Types, data abstraction, and polymorphism

Amer Diwan

Goals of the paper

• Explain the concept of “type” using sets
• Use the description of type to describe and categorize
  – polymorphism
  – abstraction
• Categorize the polymorphism in existing languages
About the authors

• Luca Cardelli
  – One of the most influential type theorists ever
  – Has designed or participated in the design of many languages: SML, Modula-3, Obliq, ...
• Peter Wegner
  – A big name in object-orientation

Untyped domains and types

• Untyped universes = 1 type
  – Bit strings in computer memory
    • But organized as integers, characters, instructions, ...
  – Sets
    • But organized as sets of pairs, functions, ...
  – Lambda expressions
    • But organized as functions that return boolean, that return int, ...
• Even untyped universes of objects decompose naturally into sets with uniform behavior
From untyped to typed worlds

Objects naturally fall into groups but code can violate the groups
A type system is a suit of armor that “protects” the groups

What is polymorphism?

- Contrast with monomorphic:
  - Functions and their operands have a unique type
  - Every value and variable can be interpreted to be of one and only one type
- More directly
  - Functions work uniformly on a range of operand types
  - Some values and variables may have more than one type
Kinds of polymorphism in functions

• Universal
  – Executes same code for an infinite number of types
  – Parametric: type parameters
  – Inclusion: inclusion of types
• Ad hoc
  – Executes distinct code for each of a small set of types
  – Overloading
  – Coercion

Parametric polymorphism

• procedure sort [t] (a: array of t;
  leq: fun(a, b: t): bool): array of t
• iarray: array of integer;
  sort [integer] (iarray, int_compare)
• sarray: array of string;
  sort [string] (sarray, string_compare)
Parametric polymorphism: discussion

• Is qsort in C an example of parametric polymorphism?
  – void qsort(
    void *base, size_t nmemb, size_t size,
    int (*compar) (const void *, const void *));

• Are templates in C++ an example of parametric polymorphism?
  – template<class T> qsort(T *a)

Inclusion polymorphism

• procedure print(int i) ...
• v: [0..128]
  print(v)
• Subtyping leads to inclusion polymorphism
Overloading

- operator +
- \( i = 10 + 15 \)
- \( f = 10.0 + 15.0 \)
- \( s = \text{“hello ”} + \text{“world”} \)

Coercion

- operator +
- \( f = 3 + 4.0 \Rightarrow f = \text{(float)} 3 + 4.0 \)
Kind of polymorphism

• The kinds of polymorphism are not obviously disjoint
  – Inclusion polymorphism is a variant of parametric polymorphism
  – What kind of polymorphism is used here?
    • 3 + 4
      3.0 + 4
      3 + 4.0
      3.0 + 4.0

Types as sets

• Consider a universe, V, of all values
• Types are subsets of these values that have something in common
• What is subtyping?
Where we are, where are we headed to?

• We know
  – The different kinds of polymorphism
  – That types can be thought of as sets of values

• Next step
  – Describe a simple typed language
  – Describe how to get universal polymorphism in this language using
    • types are sets
    • quantification

Untyped lambda calculus

• value succ = fun(x) x + 1
• value twice = fun(f) fun(y) f(f(y))
• Note that functions are curried
Typed lambda calculus

• value addn = fun (i: Int) fun(x: Int) x + i
  Int->Int->Int
• value succ = fun (x: Int) (returns Int) x + 1, or
  value succ = addn 1
  Int -> Int
• value twice = fun(f: Int->Int) fun(y: Int) f(f(y)),
  (Int->Int)->Int->Int
• let a: T = M in N
  type of N
• Return type of function is often left out for brevity

Record and variant types in typed lambda calculus

• type ARecordType = {a: Int, b: Bool, c: String}
  value r: ARecordType = {a = 3, b = 3, c = “abcd”}
• type AVariantType = [a: Int, b: Bool, c: String]
  value v1 = [a = 3]
  value v2 = [b = true]
  value v3 = [c = “abcd”]
  variant types are tagged
Universal quantification and parametric polymorphism

• value id = all[a] fun(x: a) x
  \forall a. a->a
  works regardless of the type of ‘a’ (with some representation tricks…)
• Use: id[Int](3)
• [] are used for both variant records and type parameters.

A bigger example

• rec type List[Item] =
  [nil: Unit, cons: {head: Item, tail: List[Item]}]
• Properties of ‘List’?
• value cons =
  all[Item] fun(h: Item, t: List[Item])
    [cons = {head = h, tail = t}]
• Properties of ‘cons’?
Existential quantification and data abstraction

- Difference between
  \( \forall a. a \times (a \rightarrow \text{Int}) \) and
  \( \exists a. a \times (a \rightarrow \text{Int}) \)
- \( [(3, \text{succ}); ([1;2;3],\text{length})] : \text{List[}\exists a. a \times (a \rightarrow \text{Int})] \)

Visualizing existential and universal quantification

Universal quantification \( \text{Int} \)

Existential quantification
Existential quantification example

- value p = pack[a=Int in a × (a->Int)](3,succ) :
  \( \exists a. a \times (a\rightarrow\text{Int}) \)
- open p as x in (snd(x))(fst(x))

Packages

- type Point =
  \( \exists \text{Point}. \{ \text{makepoint} : (\text{Real x Real})\rightarrow\text{Point}, \text{x Coord} : \text{Point} \rightarrow \text{Real}, \text{y Coord} : \text{Point} \rightarrow \text{Real} \} \)
Combining universal and existential quantification

- Universal quantification +
  Existential quantification =>
  Parametric data abstraction

- Eg:
  a generic stack (stack of Int, stack of Bool, …)
  with an
  abstract implementation (list, array, …)

Example

- type GenericAbstractStack =
  ∀Item. ∃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

- \( \text{all}[a <: T] e \)
- Inheritance and polymorphism in O-O languages…
- \textbf{type Point} = \{ \text{x: Int, y: Int} \}

\begin{verbatim}
value moveX = all[P <: Point] fun(p: P, dx: Int)
  p.x = p.x + dx; p
moveX works for all subtypes of Point
\end{verbatim}
Bounded existential quantification

- $\exists a <: t. t'$
- 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

Next topic: polymorphism in statically typed o-o languages

• Polymorphism in Modula-3, Java, and C++
  – What it is and how it is implemented
  – Reading: Stroustrup, “Multiple inheritance in C++”