (Eiffel, O_2, ...):

$\text{mm.max(m)} \rightsquigarrow \text{if m.min(m).n} > \text{mm.n then m else mm} \rightsquigarrow \text{CRASH!}$

Unsoundness of Covariant Object Types

With record types, it is unsound to admit covariant subtyping of record components in presence of imperative field update. With object types, the essence of that couterexample can be reproduced even in a purely functional setting.

$U \triangleq []$ 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 for object subtyping

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

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!

Unsoundness of Method Extraction

It is unsound to have an operation that extracts a method as a function.

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

$$\frac{E \vdash a : A \qquad j\in 1..n}{E \vdash a{\cdot}l_j : A\to B_j}$$

(Eval Extract) (where $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$, $a \equiv [l_i{=}\varsigma(x_i{:}A')b_i\ ^{i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad j\in 1..n}{E \vdash a{\cdot}l_j \leftrightarrow \lambda(x_j{:}A)b_j : A\to B_j}$$

$P \triangleq [f{:}[]]$

$Q \triangleq [f{:}[], y{:}[]]$ $\qquad Q <: P$

$p : P \triangleq [f{=}[]]$

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

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

hence $\quad q{\cdot}f : P\to[] \quad$ that is $\lambda(s{:}Q)s.y : P\to[]$

But $\quad q.f(p) \quad$ fails!

Unsoundness of a Naive Recursive Subtyping Rule

Assume:

$$A \;\equiv\; \mu(X)X\to Nat \;\;<:\;\; \mu(X)X\to Int \;\equiv\; B$$

Let:		Type-erased:
$f : Nat \to Nat$	(*given*)	
$a : A = fold(A, \lambda(x{:}A)\ 3)$		$= \lambda(x)\ 3$
$b : B = fold(B, \lambda(x{:}B)\ {-3})$		$= \lambda(x)\ {-3}$
$c : A = fold(A, \lambda(x{:}A)\ f(unfold(x)(a)))$		$= \lambda(x)\ f(x(a))$

By subsumption:

$c : B$

Hence:

$unfold(c)(b) : Int$	Well-typed!	$= c(b)$

But:

$unfold(c)(b) = f(-3)$ Error!

Operationally Sound Update

Luca Cardelli

Digital Equipment Corporation
Systems Research Center

Outline

- The type rules necessary for “sufficiently polymorphic” update operations on records and objects are based on unusual operational assumptions.

- These update rules are sound operationally, but not denotationally (in standard models). They arise naturally in type systems for programming, and are not easily avoidable.

- Thus, we have a situation where operational semantics is clearly more advantageous than denotational semantics.

- However (to please the semanticists) I will show how these operationally-based type systems can be translated into type systems that are denotationally sound.

The polymorphic update problem

L.Cardelli, P.Wegner

“*The need for bounded quantification arises very frequently in object-oriented programming. Suppose we have the following types and functions:*

type $Point = [x: Int, y: Int]$
value $moveX_0 = \lambda(p: Point, dx: Int)\ p.x := p.x + dx; p$
value $moveX = \lambda(P <: Point)\ \lambda(p: P, dx: Int)\ p.x := p.x + dx; p$

It is typical in (type-free) object-oriented programming to reuse functions like moveX on objects whose type was not known when moveX was defined. If we now define:

type $Tile = [x: Int, y: Int, hor: Int, ver: Int]$

we may want to use moveX to move tiles, not just points.”

$Tile <: Point$

$moveX_0([x=0, y=0, hor=1, ver=1], 1).hor$ fails

$moveX(Tile)([x=0, y=0, hor=1, ver=1], 1).hor$ succeeds

- In that paper, bounded quantification was justified as a way of handling polymorphic update, and was used in the context of *imperative* update.

- The examples were inspired by object-oriented applications. Object-oriented languages combine subtyping and polymorphism with state encapsulation, and hence with imperative update. Some form of polymorphic update is inevitable.

- Simplifying the situation a bit, let's consider the equivalent example in a functional setting. We might hope to achieve the following typing:

$$bump \triangleq \lambda(P <: Point)\,\lambda(p{:}\,P)\; p.x := p.x + 1$$

$$bump : \forall(P <: Point)\, P \to P$$

But ...

There is no bump there!

Neither semantically

J.Mitchell

In standard models, the type $\forall(P<:Point)P\rightarrow P$ contains only the identity function.

Consider $\{p\}$ for any $p\in Point$. If $f : \forall(P<:Point)P\rightarrow P$, then $f(\{p\}) : \{p\}\rightarrow\{p\}$, therefore f must map every point to itself, and must be the identity.

