The search for a “good” static typing discipline in object-oriented languages

Kim B. Bruce

Williams College
&
Newton Institute

Goals

Why are static type-checking systems hard to design for OOL's?

- OO languages: Subtype vs. Subclass.
- Why simple systems aren't good enough.
- Increasing flexibility.
- Adding polymorphism.
- Stepping back from complexity.

OO Language Features

- Objects
  - Hidden instance variables for state.
  - Methods representing operations.
- Classes (*extensible object generators*).
- Computation via message sending.
- Access to operations of `self` and `super`.
- Subtype polymorphism.
- Subclasses.

Sample Program

```
class Point
   var
      x = 0: Int;
      y = 0: Int
   methods
      fun getx(): Int {return x}
      fun gety(): Int {return y}
      proc move(dx, dy: Int)
         {x := x + dx; y := y + dy}
end class;

var
   pt: PointType;
begin
   pt := new Point;
   pt.move(3,2);
   if pt.getx() = 4 then ...
end
```

Subtype Polymorphism

`S` is a *subtype* of `T`, written $\mathrm{S} <: \mathrm{T}$,

iff

a value of type `S` can be used in any context expecting a value of type `T`.

Subsumption: `e: S` & $\mathrm{S} <: \mathrm{T} \Rightarrow$ `e: T`.

If $\mathrm{S} <: \mathrm{T}$, a value of type `S` can be

- used for a parameter of type `T`.
- assigned to a variable of type `T`.

Important: Value of expression can correspond to subtype of its *static* type.

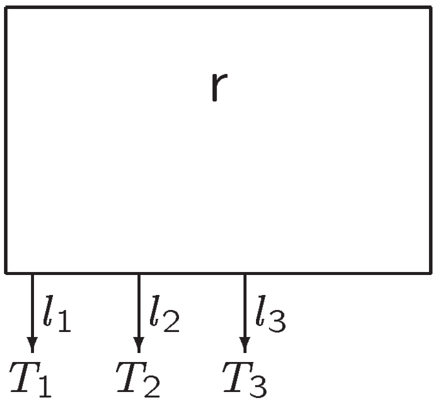
Immutable Records

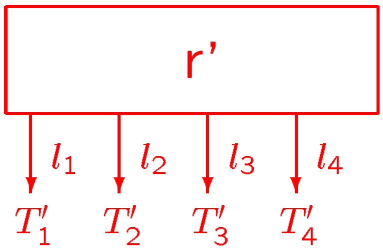
If r: $\{\mathtt{l_i} : \mathtt{T_i}\}_{1 \le i \le k}$ then $\mathtt{r.l_i} : \mathtt{T_i}$.

When is $\{\mathtt{l_i} : \mathtt{T'_i}\}_{1 \le i \le n}$ <: $\{\mathtt{l_i} : \mathtt{T_i}\}_{1 \le i \le k}$?

Suppose r': $\{\mathtt{l_i} : \mathtt{T'_i}\}_{1 \le i \le n}$

Still need $\mathtt{r'.l_i} : \mathtt{T_i}$.





$$\{\mathtt{l_i} : \mathtt{T'_i}\}_{1\le i\le n} <: \{\mathtt{l_i} : \mathtt{T_i}\}_{1\le i\le k}$$

if

$k \le n$ and for all $1 \le i \le k, \mathtt{T'_i} <: \mathtt{T_i}$.

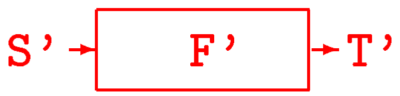
Functions

If `F`: $S \to T$ and `s: S` then `F(s): T`

When is $S' \to T' <: S \to T$?

If `F'`: $S' \to T'$, need `F'(s): T`.

$$\mathtt{S} \longrightarrow \boxed{\quad\quad\mathtt{F}\quad\quad} \longrightarrow \mathtt{T}$$



$$S' \to T' <: S \to T$$

if

$$S <: S' \text{ and } T' <: T.$$

Subtyping *contravariant* for parameters, *covariant* for result.

Variables

Variables can be *suppliers* of values &
receivers of values.

