Implicitly Parallel Multithreaded Programming

Arvind
Laboratory for Computer Science
MIT

Fibonacci Revisited

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)
⇒ annotations (spawns and sync) for parallelism

Alternative: *declarative languages*

Why Declarative Programming?

- *Implicit Parallelism*
  only specify a partial order on operations

- Clear and relatively *small programs*
  source code size ($\frac{\text{Fortran}}{\text{pH}}$) = 3 to 10

- *Compiler optimizations* go farther
  than in imperative languages

pH is *Implicitly Parallel* and a *Layered* Language

Non-Deterministic Extensions
- M-structures

Deterministic Extensions
- I-structures

Purely Functional
- higher order
- non strict
- strongly typed + polymorphic

← cleaner semantics

→ more expressive power

Function Execution by Substitution

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


Confluence $\Rightarrow$ ***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` $\equiv$ `foo'` ?

While renaming, entirely new names should be introduced!

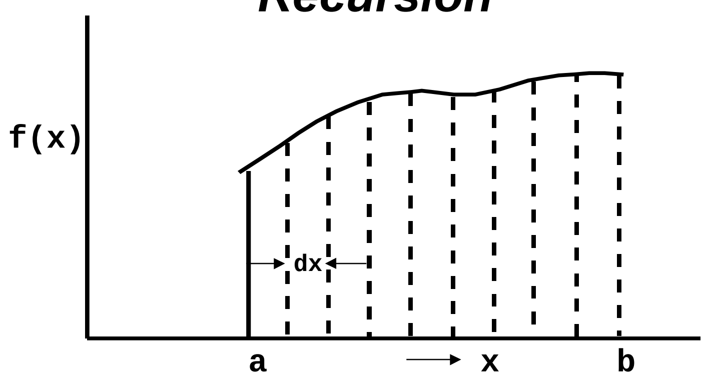
Curried functions

```
plus x  y =  x + y

let
     f  = plus 1
in
     f 3
```

$\rightarrow$ (plus 1) 3 $\rightarrow$ 1 + 3 $\rightarrow$ 4

Recursion



(f(a + dx/2) + f(a + 3dx/2) + ...) $*$ dx

```
integrate dx a b f =
  (sum dx b f (a+dx/2) 0) * dx

sum dx b f x tot =
  if x > b then tot
  else sum dx b f (x+dx) (tot+(f x))
```

Loops

Recursion is adequate to express all our computations. However, Loops are

- **sometimes more readable**
- **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
```


Local Function Definitions

Improve *modularity* and reduce clutter.

```
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
```

What are the free variables in the definition `sum`?

Each invocation of integrate defines a new `sum` function!

Higher-Order Computation Structures

```
apply_n f n x = if (n == 0) then x
                else apply_n f (n-1) (f x)

succ x =  x + 1

plus a b = apply_n succ b a

mult a b = apply_n                 ?
```

Syntactic spice: `x + y` ≡ `(+) x y`

thus `succ` can also be written as `((+) 1)`

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)
```

and

```
plus x  y =  x + y
```

can be written as

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

or as

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

(`\x` is a syntactic approximation of λx in Haskell)

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

$$
\begin{array}{l}
(\textit{let} \\
\quad x_1 = e_1 \\
\quad . \\
\quad . \\
\quad . \\
\quad x_n = e_n \\
\textit{in} \\
\quad e\ ) \qquad :: \quad t
\end{array}
$$

provided

$$e \quad :: \quad t$$

Type of a Conditional

$$(\textit{if}\ e\ \textit{then}\ e_1\ \textit{else}\ e_2\ )\ ::\ t$$

provided

$$
\begin{array}{lll}
e & :: & \mathtt{Bool} \\
e_1 & :: & t \\
e_2 & :: & t
\end{array}
$$

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`

   →

   `twice :: ?`

2. `twice (s_append "two") "Desmond"`

   →

   `twice :: ?`

where `s_append "foo" "baz"` → `"bazfoo"`

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 = t2 -> t3       because of f (f x)
   ```

3. **Resolve the constraints**
   ```
   t0 -> t2 = t2 -> t3
   ```
   ⇒ `t0 = t2 and t2 = t3` ⇒ `t0 = t2 = t3`
   ⇒ `twice :: (t0 -> t0) -> (t0 -> t0)`

Another Example: *Compose*

```
compose f g x = f (g x)
```

```
compose ::                          ?
```


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.

Generic vs. Non-generic type variables

Type Instances

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

twice :: (t -> t) -> (t -> t)
```

$\texttt{twice}_1$ $\texttt{twice}_2$ `(plus 3) 4`

$\texttt{twice}_1$ `::`

$\texttt{twice}_2$ `::`

Generic vs. Non-Generic

```
let
    twice f x = f (f x)
in
    twice twice (plus 3) 4
```

vs.

```
let
    twice f x = f (f x)
    foo g = (g g (plus 3)) 4
in
   foo twice
```

Only let-bound identifiers are polymorphic.