Nor parametrically

M.Abadi, L.Cardelli, G.Plotkin

By parametricity (for bounded quantifiers), we can show that if $f : \forall(P<:Point)P\rightarrow P$, then $\forall(P<:Point)\ \forall(x:P)\ f(P)(x) =_P x$. Thus f is an identity.

Nor by standard typing rules

As shown next ...

The simple rule for update

(Val Simple Update)

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

- According to this rule, bump does not typecheck as desired:

$$bump \;\triangleq\; \lambda(P <: Point)\, \lambda(p{:}\, P)\; p.x := p.x + 1$$

We must go from $p{:}P$ to $p{:}Point$ by subsumption before we can apply the rule. Therefore we obtain only:

$$bump : \forall(P <: Point)\; P \rightarrow Point$$

The “structural” rule for update

(Val Structural Update)

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

- According to this rule, bump typechecks as desired, using the special case where A is a type variable.

$$bump \triangleq \lambda(P <: Point)\, \lambda(p{:}\, P)\, p.x := p.x + 1$$

$$bump : \forall(P <: Point)\, P \to P$$

- Therefore, (Val Structural Update) is not sound in most semantic models, because it populates the type $\forall(P{<:}Point)P{\to}P$ with a non-identity function.

- However, (Val Structural Update) is in practice highly desirable, so the interesting question is under which conditions it is sound.

Can't allow too many subtypes

- Suppose we had:

  $$BoundedPoint \;\triangleq\; \{x: 0..9,\, y: 0..9\}$$
  $$BoundedPoint <: Point$$

  then:

  $$bump(BoundedPoint)(\{x{=}9,\, y{=}9\}) : BoundedPoint$$

  unsound!

- To recover from this problem, the subtyping rule for records/objects must forbid certain subtypings:

(Sub Object)

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

- Therefore, for soundness, the rule for structural updates makes implicit assumptions about the subtype relationships that may exist.

Relevant rules for structural update

(Sub Object)

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

(Val Subsumption)

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

(Val Object)

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

(Val Structural Update)

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

(Red Update)

$$\frac{\vdash a \leadsto [l_i{=}v_i{}^{i\in 1..n}] \quad \vdash b \leadsto v \quad j \in 1..n}{\vdash a.l_j{:=}b \leadsto [l_j{=}v,\ l_i{=}v_i{}^{i\in 1..n-\{j\}}]}$$

The structural subtyping lemmas

Lemma (Structural subtyping)

If $E \vdash [l_i{:}B_i{}^{i\in I}]{<:}C$ then either $C \equiv Top$, or $C \equiv [l_i{:}B_i{}^{i\in J}]$ with $J{\subseteq}I$.

If $E \vdash C{<:}[l_i{:}B_i{}^{i\in J}]$ then either $C \equiv [l_i{:}B_i{}^{i\in I}]$ with $J{\subseteq}I$,

or $C \equiv X_1$ and E contains a chain $X_1 <: \ldots <: X_p <: [l_i{:}B_i{}^{i\in I}]$ with $J{\subseteq}I$.

Proof

By induction on the derivations of $E \vdash [l_i{:}B_i{}^{i\in I}]{<:}C$ and $E \vdash C{<:}[l_i{:}B_i{}^{i\in I}]$.

□

Soundness by subject reduction

Theorem (Subject reduction)

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

Proof **By induction on the derivation of $\vdash a \leadsto v$.**

Case (Red Update)

$$\frac{\vdash c \leadsto [l_i{=}z_i{}^{i\in 1..n}] \qquad \vdash b \leadsto w \qquad j\in 1..n}{\vdash c.l_j{:=}b \leadsto [l_j{=}w,\ l_i{=}z_i{}^{i\in 1..n-\{j\}}]}$$

By hypothesis $\varnothing \vdash c.l_j{:=}b : A$. This must have come from (1) an application of (Val Structural Update) with assumptions $\varnothing \vdash c : C$, and $\varnothing \vdash C <: D$ where $D \equiv [l_j{:}B_j, ...]$, and $\varnothing \vdash b : B_j$, and with conclusion $\varnothing \vdash c.l_j{:=}b : C$, followed by (2) a number of subsumption steps implying $\varnothing \vdash C <: A$ by transitivity.

By induction hypothesis, since $\varnothing \vdash c : C$ and $\vdash c \leadsto z \equiv [l_i{=}z_i{}^{i\in 1..n}]$, we have $\varnothing \vdash z : C$.

By induction hypothesis, since $\varnothing \vdash b : B_j$ and $\vdash b \leadsto w$, we have $\varnothing \vdash w : B_j$.