If **x** is a vble of type T, write **x: ref T**.

To replace vble **x: ref T** by **x': ref T'** in:

<u>expression</u>: **... x ...**

Need $\mathtt{T}' <: \mathtt{T}$.

<u>assignment</u>: **x := e** where **e: T**.

Need $\mathtt{T} <: \mathtt{T}'$.

Supplier: *covariant*; Receiver: *contravariant*

Thus $\mathtt{ref}\ \mathtt{T}' <: \mathtt{ref}\ \mathtt{T}$ *iff* $\mathtt{T}' \simeq \mathtt{T}$

Subclasses

Support incremental modification of code.

```
class Point
   var
      x = 0: Int;
   methods
      fun getx():int {return x}
      proc move(dx: int) {x := x + dx}
end class;

subclass Colorpoint of Point modifying move
   var
      color = blue: ClrType
   methods
      fun getColor():ClrType {return color}
      proc setColor(nuColor: ClrType)
                         {color := nuColor}
      proc move(dx: int) {super.move(dx);
                          color := red}
end class;
```

C++, Object Pascal, & Modula-3 type-checking too restricted!

Simple type systems don't allow changes in method types in subclasses.

```
class C
      ...
   methods
      Clone(): CType {...};
   end class

subclass SC of C redefining Clone
      ...
   methods
      Clone(): SCType {...};  --illegal!
   end class
```

Object Pascal: `Clone` returns type `Top`.

Must perform type-casts to use it!

Another example

Circle class with method returning center point.

```
class Circle
   var
      ...
   methods
      getCenter(): PointType {...}
      ...
   end class
```

Color Circle should return color point.

```
subclass ColorCircle of Circle
      redefining getCenter
      ...
   methods
      getCenter(): ColorPtType {...}
                                --illegal!
      ...
   end class
```

Binary Methods

```
class Node
   var
      val = 0: Int;
      next = nil: NodeType
   methods
      fun getVal(): Int {return val}
      proc setVal(nuVal: Int) {val := nuVal}
      fun getNext(): NodeType {return next}
      proc setNext(nuNext: NodeType)
                                 {next := nuNext}
end class;

subclass DbleNode of Node
   var
      prev = nil: DbleNodeType
   methods
      fun getPrev(): DbleNodeType {...}
      ...
end class;
```

What about types of `next`, `getNext`, and `setNext`?

More flexible subclasses

Why restrict changing types in subclasses?

Methods can be mutually recursive!

```
class Example
      :
methods
    proc m(...) {... n(s) ...}
    fun n(x: S): T {...}
end class;

subclass SubExample modifying n
methods
    fun n(x: S’): T’ {...}
    proc newMeth(...) {...}
end class;
```

What is relationship of new type of **n** to old?

Changing Types in Subclasses

Subtype will always be fine!

I.e., $\mathtt{S} <: \mathtt{S}'$ and $\mathtt{T}' <: \mathtt{T}$

ColorCircle's **getCenter** can return **ColorPoint**.

Can change return type of **Clone**
in subclass to type of objects from class.

If subclass updates method types so they
are subtypes of original then type-safe.

Subclasses always generate subtypes.

What about instance variables?

Instance variables can be values and receivers
— *No subtypes*!

If `Circle` has instance variable `center: Point`, `ColorCircle`'s `center` must have same type.

Hard to redefine `getCenter` in `ColorCircle`.

`Clone` example:

How to inherit `Clone` method if unchanged?

Must rewrite just to change return type!

`Node` example:

Must change type of `next`!!

Must change parameter type of `setNext`!!

Now what?

In `Clone` and `Node` needed type to change *automatically* to type of receiver.

Let `MyType` represent type of `self` in method.

```
class Node
   var
      val = 0: Int;
      next = nil: MyType
   methods
      fun getVal(): Int {return val}
      proc setVal(nuVal: Int) {val := nuVal}
      fun getNext(): MyType {return next}
      proc setNext(nuNext: MyType)
                              {next := nuNext}
      proc attachRight(nuNext: MyType)
                              {setNext(nuNext)}
end class;
```

If `n: NodeType` then `n.getNext: NodeType`.

