Types, data abstraction, and polymorphism

Amer Diwan

Goals of the paper

- Use “types as sets” to describe and categorize
  - polymorphism
  - abstraction
- The mechanisms in the paper are not hindered by implementation issues
  - They let us see the full power of polymorphism
  - Expose strengths and weaknesses of mechanisms in existing languages
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

- Ad hoc
  - Executes distinct code for each of a small set of types
  - Overloading or Coercion
- Universal
  - Executes same code for an infinite number of types
  - Parametric or Inclusion
Ad-hoc polymorphism*

- **Overloading**: Operator name has many implementations; correct implementation chosen at compile time
- **Coersion**: Automatically applied casts give the impression of polymorphism
- We won’t say much more about ad-hoc polymorphism in this topic

Inclusion polymorphism*

- **Uses inclusion amongst types**
- **Subtyping leads to inclusion polymorphism**
Parametric polymorphism*

- Type parameters
- Example: Id function, min function, sort
- Parametric polymorphic functions can be typechecked and compiled just once and used for all types

Parametric polymorphism: discussion*

- Is qsort in C an example of parametric polymorphism?
- Are templates in C++ an example of parametric polymorphism?
Which polymorphism is it?

- The kinds of polymorphism are not obviously disjoint
  - Inclusion polymorphism is a variant of parametric polymorphism (as we will see later)
  - What kind of polymorphism is used here?
    - $3 + 4$
    - $3.0 + 4$
    - $3 + 4.0$
    - $3.0 + 4.0$

Typed lambda calculus*

- The language we will use for exploring polymorphism
  - Similar in many ways to real languages (such as SML)!
- Key concepts:
  - Everything is a lambda expression
  - no assignments
  - first-class functions
- Let's fill in the types
Richer types*

- type ARecordType = {a: Int, b: Bool, c: String}
- type AVariantType = [a: Int, b: Bool, c: String]

Subtyping of records*

- \{a_1: t_1, \ldots, a_n: t_n, \ldots, a_m: t_m\} <: \{a_1: u_1, \ldots, a_n: u_n\}
- (The order of fields does not matter)
- Can you express single inheritance with it?
- Can you express multiple inheritance with it?
- Is this more powerful than the above?
- Can you express the id function with it?
Parametric polymorphism via universal quantification*

- value id = all[a] fun(x: a): t x
  \( \forall a. \ a \to a \)
  works regardless of the type of ‘x’ (with some representation tricks…)
- Use: id[Int](3)

Data abstraction via existential quantification*

- (3,4): \( \exists a. \ a \times a \)
  For some type ‘a’, (3,4) has type a x a
- (3,4): \( \exists a. \ a \)
  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

- **type**
- **value** \( p = \text{pack}[a=\text{Int} \text{ in } a \times (a->\text{Int})](3,\text{succ}) : \exists a. \ a \times (a->\text{Int}) \)
- **open** \( p \text{ as } x \text{ in } (\text{snd}(x))(\text{fst}(x)) \)
- **What is hidden? What is not hidden?**
Combining universal and existential quantification*

- Universal quantification + Existential quantification => Parametric data abstraction
- Eg:
  - a generic stack
  - with an abstract implementation

Bounded universal quantification*

- type Point = {x: real, y: real, copy: () -> Point}; type Circle = {x: real, y: real, radius: real, copy: () -> Circle}
- Want to write a routine that takes two points and returns the one with the greater x coordinate
- Can you do the same in your favorite O-O language?
Bounded existential quantification*

- type Token = {
  kind:[number: Int, identifier: String];
  pos: Int;
}

- type TokenManager = {
  getNextToken(): Token;
  pushBackToken(t: Token): String;
}

- What if TokenManager doesn’t want anyone to know that “Token” has a “pos”?

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: Parametric polymorphism in languages

- Standard ML: a language with parametric polymorphism
- Reading: SML tutorial