Now, $\varnothing \vdash z : C$ must have come from (1) an application of (Val Object) with assumptions $\varnothing \vdash z_i : B_i'$ and $C' \equiv [l_i'{:}B_i'{}^{i\in 1..n}]$, and with conclusion $\varnothing \vdash z : C'$, followed by (2) a number of subsumption steps implying $\varnothing \vdash C' <: C$ by transitivity. By transitivity, $\varnothing \vdash C' <: D$. Hence by the Structural Subtyping Lemma, we must have $B_j \equiv B_j'$. Thus $\varnothing \vdash w : B_j'$. Then, by (Val Object), we obtain $\varnothing \vdash [l_j{=}w,\ l_i{=}z_i{}^{i\in 1..n-\{j\}}] : C'$. Since $\varnothing \vdash C'{<:}A$ by transitivity, we have $\varnothing \vdash [l_j{=}w,\ l_i{=}z_i{}^{i\in 1..n-\{j\}}] : A$ by subsumption.

Other structural rules

- Rules based on structural assumptions (structural rules, for short) are not restricted to record/ object update. They also arise in:
  - ~ method invocation with Self types,
  - ~ object cloning,
  - ~ class encodings,
  - ~ unfolding recursive types.

- The following is one of the simplest examples of the phenomenon (although not very useful in itself):

A structural rule for product types

M.Abadi

- The following rule for pairing enables us to mix two pairs a and b of type C into a new pair of the same type. The only assumption on C is that it is a subtype of a product type $B_1 \times B_2$.

$$\frac{E \vdash C <: B_1 \times B_2 \qquad E \vdash a : C \qquad E \vdash b : C}{E \vdash \langle fst(a), snd(b) \rangle : C}$$

  The soundness of this rule depends on the property that every subtype of a product type $B_1 \times B_2$ is itself a product type $C_1 \times C_2$.

- This property is true operationally for particular systems, but fails in any semantic model where subtyping is interpreted as the subset relation. Such a model would allow the set $\{a,b\}$ as a subtype of $B_1 \times B_2$ whenever a and b are elements of $B_1 \times B_2$. If a and b are different, then $\langle fst(a), snd(b) \rangle$ is not an element of $\{a,b\}$. Note that $\{a,b\}$ is not a product type.

A structural rule for recursive types

M.Abadi, L.Cardelli, R.Viswanathan

- In the paper "An Interpretation of Objects and Object types" we give a translation of object types into ordinary types:

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

  this works fine for non-structural rules.

- In order to validate a structural update rule in the source calculus, we need a structural update rule in the target calculus. It turns out that the necessary rule is the following, which is operationally sound:

$$\frac{E \vdash C <: \mu(X)B\{X\} \qquad E \vdash a : C}{E \vdash unfold(a) : B\{\!\{C\}\!\}}$$

A structural rule for method invocation

- In the context of object types with Self types:

(Val Select)

$$\frac{E \vdash a : A \qquad E \vdash A <: Obj(X)[l_i{:}B_i\{X\}^{\,i\in 1..n}] \qquad j\in 1..n}{E \vdash a.l_j : B_j\{\!\{A\}\!\}}$$

This structural rule is necessary to “encapsulate” structural update inside methods:

$$A \triangleq Obj(X)[n{:}\ Int,\ bump{:}\ X]$$

$$\lambda(Y <: A)\ \lambda(y{:}\ Y)\ y.bump$$

$$:\ \ \forall(Y <: A)\ Y \rightarrow Y$$

Structural rules and class encodings

Types of the form $\forall(X{<:}A)X{\rightarrow}B\{X\}$ are needed also for defining classes as collections of pre-methods. Each pre-method must work for all possible subclasses, parametrically in self, so that it can be inherited.

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

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

$$Bump \triangleq Obj(X)[n{:}\ Int,\ bump{:}\ X]$$

$$Class(Bump) \triangleq [new{:}\ Bump,\ bump{:}\ \forall(X{<:}Bump)X{\rightarrow}X]$$

$$c : Class(Bump) \triangleq$$

$$[new = \varsigma(c{:}\ Class(Bump))\ [n = 0,\ bump = \varsigma(s{:}\ Bump)\ c.bump(Bump)(s)],$$
$$bump = \lambda(X{<:}Bump)\ \lambda(x{:}X)\ x.n{:=}x.n{+}1\}]$$

A structural rule for cloning

- In the context of imperative object calculi:

(Val Clone)

$$\frac{E \vdash a : A \qquad E \vdash A <: [l_i{:}B_i\ ^{i\in 1..n}] \qquad j\in 1..n}{E \vdash clone(a) : A}$$

This structural rule is necessary for bumping and returning a clone instead of the original object:

$$bump \quad \triangleq \quad \lambda(P <: Point)\ \lambda(p{:}\ P)\ clone(p).x := p.x + 1$$

$$bump : \forall(P <: Point)\ P \rightarrow P$$

Comments

- Structural rules are quite satisfactory. The operational semantics is the right one, the typing rules are the right ones for writing useful programs, and the rules are sound for the semantics.