What happens with subclass?

```
class DbleNode subclass of Node
      modifying attachRight
   var
      prev = nil: MyType
   methods
      fun getPrev(): MyType {return prev}
      proc setPrev(nuPrev: MyType)
                       {prev := nuPrev}
      proc attachRight(nuNext: MyType)
                       {setNext(nuNext);
                        nuNext.setPrev(self)}
end class;
```

If `dn: DbleNodeType` then
      `dn.getNext: DbleNodeType`.

`MyType` meaning flexible in subclasses,
      but fixed in objects.

Also solves problems with `Clone`!

Type-checking with `MyType`?

Can inherited methods break in subclasses?

Yes, unless careful about assumptions
on value of **`MyType`**!

Define *matching* (written $<\#$) on object types.

$$\mathtt{ObjType}\{\mathtt{m_j} : \mathtt{S_j}\}_{1 \leq j \leq m} <\# \, \mathtt{ObjType}\{\mathtt{m_i} : \mathtt{T_i}\}_{1 \leq i \leq n}$$

iff

$n \leq m$ and for each $1 \leq i \leq n$, $\mathtt{S_i} <: \mathtt{T_i}$.

If `SC` is subclass of `C` then `SCType` $<\#$ `CType`.

Anticipating Changes in MyType

How to type-check methods if only know
`self`: `MyType`?

If `C` generates objects of type `T`, then
type-check methods assuming $\texttt{MyType} <\# \, \texttt{T}$.

Objects generated by subclasses match.

If $\texttt{T'} = \texttt{ObjType}\{\ldots\texttt{;m:S';}\ldots\}$ and $\texttt{T}' <\# \, \texttt{T}$
then (*by def*) $\texttt{S}' <: \texttt{S}$

If $\texttt{o} : \texttt{T}' <\# \, \texttt{T}$
then $\texttt{o.m} : \texttt{S}'[\texttt{T}'/\texttt{MyType}] <: \texttt{S}[\texttt{T}'/\texttt{MyType}]$.

Thus if $\texttt{o} : \texttt{T}' <\# \, \texttt{ObjType}\{\ldots; \texttt{m} : \texttt{S}; \ldots\}$ then
`o.m: S[T'/MyType]`.

Same reasoning holds for `self`:
$\texttt{self.m: S[MyType/MyType]} = \texttt{S}$

But is this type-safe?

Changed parameter types *covariantly*!

Subclasses are no longer subtypes.

```
procedure breakit(n1, n2: NodeType)
begin
   n1.attachRight(n2)
end
```

Call of `breakit(nd1,nd2)` OK
if `nd1` & `nd2` both nodes.

Call of `breakit(dble_nd,nd)` *not* OK
if `dble_nd` is doubly-linked & `nd` is node!

Results in a call of `dble_nd.attachRight(nd)`.

$\boxed{\text{BANG!}}$

`DoubleNodeType` *not* subtype of `NodeType`!

Subtypes of Object Types: Coping with MyType

$$\texttt{ObjType}\{\texttt{m}_\texttt{j} : \texttt{S}_\texttt{j}\}_{1 \leq j \leq m} <: \texttt{ObjType}\{\texttt{m}_\texttt{i} : \texttt{T}_\texttt{i}\}_{1 \leq i \leq n}$$

if

1. $n \leq m$ and for each $1 \leq i \leq n$, $\texttt{S}_\texttt{i} <: \texttt{T}_\texttt{i}$.
2. No $\texttt{T}_\texttt{i}$ has a contravariant **MyType**.
   Thus, no **MyType** parameters in supertype.

Only difference between *subtyping* and *matching* is the second clause.

If two object types are subtypes, they match, but not vice-versa.

DoubleNodeType $<\#$ NodeType but *not* subtype because contravariant occurrence of **MyType**.

Constrained Genericity

Polymorphic structures: `List[T]`, `Tree[T]`, etc.

Need constraints on type parameters:

  `BinSrchTree[T]` requires ordered T.

