Functional Programming: Functions and Types

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

September 12, 2006

Explicitly Parallel Fibonacci

C code

```
int fib (int n)
 {if (n < 2)
    return n;
  else
    return
    fib(n-1)+fib(n-2);
     }
 }
```

Cilk code

```
cilk int fib (int n)
 {if (n < 2)
      return n;
  else
    {int x, y;
      x = spawn fib(n-1);
      y = spawn fib(n-2);
      sync;
      return x + y;
    }
 }
```

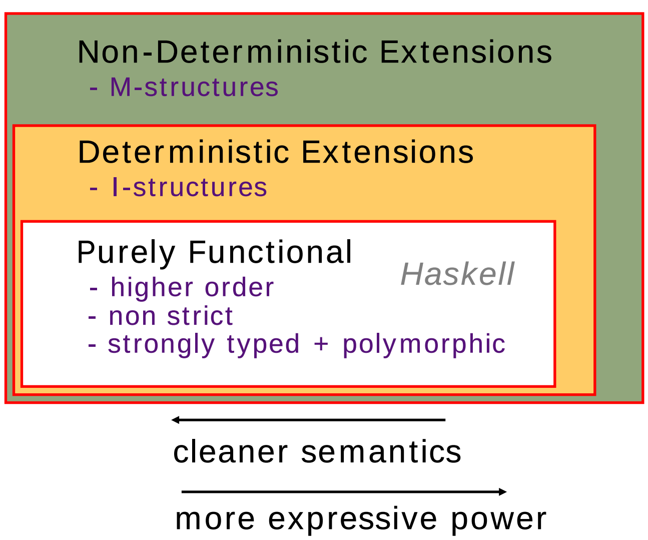
C dictates that fib(n-1) be executed before fib(n-2)
$\Rightarrow$ annotations (spawns and sync) for parallelism

Alternative: *declarative languages*

Why Declarative Programming?

- *Implicit Parallelism*
  - language only specifies a partial order on operations
- *Powerful programming idioms and* efficient code *reuse*
  - Clear and relatively small programs
- *Declarative language semantics have good algebraic properties*
  - *Compiler optimizations* go farther than in imperative languages

pH (parallel Haskell): An *Implicitly Parallel & Layered* Language



Function Execution by Substitution

```
      plus x  y = x + y
1.    plus  2  3  →  2 + 3  →  5
2.    plus  (2*3)  (plus 4 5)
```

```
→ plus 6 (4+5)
→ plus 6 9
→ 6 + 9
→ 15
```


Confluence

All Functional pH programs (right or wrong) have *repeatable behavior*

Blocks

```
let
    x =  a * a
    y =  b * b
in
    (x - y)/(x + y)
```

- a variable can have at most one definition in a block
- ordering of bindings does not matter

Layout Convention

This convention allows us to omit many delimiters

```
let
    x =  a * a
    y =  b * b
in
    (x - y)/(x + y)
```

is the same as

```
let
    { x =  a * a ;
      y =  b * b ;}
in
    (x - y)/(x + y)
```

Lexical Scoping

```
let
    y = 2 * 2
    x = 3 + 4
    z = let
           x = 5 * 5
           w = x + y * x
         in
            w
in
    x + y + z
```

Lexically closest definition of a variable prevails.

Renaming Bound Identifiers
(α-renaming)

```
let                               let
  y = 2 * 2                         y = 2 * 2
  x = 3 + 4                         x = 3 + 4
  z = let                           z = let
         x = 5 * 5        ≡                x' = 5 * 5
         w = x + y * x                     w = x' + y * x'
      in                                in
         w                                 w
in                                in
   x + y + z                         x + y + z
```

Lexical Scoping and α-renaming

```
plus  x y = x + y

plus' a b = a + b
```

plus and **plus'** are the same because **plus'** can be obtained by *systematic renaming of bound identifiers* of **plus**

Capture of Free Variables

```
f x = . . .
g x = . . .
foo f x = f (g x)
```

Suppose we rename the bound identifier **f** to **g** in the definition of **foo**

```
foo' g x = g (g x)
```

foo ≡ **foo'** ?