- We do not have a denotational semantics (yet?). (The paper “Operations on Records” by L.Cardelli and J.Mitchell contains a limited model for structural update; no general models seems to be known.)

- Even without a denotational semantcs, there is an operational semantics from which one could, hopefully, derive a theory of typed equality.

- Still, I would like to understand in what way a type like $\forall(X<:\mathrm{Point})X\to X$ does not mean what most people in this room might think.

- Insight may come from translating a calculus with structural rules, into one without structural rules for which we have a standard semantics.

Translating away structural rules

- The "Penn translation" can be used to map $F_{<:}$ into F by threading *coercion* functions.
- Similarly, we can map an $F_{<:}$-like calculus with structural rules into a normal $F_{<:}$-like calculus by threading *update* functions (*c.f.* M.Hofmann and B.Pierce: Positive <:).
- Example :

$$f : \forall(X <: [l: Int])\, X \to X \;\triangleq$$
$$\lambda(X <: [l: Int])\, \lambda(x: X)\, x.l := 3$$

$$f\,([l: Int])$$

(N.B. the update $x.l$:=3 uses the structural rule)

translates to:

$$f : \forall(X <: [l: Int])\, [l: X{\to}Int{\to}X] \to X \to X \;\triangleq$$
$$\lambda(X <: [l: Int])\, \lambda(\pi_X: [l: X{\to}Int{\to}X\,])\, \lambda(x: X)\, \pi_X.l(x)(3)$$

$$f\,([l: Int])\,([l = \lambda(x: [l: Int])\, \lambda(y: Int)\, x.l := y])$$

(N.B. the update $x.l$:=y uses the non-structural rule)

- Next: a simplified, somewhat ad-hoc, calculus to formalize this translation.

Syntax

$A,B ::=$	types
X	type variable
$[l_i{:}B_i \;^{i\in 1..n}]$	object type (l_i distinct)
$A\to B$	function types
$\forall(X<:[l_i{:}B_i \;^{i\in 1..n}])B$	bounded universal type
$a,b ::=$	terms
x	variable
$[l_i=\varsigma(x_i{:}A_i)b_i \;^{i\in 1..n}]$	object (l_i distinct)
$a.l$	method invocation
$a.l\Leftarrow\varsigma(x{:}A)b$	method update
$\lambda(x{:}A)b$	function
$b(a)$	application
$\lambda(X<:[l_i{:}B_i \;^{i\in 1..n}])b$	polymorphic function
$b(A)$	polymorphic instantiation

- We consider method update instead of field update ($a_A.l{:=}b \;\triangleq\; a.l\Leftarrow\varsigma(x{:}A)b$).
- We do not consider object types with Self types.
- We do not consider arbitrary bounds for type variables, only object-type bounds.

Environments

(Env $\emptyset$)

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

(Env x)

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

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

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

Types

(Type $X<:$)

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

(Type Object) (l_i distinct)

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

(Type Arrow)

$$\frac{E \vdash A \qquad E \vdash B}{E \vdash A \rightarrow B}$$

(Type All<:)

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

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 Object) (l_i distinct)

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

(Sub Arrow)

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

(Sub All)

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

Typing

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

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

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

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

(Val Fun)

$$\frac{E, x{:}A \vdash b : B}{E \vdash \lambda(x{:}A)b : A \rightarrow B}$$

(Val Appl)

$$\frac{E \vdash b : A \rightarrow B \qquad E \vdash a : A}{E \vdash b(a) : B}$$

(Val Fun2<:)

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

(Val Appl2<:) (where $A' \equiv [l_i{:}B_i\ ^{i \in 1..n}]$ or $A' \equiv Y$)

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

- The source system for the translation is the one given above. The target system is the one given above minus the (Val Update X) rule.
- Derivations in the source system can be translated to derivations that do not use (Val Update X). The following tables give a slightly informal summary of the translation on derivations.