$\texttt{Compare} = \texttt{ObjType}\{\texttt{lt},\ \texttt{eq},\ \texttt{gt} : \ \texttt{MyType} \rightarrow \texttt{Bool}\}$.

Try $\texttt{T} <: \texttt{Compare}$.

Won't work!

  Contravariant `MyType` $\Rightarrow$ no subtypes.

Need $\texttt{T} <\# \texttt{Compare}$.

Binary Search Tree

```
class BTreeNode(T <# Compare; v: T)
   var
      val = v: T;
      l, r = nil: MyType
   methods
      fun getVal(): T {return val}
      proc setVal(nuVal: T) {val := nuVal}
      fun getL(): MyType {return l}
      proc setL(newL: MyType) {l := newL}
      fun getR(): MyType {return r}
      proc setR(newR: MyType) {r := newR}
end class;

class BinSearchTree(T <# Compare)
   var
      root = nil: BTreeNodeType(T);
   methods
      fun find(elt: T): Bool; {...}
      proc insert(nuElt: T);
        {if root.getVal().lt(nuElt) then ...}
      fun empty?():Bool; {return root = nil}
end class;
```

Arguments for Covariance

Eiffel: *Covariant* changes necessary
in instance variables and parameters.

```
class Animal
   var
      ...
   methods
      proc eat(meal: FoodType) {...}
      ...
end class;

class Herbivore subclass of Animal
      modifying eat
   var
      ...
   methods
      proc eat(meal: PlantType) {...}
                 -- illegal change of type
      ...
end class;
```

Providing Covariance using Bounded Matching

```
class PAnimal(MyFoodType <# FoodType)
      ...
   methods
      proc eat(meal: MyFoodType) {...}
      ...
end class;

class PHerbivore(MyFoodType <# PlantType)
      subclass of PAnimal(MyFoodType)
            modifying eat
      ...
   methods
      proc eat(meal: MyFoodType) {...}
              -- Same type as in PAnimal!
      ...
end class;
```

Like `MyType` provided by “hand”.

Same idea solves problem
with `center` of `Circle`, `ColorCircle`.

Are We Done?

Solved all of typing problems!

Type system like PolyTOIL (*ECOOP '95*).

But did we need all of this?

Matching and subtyping too similar!

Final solutions didn't use subtyping!

What happens to subtype polymorphism
if drop subtyping?

```
proc p(w: Window){...}
```

Replace by `proc p(W <# Window; w : W){...}`.

or better: `proc p(w: #Window){...}`.

Treat #-types as abbreviations.

Extending #-types

What about assignments to variables?

```
aWindow := someWindow
```

Without subtyping, types must match exactly!

Create new declaration:

```
var aWindow: #Window
```

Now: $\#\mathtt{T} \quad \widehat{=} \quad \exists \mathtt{t} <\!\#\, \mathtt{T.t}$

What about *heterogeneous* data structures?

`List(Window)` *vs.* `List(#Window)`

Finer control over *homogeneous*
versus *heterogeneous* data structures.

Introducing LOOM

Supports `MyType` and bounded matching.

No subtyping.

No *explicit* changes allowed to method types in subclasses.

Addition of #-types supports flexible types.

<u>Drawback</u>:
Can't send binary message to #-type.

If `n: #NodeType` then `n.getNext()` returns value of type `#NodeType`.

If `n1, n2: #NodeType` then
`n1.setNext(n2)` *illegal*.

Can traverse, but not update!

Use bounded matching if need binary methods.

Summary

Two new *provably* type-safe OOLs:
PolyTOIL and LOOM.

PolyTOIL full-featured:

- Classes and objects: **`self`** and **`MyType`**.
- Subtype and Subclass
- Bounded matching
- Modular type-checking

LOOM similar, but no subtyping.

Can get *extended* LOOM by defining
matching on other types.

What does this have to do with semantics?

Design of safe type systems dependent on knowledge of semantics of typed OOL's:

Cardelli, Mitchell, Cook et al, ...

Many more recent contributors.

Safety of type rules proven via subject reduction based on natural semantics.

Don't believe possible to design *safe* type systems for OOL's without semantic basis!

Full paper available via: `http://cs.williams.edu/~kim`.