# Problemas de Tipos en Lenguajes Orientados a Objetos

## Teoría de Pruebas para Lenguajes de Programación

Profesor Eduardo Bonelli

Nicolás Passerini

Facultad de Ciencias Exactas – Universidad de Buenos Aires

12 de julio de 2012

# Overview

# Objetos y mensajes

- Los objetos que encapsulan
  - *estado* = colección de variables de instancia
  - y *comportamiento* = colección de métodos
    - subrutinas pueden manipular a las variables de instancia
    - tienen efecto.
- Cuando se envía un mensaje a un objeto, se ejecuta el método correspondiente.
- La palabra clave self referencia al objeto que está ejecutando el método.
- Todos los objetos son referencias: la asignación tiene semántica de sharing.
  - `o := p` implica que `o = p`.
  - `o = p` sii *referencian al mismo objeto*.

# Clases

- Las clases son templates extensibles para crear objetos.
- Todo objeto se crea a partir de una clase y el operador new.

```
o := C new
```

  El objeto `o` es instancia de la clase `C`.
- Los *instancias* de una clase:
  - Comparten *los mismos* métodos.
  - Cada uno tiene su propia copia de las variables de instancia.
- Los ejemplos de hoy se concentran lenguajes class-based.

# Ejemplo 1

- Tomamos el lenguaje $SOOL$, definido por [Bru02]

```
class CellClass {
  x: Integer := 0;

  function get(): Integer is
    { return self.x }

  function set(nuVal: Integer): Void is
    { self.x := nuVal }

  function bump(): Void is
    { self <= set(self <= get() + 1) }
}
```

# Subclases

- Permiten definir una clase nueva agregando o modificando métodos y variables de instancia a una superclase.

```
class ClrCellClass inherits CellClass modifies set {
  color: ColorType := blue;

  function getColor(): ColorType is
    { return self.color }

  function set(nuVal: Integer): Void is
    { super <= set(nuVal); self.color := red }
}
```

- El método `set`:
  - Sobreescribe (*overrides*) el `set` de la superclase.
  - `super` permite acceder al método sobreescrito.

# Tipos

- Consideramos un tipo como una abstracción que representa:
  - un conjunto de valores
  - operaciones y relaciones aplicables a esos valores
- Distinguimos las clases de los tipos de los objetos
  - Se diferencia de muchos lenguajes OO.
  - Permite un mayor nivel de abstracción.
  - El tipo sólo contiene los nombres y tipos (firmas) de los mensajes.
- Se agrega una construcción para representar los tipos de los objetos:

```
CellType = ObjectType {
  get: Void -> Integer;
  set: Integer -> Void;
  bump: Void -> Void
}
```

# Subtipos

- Indicamos `T <: U` si un valor de tipo T puede ser utilizado en cualquier contexto en que se espera un valor de tipo U.
- El subtipado es estructural.
- También la relación entre clase y tipo.
- No depende de la herencia: `ClrCellType <: CellType`

```
ClrCellType = ObjectType {
  get: Void -> Integer;
  set: Integer -> Void;
  bump: Void -> Void;
  getColor: Void -> ColorType
}
```

- Vamos a estudiar diferentes esquemas de subtipado.
- Algunas extensiones al lenguaje pueden hacer que una subclase no genere un subtipo.

# Polimorfismo de subtipos

- Un objeto tiene muchos tipos.
- Por ejemplo las instancias de `ClrCellClass` tienen los tipos `ClrCellType` y `CellType`.
- Subsumption rule:

$$\frac{\Gamma \vdash M : S' \quad S' <: S}{\Gamma \vdash M : S}$$

- Eso me permite cosas como:

```
c1:CellType := new ClrCellClass
```

- La subsumption conlleva una pérdida de información:

```
c1 <= setColor(green) // Imposible!!!
```

# Dynamic Method Invocation

- El método invocado depende del receptor.
- Ejemplo

```
c1:CellType := new ClrCellClass
c1 <= bump() // Ejecuta el bump de CellClass

class CellClass { ...
  function bump(): Void is {
    self <= set(...) // Depende del receptor
  }
}

class ClrCellClass ...
  function set(nuVal: Integer): Void is
    { super <= set(nuVal); self.color := red }
}
```

- Incluye los mensajes a `self`
- Dentro de `set`, la variable `self` tiene tipo `ClrCellType`.