Translation of Environments

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

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

$$\langle\!\langle E, X{<:}[l_i{:}B_i{}^{i\in 1..n}] \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, X{<:}\langle\!\langle [l_i{:}B_i{}^{i\in 1..n}] \rangle\!\rangle, \pi_X{:}[l_i{:}X\to(X\to\langle\!\langle B_i \rangle\!\rangle)\to X^{\,i\in 1..n}]$$

where each l_i: $X\to(X\to\langle\!\langle B_i \rangle\!\rangle)\to X$ is an updator that takes an object of type X, takes a pre-method for X (of type $X\to\langle\!\langle B_i \rangle\!\rangle$), updates the i-th method of the object, and returns the modified object of type X.

Translation of Types

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

$$\langle\!\langle [l_i{:}B_i{}^{i\in 1..n}]\rangle\!\rangle \triangleq [l_i{:}\langle\!\langle B_i\rangle\!\rangle{}^{i\in 1..n}]$$

$$\langle\!\langle A\to B\rangle\!\rangle \triangleq \langle\!\langle A\rangle\!\rangle\to\langle\!\langle B\rangle\!\rangle$$

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

- N.B. the translation preserves subtyping. In particular:

$$\langle\!\langle \forall(X<:[l_i{:}B_i{}^{i\in 1..n}])B\rangle\!\rangle <: \langle\!\langle \forall(X<:[l_i{:}B_i{}^{i\in 1..n+m}])B\rangle\!\rangle$$

since:

$$\forall(X<:\langle\!\langle [l_i{:}B_i{}^{i\in 1..n}]\rangle\!\rangle)\ [l_i{:}X\to(X\to\langle\!\langle B_i\rangle\!\rangle)\to X{}^{i\in 1..n}]\to\langle\!\langle B\rangle\!\rangle \quad <:$$
$$\forall(X<:\langle\!\langle [l_i{:}B_i{}^{i\in 1..n+m}]\rangle\!\rangle)\ [l_i{:}X\to(X\to\langle\!\langle B_i\rangle\!\rangle)\to X{}^{i\in 1..n+m}]\to\langle\!\langle B\rangle\!\rangle$$

- We have a calculus with polymorphic update where quantifier and arrow types are contravariant on the left (*c.f.* Positive Subtyping).

Translation of Terms

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

$\langle\!\langle [l_i{=}(x_i{:}A_i)b_i \;^{i\in 1..n}] \rangle\!\rangle \triangleq [l_i{=}\varsigma(x_i{:}\langle\!\langle A_i \rangle\!\rangle)\langle\!\langle b_i \rangle\!\rangle \;^{i\in 1..n}]$

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

$\langle\!\langle a.l{\Leftarrow}\varsigma(x{:}A)b \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l{\Leftarrow}(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b \rangle\!\rangle)$ for (Val Update Obj)

$\langle\!\langle a.l{\Leftarrow}\varsigma(x{:}X)b \rangle\!\rangle \triangleq \pi_X.l(\langle\!\langle a \rangle\!\rangle)(\lambda(x{:}X)\langle\!\langle b \rangle\!\rangle)$ for (Val Update X)

$\langle\!\langle \lambda(x{:}\mathrm{A})b \rangle\!\rangle \triangleq \lambda(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b \rangle\!\rangle$

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

$\langle\!\langle \lambda(X{<:}[l_i{:}B_i \;^{i\in 1..n}])b \rangle\!\rangle \triangleq$

$\lambda(X{<:}\langle\!\langle [l_i{:}B_i \;^{i\in 1..n}] \rangle\!\rangle)\ \lambda(\pi_X{:}[l_i{:}X{\to}(X{\to}\langle\!\langle B_i \rangle\!\rangle){\to}X \;^{i\in 1..n}])\ \langle\!\langle b \rangle\!\rangle$

$\langle\!\langle b(A) \rangle\!\rangle \triangleq$ for $A = [l_i{:}B_i \;^{i\in 1..n}]$

$\langle\!\langle b \rangle\!\rangle(\langle\!\langle A \rangle\!\rangle)\ ([l_i = \lambda(x_i{:}\langle\!\langle A \rangle\!\rangle)\ \lambda(f{:}\langle\!\langle A \rangle\!\rangle{\to}\langle\!\langle B_i \rangle\!\rangle)\ x.l_i{\Leftarrow}\varsigma(z{:}\langle\!\langle A \rangle\!\rangle)f(z) \;^{i\in 1..n}])$

$\langle\!\langle b(Y) \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle(Y)(\pi_Y)$

Conclusions

- Structural rules for polymorphic update are sound for operational semantics. They work equally well for functional and imperative semantics.

- Structural rules can be translated into non structural rules. I have shown a translation for a restricted form of quantification.

- Theories of equality for systems with structural rules have not been studied directly yet. Similarly, theories of equality induced by the translation have not been studied.