Continuing with Cardelli and Wegner

Amer Diwan

Outline

• What we know
  – Different kinds of polymorphism
  – Typed lambda calculus
• Next
  – Adding universal quantification to get parametric polymorphism
  – Adding existential quantification to get data abstraction
  – Adding bounded quantification to get inclusion polymorphism
Universal quantification and parametric polymorphism

• value id = all[a] fun(x: a) x
  \(\forall a. a \rightarrow a\)
  works regardless of the type of ‘x’ (with some representation tricks…)
• Use: id[Int](3)
• Type of ‘id’ is universally quantified

A bigger example

• List =
  \(\forall\text{Item}. \ [\text{nil: Unit} \ (\text{similar to NIL}),\)
  \text{cons\textunderscore cell}: \{\text{head: Item, tail: List[Item]}\}]\)
• Properties of ‘List’?
• value cons =
  all[Item] fun(h: Item, t: List[Item])
  \[\text{cons\textunderscore cell} = \{\text{head} = h, \text{tail} = t\}\]
• Properties of ‘cons’?
More on universal quantification

• Can define functions that operate uniformly for all types
• E.g., reverse: $\forall$Item. List[Item] -> List[Item]
• reverse works for arrays of any element type
• It preserves parameter type: if you pass it a list of integers, you get back a list of integers
• Let’s not worry about implementation for now...

Existential quantification and data abstraction

• $(3,4)$: $\exists$ a. a x a
  For some type ‘a’, $(3,4)$ has type a x a
• $(3,4)$: $\exists$ a
  For some type ‘a’, $(3,4)$ has type a
• The second form does not reveal anything about the structure of $(3,4)$--information hiding
Visualizing existential and universal quantification

Universal quantification

Existential quantification

Existential quantification example

- value p = pack[a=Int in a × (a→Int)](3,succ) : ∃a. a × (a→Int)
- open p as x in (snd(x))(fst(x))
- What is hidden? What is not hidden?
Abstract data types

- **type** Point = Real x Real
- **type** PointPkgType1 =
  { makepoint: (Real x Real) -> Point,
    x_coord: Point -> Real,
    y_coord: Point -> Real }
- **value** point_pkg1 =
  { makepoint = fun(x: Real, y: Real) (x,y),
    x_coord = fun(p: Point) fst(p),
    y_coord = fun(p: Point) snd(p) }

Using the type

- val p = point_pkg1.makepoint(2.0, 3.0)
  val f = point_pkg1.x_coord(p)
  val s = point_pkg1.y_coord(p)
- Is this an ADT?
Packages

• type PointPkgType2 =
  \exists Point. \{ makepoint: (Real x Real)->Point,
  x_coord: Point ->Real,
  y_coord: Point ->Real \}

• value point_pkg2: PointPkgType2 =
  pack[Point = (Real x Real)
  in PointPkgType2] point-pkg1

• point_pkg2 is a “abstracted” version of
  point_pkg1: the representation of Point is hidden!

Using the ADT

• val p = point_pkg2.makepoint(2.0, 3.0)
  val f = point_pkg2.x_coord(p)
  val s = point_pkg2.y_coord(p)

• Same as before, except that now the type of ‘p’ is
  \exists a.a and not Real x Real
Combining universal and existential

• Universal quantification + Existential quantification => Parametric data abstraction

• Eg: a generic stack with an abstract implementation

Example

• type GenericAbstractStack =
  ∀ Stack. {emptystack: Stack,
  push(Item, Stack) ->Stack,
  pop: Stack->Stack,
  top: Stack->Item}

• A “better” design can somehow link Stack and Item
Bounded quantification

- Issue: type parameters have no constraints on them
- Why do we want constraints?

Bounded universal quantification

- $\forall[a : T] e$
- Inheritance and polymorphism in O-O languages…
- type Point = {x: Int, y: Int}
  value moveX = $\forall[P : \text{Point}] \text{fun}(p: P, dx: \text{Int})$
    p.x = p.x + dx; p
  moveX works for all subtypes of Point
Why need bounded universal polymorphism?

- Instead of
  \[
  \text{value moveX =all[P<: Point] fun(p: P, dx: Int)}
  \]
  \[
  p.x = p.x + dx; p
  \]
- Could write:
  \[
  \text{value moveX = fun(p: Point, dx: Int)}
  \]
  \[
  p.x = p.x + dx; p
  \]
Since p will accept any subtype of Point

Bounded existential quantification

- \text{type TirePublic} = \{\text{rating: Integer}\}
- \text{type Car} = \exists a<:\text{Tire}. \{\text{tires: Tire, color: Color}\}
- What does this mean?
- Have you seen this in any language?
Discussion topics

• Is this paper useful?
  – Is the model intuitive?
  – Is the model powerful enough to describe mechanisms in existing languages?
  – Is the model powerful enough to expose weaknesses in existing languages?
  – Is the model all you need to know to design a good type system?

Summary

• Describes a formal framework for talking about
  – Types
  – Polymorphism
  – Data abstraction
typed o-o languages

• Polymorphism in Modula-3, Java, and C++
  
  – Reading: , “Multiple inheritance in C++”