# Problemas con subtipos en objetos

- Cuadrados vs. rectángulos.
- Si entendemos herencia como **"es un"**, un cuadrado "es un" rectángulo
- Por lo tanto cuadrado debería ser un subtipo de rectángulo.
- Sin embargo, los rectángulos podrían tener métodos que son inadecuados para un cuadrado.

```
Rectangle = ObjectType {
  height: Void -> Integer
  width: Void -> Integer
  stretch: (Integer × Integer) -> Void // Problema
}
```

- Este problema se magnifica si colapsamos clases y tipos.
- **Programa: Estudiar la relación entre subclases y subtipos.**

# Cast

- La regla de subsumption conlleva una pérdida de información.
- Algunos lenguajes permiten agregar información en forma manual mediante un cast

```
c2 := (ClrCellClass) c1;
c2 <= setColor(green)
```

- Se posterga el chequeo de tipos al tiempo de ejecución.
- Es obviamente algo no deseado.
- No se debe confundir con una coerción.
  - Java tiene coerciones sólo para los tipos básicos
  - Utiliza la misma sintaxis para ambas cosas.

# Sobrecarga

Un nombre de método está sobrecargado en un contexto si:

- Es utilizado para representar dos o más métodos *distintos*.
- El método representado es determinado por el tipo o firma (*signature*) del método.

```
class C {
  function equals(other:CType) : Boolean is { writeln (1) }
}

class SC {
  function equals(other:CType) : Boolean is { writeln (2) }
  function equals(other:SCType) : Boolean is { writeln (3) }
}

CType = ObjectType { equals: CType -> Boolean }
SCType = ObjectType {
  equals: CType -> Boolean;
  equals: SCType -> Boolean
}
```

# Sobrecarga - Ejemplo

¿Qué método se invoca en cada caso?

```
c1: CType := new C;
c2: CType := new SC;
sc: SCType := new SC;

c1 <= equals(c1);
c1 <= equals(c2);
c1 <= equals(sc);

c2 <= equals(c1);
c2 <= equals(c2);
c2 <= equals(sc);

sc <= equals(c1);
sc <= equals(c2);
sc <= equals(sc);
```

# Multimethods

- El método a ejecutar depende de los valores de uno o más de los parámetros del método.
- [Bru02] los presenta con una sintaxis procedural:

```
function equal(p1:Point, p2:Point): Boolean is { ... }
function equal(p1:ColorPoint, p2:ColorPoint): Boolean is
    { ... }
```

- Otros lenguajes mantienen la sintaxis que destaca a uno de los parámetros como *receptor*.
- Suele confundirse con la sobrecarga.

# Inmutable Records

- Breadth subtyping: less fields with same types
- Depth subtyping: same fields but each field is subtype of the corresponding field in the record supertype
- General record subtyping rule

$$\frac{k \leq n \quad \text{for all } 1 \leq i \leq k, T_i <: U_i}{\{\!|l_j : T_j|\!\}_{1 \leq j \leq n} <: \{\!|l_i : U_i|\!\}_{1 \leq i \leq k}}$$

- Subtyping for record fields is covariant.

# Function types

$$\frac{S <: S' \quad T' <: T}{S' \to T' <: S \to T}$$

- Subtyping for result types is covariant.
- Subtyping for parameter types is contravariant.

# Reference types

- References

$$\frac{\Gamma \vdash N : \mathsf{Ref}\ T \quad \Gamma \vdash M : T}{\Gamma \vdash N := M : \mathsf{Void}} \qquad \frac{\Gamma \vdash M : \mathsf{Ref}\ T}{\Gamma \vdash \mathit{val}\ M : T}$$

- Reference types are invariant

$$\frac{\mathsf{S'} \simeq \mathsf{S}}{\mathsf{Ref}\ S <: \mathsf{Ref}\ S'} \qquad S' \simeq S \text{ iff } S <: S' \text{ and } S' <: S$$

- Think of two record types with the same components in different order.
- There are no non-trivial subtypes of Ref $S$.

# Arrays

- Read only arrays are variant.

$$\frac{S' <: S}{\mathrm{ROArray}[S'] <: \mathrm{ROArray}[S]}$$

- Updatable arrays are invariant.

$$\frac{S' \simeq S}{\mathrm{Array}[S'] <: \mathrm{Array}[S]}$$

- The same holds for updatable records.
- Java 1.4 arrays are unsafe!

# Objetos

- Recordamos los tipos de los objetos:

```
CellType = ObjectType {
  get: Void -> Integer;
  set: Integer -> Void;
  bump: Void -> Void
}
```

- Es un registro de funciones
  $\Longrightarrow$ podríamos usar la misma regla.
- Si consideráramos también **variables**, deberían ser **exactamente del mismo tipo**.
- La visibilidad puede introducir más sutilezas.

# Sistema de tipos invariante

- La forma más tradicional de tipado en lenguajes OO.
- Requiere que un método sobreescrito tenga **exactamente el mismo tipo** que en la superclase.
- Siempre una subclase genera un subtipo.
- Son muy restrictivos.
- Ejemplos:
  - C++
  - Object Pascal
  - Java (hasta 1.4)

# Problemas I - Clone

- ¿Cuál sería el tipo de `clone`?
- Imaginemos este método en `ClrCellClass`

```
funtion m() : Void is
  { ...
    v := self <= clone();

    v.getColor() // problema
    ...
  }
```

- Definido en `Object` debería tener tipo `ObjectType`
- Se pierde mucha información de tipos.
- No puede ser utilizado sin un cast.
- Hay otros casos en los que quiero cambiar el tipo de retorno de un método.

