

# Types, data abstraction, and polymorphism

Amer Diwan

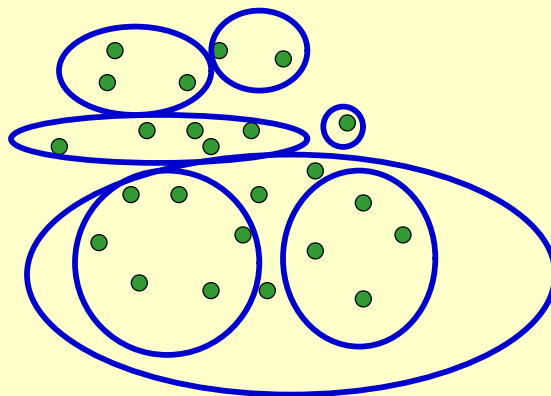
## Goals of the paper

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

## 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** or **Inclusion**
- Ad hoc
  - Executes distinct code for each of a small set of types
  - **Overloading** or **Coercion**

## Parametric polymorphism

- Type parameters
- fun 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)

## A simpler but fuller example

- fun min[t] (a: t;  
b: t;  
leq: fun(x: t, y: t): boolean): t =  
if leq(a, b) return a else return b
- Body of min executes the same code regardless of argument type
- Return type of min changes with argument type

## 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 \*));

## Parametric polymorphism discussion (cont.)

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

## Inclusion polymorphism

- Uses inclusion amongst types
- procedure print(int i) ...
- v: [0..128]  
print(v)
- Subtyping leads to inclusion polymorphism
- Other examples:

## Overloading

- Operator name has many implementations;  
correct implementation chosen at compile time
- operator +
- $i = 10 + 15$
- $f = 10.0 + 15.0$
- $s = \text{"hello"} + \text{"world"}$

## Coercion

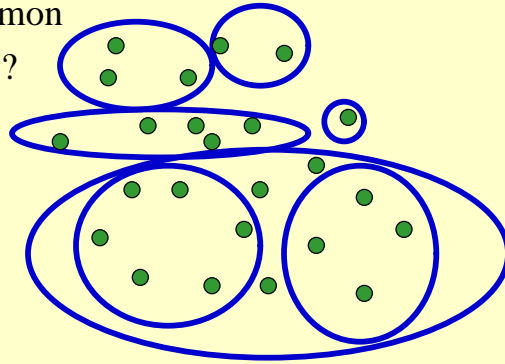
- Coersions give impression of polymorphism
- operator +
- $f = 3 + 4.0 \Rightarrow f = (\text{float}) 3 + 4.0$

## Kind of polymorphism

- The kinds of polymorphism are not obviously disjoint
  - Inclusion polymorphism is a variant of parametric polymorphism (as we will see later)
  - 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?



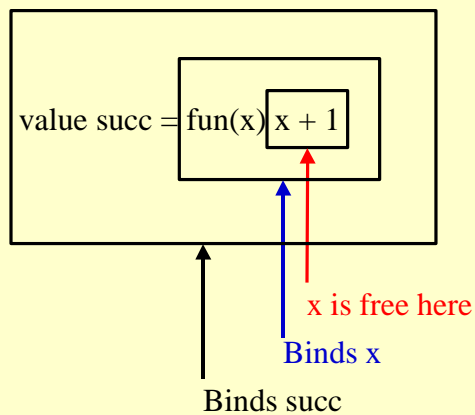
## Where we are, where are we headed to?