Curried functions

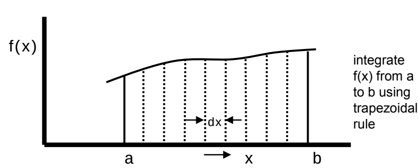
```
plus x  y =  x + y

let
    f  = plus 1
in
    f 3
```

→ (plus 1) 3 → 1 + 3 → 4

Local Function Definitions

Free variables of **sum** ?

```
integrate dx a b f =
     let
        sum x tot =
             if x > b then tot
             else sum (x+dx) (tot+(f x))
     in
        (sum (a+dx/2) 0) * dx
```



Integral(a,b) = (f(a + dx/2) + f(a + 3dx/2) + ...) dx

Any function definition can be "closed"

Loops (Tail Recursion)

- Loops or tail recursion is a restricted form of recursion but it is adequate to represent a large class of common programs.
  - Special syntax can make loops easier to read and write
  - Loops can often be implemented with greater efficiency

```
integrate dx a b f =
    let
        x = a + dx/2
        tot = 0
    in
        (while x <= b do
          next   x = x + dx
          next tot = tot + (f x)
        finally tot) * dx
```


Types

All expressions in pH have a type

23 :: Int

"**23** belongs to the set of integers"
"The type of **23** is **Int**"

true :: Bool
"hello" :: String

Type of an expression

```
(sq 529)  :: Int
 sq       :: Int -> Int
```

"**sq** is a function, which when applied to an integer produces an integer."

"**Int -> Int** is the set of functions which when applied to an integer produce an integer."

"The type of **sq** is **Int -> Int.**"

Type of a Curried Function

```
        plus x  y =  x + y

(plus 1) 3     :: Int

(plus 1)       :: Int -> Int

 plus          ::
```

?

λ-Abstraction

Lambda notation makes it explicit that a value can be a function. Thus,

(plus 1) can be written as **\y -> (1 + y)**

plus x y = x + y

can be written as

plus = \x -> \y -> (x + y)

or as

plus = \x y -> (x + y)

(In Haskell **\x** is a syntactic approximation of λ**x**)

Parentheses Convention

f e1 e2 ≡ ((f e1) e2)

f e1 e2 e3 ≡ (((f e1) e2) e3)

application is *left associative*

Int -> (Int -> Int) ≡ Int -> Int -> Int

type constructor **"->"** is *right associative*

Type of a Block

```
(let
    x1 =  e1
    .
    .
    .
    xn =  en
 in
     e )         ::   t
```

provided

$$e :: t$$

Type of a Conditional

(*if* e *then* e_1 *else* e_2) :: t

provided

e	**::**	**Bool**
e_1	**::**	**t**
e_2	**::**	**t**

The type of expressions in both branches of conditional must be the same.

Polymorphism

```
twice f x = f (f x)
```

1. **twice (plus 3) 4**
   → (Plus 3) ((plus 3) 4)
   → ((plus 3) 7)
   → 10
   twice :: ?
2. **twice (appendR "two") "Desmond"**

   twice :: ?

where **appendR "baz" "foo"** → **"foobaz"**

Deducing Types

twice f x = f (f x)
What is the most "general type" for twice?

1. Assign types to every subexpression
   x :: t0   **f :: t1**
   f x :: t2   **f (f x) :: t3**
   ⇒ **twice :: t1 -> (t0 -> t3)**
2. Set up the constraints
   t1 = t0 -> t2   because of **(f x)**
   t1 =   because of **f (f x)**
3. Resolve the constraints

Another Example: *Compose*

`compose f g x = f (g x)`
What is the type of **`compose`** **?**

1. Assign types to every subexpression

```
     x :: t0     f :: t1       g :: t2
   g x :: t3     f (g x) :: t4
⇒ compose :: t1 -> t2 -> t0 -> t4
```

2. Set up the constraints

`t1 = t3 -> t4`	because of **`f (g x)`**
`t2 = t0 -> t3`	because of **`(g x)`**

3. Resolve the constraints

⇒ **`compose ::`**

Now for some fun

`twice f x = f (f x)`

1. **twice$_1$ (twice$_2$ succ) 4**

twice$_1$ ::
twice$_2$ ::

same?

2. **twice$_3$ twice$_4$ succ 4**

twice$_3$ ::
twice$_4$ ::

same?

The first person with the right types gets a prize!

Hindley-Milner Type System

pH and most modern functional languages follow the Hindley-Milner type system.

The main source of polymorphism in this system is the *Let block*.

The type of a variable can be instantiated differently within its lexical scope.

much more on this later ...