# Problemas II - Métodos Binarios

- ¿Qué tipo debería tener `getColor` en `ClrCellClass`?

```
class CellClass {
  ...
  function equals(other:CellType): Boolean is
    { return this <= get() = other <= get() }
}

class ClrCellClass {
  ...
  function equals(other:???): Boolean is
    { return super <= get() &&
             this <= getColor() = other <= getColor) }
}
```

- Opciones:
  - Agregar un *casteo*
  - Cambiar el tipo de `other` y utilizar sobrecarga en lugar de sobreescritura $\Longrightarrow$ Se pierde el polimorfismo.

# Method specialization

- Permite modificaciones sobre:
  - los tipos de los **parámetros**,
  - el tipo de **retorno**.
- Sigue las mismas reglas de varianza que para las funciones.
- Clases aún generan subtipos.
- Ejemplos:
  - Java 1.5 permite especializar el valor de retorno
  - Eiffel permite modificar los parámetros en forma covariante $\Longrightarrow$ potenciales errores de tipos.
- **Problema:** Convivencia con otras características como sobrecarga o multimethods.

# Self type

- La keyword `Self` representa el tipo de `self`
- Es especializado automáticamente en las subclases.

```
class CellClass {
  function double() : Self is
    { return new CellClass(self.x * 2) }
}
class ClrCellClass {
  function double() : Self is
    { return new ClrCellClass(self.x * 2, self.color) }
}
```

- Usado como valor de retorno permite que subclases generen subtipos.
- En [AC96] también se permite para atributos.
- Eiffel lo permite en posición contravariante.

# Self en posición contravariante

- Las subclases dejan de generar subtipos.

```
class CellClass { ...
  function max(other:Self): Self is { ... }
}

CellType = ObjectType { ...
  max: CellType -> CellType
}
ClrCellType = ObjectType { ...
  max: ClrCellType -> ClrCellType
}

c1: CellType = new ClrCellClass(...)
c1.max(new CellType(...))
```

- Intentos de solución en Eiffel:
  - Dataflow analysis
  - No polymorphic catcalls (*Changing Availability or Type*)
  - Que las clases no generen subtipos.
- Idea: matching.

# Otras extensiones

- Tipos paramétricos
- Bound polymorphism
- F-Bound polymorphism [CCH+89]
- Mixins

# Object Protocols

- Tomado de [AC96]

```
CellType = ObjectType { ...
  max: CellType -> CellType
}

CellProtocol[X] = ObjectOperator { ...
  max: X -> X
}
```

- Notar que:

```
CellType <: CellProtocol[CellType]
```

- Para todo tipo recursivo T se puede abstraer un protocolo T-Protocol.

# Relaciones entre protocolos

- Definimos $\preceq$ como la *higher order subtype relation*:

$$P_1 \preceq P_2 \text{ iff } P_1[T] <: P_2[T] \text{ for all types } T$$

- Ejemplo:

```
CellProtocol[X] = ObjectOperator { ...
  max: X -> X
}
ClrCellProtocol[X] = ObjectOperator { ...
  max: X -> X,
  getColor: Color
}
```

# Subprotocol

- La relación **S subprotocol T** se puede definir de dos maneras:
  - `S <: T-Protocol[S]`
  - `S-Protocol` $\preceq$ `T-Protocol`

# Matching

- F-Bounded interpretation

$$\texttt{S <\# T} \quad \text{if} \quad \texttt{S <: T-Protocol[S]}$$

- Higher order interpretation

$$\texttt{S <\# T} \quad \text{if} \quad \texttt{S} \preceq \texttt{T-Protocol[S]}$$

- Con ambas definiciones tenemos `ClrCellType <# CellType`

# Extensión de Bound Polymorphism

- Siempre que tengamos una propiedad común a muchos tipos, podemos pensar en abstraer sobre los tipos que compartan esa propiedad.
- Ejemplos:
  - `P1[X <: CellProtocol[X]]`
  - `P2[P` $\preceq$ `CellProtocol]`
  - `P3[X <# CellType]`

# Bibliografía



Martín Abadi and Luca Cardelli.
*A theory of objects*.
Springer, 1996.


K.B. Bruce.
*Foundations of Object-Oriented Languages: Types and Semantics*.
Mit Press, 2002.


Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell.
F-bounded polymorphism for object-oriented programming.
In *FPCA*, pages 273–280, 1989.


Benjamin C. Pierce.
*Types and programming languages*.
MIT Press, 2002.