- **We know**
  - The different kinds of polymorphism
  - That types can be thought of as sets of values
- **Next step**
  - Describe a simple typed language
  - (next lecture) Describe how to get universal polymorphism in this language using
    - types are sets
    - quantification

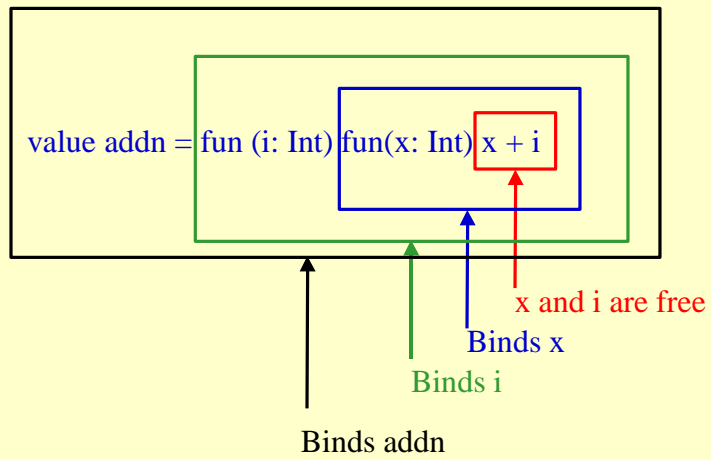
## Untyped lambda calculus

- Key concepts: functions and calls. no assignments
- value succ =  $\text{fun}(x)$   
 $x + 1$
- value addn =  $\text{fun } (i: \text{Int})$   
 $\text{fun}(x: \text{Int})$   
 $x + i$
- Let's analyze these functions in more detail

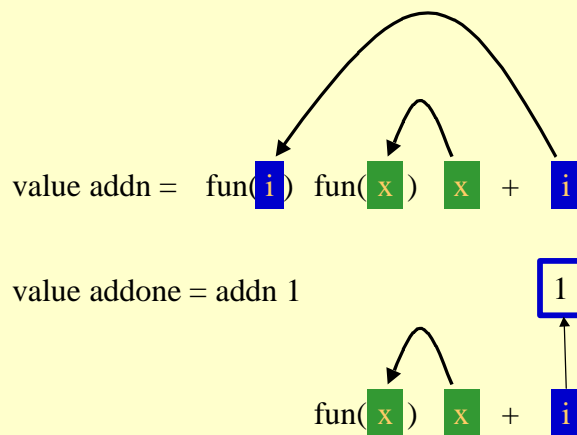
## Anatomy of a lambda expression



## A more elaborate example



## Another view of lambda expressions



## Take home messages

- To understand a lambda expression, break it down!
- To understand a call, copy the callee and plug in the actuals

## 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 = true, 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

## Let's talk about currying

- A **curried function**
  - take a single argument
  - return a single value of function type
- How do we write a function "curry" that
  - takes a function that takes a pair of arguments and
  - returns a function that takes one argument at a time?

## curry

- `val takepair(x,y) = E`
- Lets think about a function that curries things like `takepair`
  - Its type must be
    - `('a * 'b->'c) -> ('a->'b->'c)`
- By convention 'a etc. are type variables

`curry: ('a * 'b->'c) -> ('a->'b->'c)`

- Let's write what curry must look like
  - `val curry =`
    - `fun(f: 'a * 'b -> 'c)`
    - `fun(x: 'a)`
    - `fun(y: 'b) (returns 'c) ...`

## Curry's body

- When curried function gets its two arguments, it must produce the same result as the function being curried
  - $(\text{curry } f) \ i \ j \equiv f(i,j)$
- Putting them together:
  - `val curry =`  
`fun(f: 'a * 'b -> 'c)`  
`fun(x: 'a)`  
`fun(y: 'b) (returns 'c) f(x, y)`

## Why does curry work?

`val curry = fun(f) fun(x) fun(y) f(x, y)`

`val add = fun(x:Int, y: Int) x + y`  
`val cadd = curry add`

`val cadd =` `fun(x) fun(y) f(x, y)`

## Thoughts on cadd

- f is bound to the function being curried
- x and y are parameters that are accepted one at a time and eventually passed to f

## Let's play with curried functions

```
fun add x y = x + y
```

val add\_two = add 2

```
fun add_two y = x + y
```

↓  
2

## Next topic

- Sections 3, 4, and 5 of Cardelli and Wagner
- Topics: Universal and existential quantification for polymorphism and information hiding