Implicitly Parallel Programming in pH: Functions and Types

Arvind
Laboratory for Computer Science
M.I.T.

September 9, 2002

http://www.csg.lcs.mit.edu/6.827

Explicitly Parallel Fibonacci

\[
\begin{align*}
\text{C code} & \quad \text{Cilk code} \\
\text{int fib (int n)} & \quad \text{cilk int fib (int n)} \\
\{ & \quad \{ \\
\text{if (n < 2)} & \quad \text{if (n < 2)} \\
\text{\quad return n; } & \quad \text{\quad return n; } \\
\text{\quad else } & \quad \text{\quad else} \\
\text{\quad \quad return } & \quad \text{\quad \quad return} \\
\text{\quad \quad fib(n-1)+fib(n-2); } & \quad \text{\quad \quad spawn fib(n-1); } \\
\text{\quad \quad } & \quad \text{\quad \quad spawn fib(n-2); } \\
\text{\quad \quad } & \quad \text{\quad \quad sync; } \\
\text{\quad \quad } & \quad \text{\quad \quad return x + y; } \\
\text{\} & \quad \text{\} }
\end{align*}
\]

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

Alternative: \textit{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 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

\[ \text{plus } x \ y = x + y \]

1. \[ \text{plus } 2 \ 3 \rightarrow 2 + 3 \rightarrow 5 \]
2. \[ \text{plus } (2*3) \ (\text{plus } 4 \ 5) \]

Confluence

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

\[
\text{let} \\
\quad x = a * a \\
\quad y = b * b \\
\text{in} \\
\quad (x - y)/(x + y)
\]

- 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

\[
\text{let} \\
\quad x = a * a \\
\quad y = b * b \\
\text{in} \\
\quad (x - y)/(x + y)
\]

is the same as

\[
\text{let} \\
\quad \{ x = a * a ; \\
\quad \quad y = b * b ; \} \\
\text{in} \\
\quad (x - y)/(x + y)
\]
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
  y = 2 * 2
  x = 3 + 4
  z = let
      x = 5 * 5
      w = x + y * x
      in
      w
  in
  x + y + z
```

```
let
  y = 2 * 2
  x = 3 + 4
  z = let
      x’ = 5 * 5
      w = x’ + y * x’
      in
      w
  in
  x + y + z
```
Lexical Scoping and $\alpha$-renaming

\begin{align*}
\text{plus} &\ x\ y = x + y \\
\text{plus}' &\ a\ b = a + b \\
\end{align*}

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

Capture of Free Variables

\begin{align*}
f &\ x = \ldots \\
g &\ x = \ldots \\
\text{foo } f\ x &\ = f\ (g\ x) \\
\end{align*}

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

\begin{align*}
\text{foo}' \ g\ x &\ = g\ (g\ x) \\
\text{foo } &\ \equiv \ \text{foo}' \\
\end{align*}

While renaming, entirely new names should be introduced!
Curried functions

\[
\text{plus } x \ y = x + y
\]

\[
\text{let } \quad f = \text{plus } 1 \\
\text{in } \quad f 3
\]
Local Function Definitions

Improve *modularity* and reduce clutter.

\[
\text{integrate } dx \ a \ b \ f = \\
(\text{sum } dx \ b \ f \ (a+dx/2) \ 0) \ast dx
\]

\[
\text{sum } dx \ b \ f \ x \ tot = \\
\text{if } x > b \ \text{then tot} \\
\text{else sum } dx \ b \ f \ (x+dx) \ (tot+(f \ x)) 
\]

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

\[
\text{integrate } dx \ a \ b \ f = \\
\text{let} \\
x = a + dx/2 \\
tot = 0 \\
in \\
(\text{while } x <= b \ \text{do} \\
\text{next } x = x + dx \\
\text{next } tot = tot + (f \ x) \\
\text{finally tot) } \ast dx
\]
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

apply_n succ b a

succ can be written as ((+) 1) also because of
the syntactic convention:

x + y ≡ (+) x y

apply_n ((+) 1) b a

mult a b = apply_n

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

\[(\text{sq } 529) :: \text{Int}\]
\[\text{sq} :: \text{Int} \rightarrow \text{Int}\]

"\text{sq} is a function, which when applied to an integer produces an integer."

"\text{Int} \rightarrow \text{Int} is the set of functions which when applied to an integer produce an integer."

"The type of \text{sq} is \text{Int} \rightarrow \text{Int}."

Type of a Curried Function

\[\text{plus } x \ y = x + y\]
\[(\text{plus } 1) \ 3 :: \text{Int}\]
\[(\text{plus } 1) :: \text{Int} \rightarrow \text{Int}\]
\[\text{plus} :: ?\]
\(\lambda\)-Abstraction

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

\((\text{plus 1})\) can be written as \(\lambda y \rightarrow (1 + y)\)

\[\text{plus } x \ y = x + y\]

can be written as

\[\text{plus } = \lambda x \rightarrow \lambda y \rightarrow (x + y)\]

or as

\[\text{plus } = \lambda x \ y \rightarrow (x + y)\]

(\(\lambda x\) is a syntactic approximation of \(\lambda x\) in Haskell)

Parentheses Convention

\[\text{f } e_1 \ e_2 \equiv ((\text{f } e_1) \ e_2)\]

\[\text{f } e_1 \ e_2 \ e_3 \equiv (((\text{f } e_1) \ e_2) \ e_3)\]

application is left associative

\[\text{Int } \rightarrow (\text{Int } \rightarrow \text{Int}) \equiv \text{Int } \rightarrow \text{Int } \rightarrow \text{Int}\]

type constructor “\(\rightarrow\)” is right associative
Type of a Block

\[
(let
\begin{align*}
x_1 &= e_1 \\
&\quad \vdots \\
&\quad \vdots \\
x_n &= e_n
\end{align*}
\text{in}
\begin{align*}
e
\end{align*}
) :: t
\]

provided
\[
e :: t
\]

Type of a Conditional

\[
(if \ e \ then \ e_1 \ else \ e_2 ) :: t
\]

provided
\[
\begin{align*}
e & :: \text{Bool} \\
e_1 & :: t \\
e_2 & :: t
\end{align*}
\]

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 (appendR "two") "Desmond"

twice :: ?

where appendR "baz" "foo" \(\rightarrow\) "foobaz"

Deducing Types

twice \( f \ x = f \ (f \ x) \)

What is the most "general type" for twice?

1. Assign types to every subexpression

\[
\begin{align*}
x &:: t_0 \\
f &:: t_1 \\
f \ x &:: t_2 \\
f \ (f \ x) &:: t_3 \\
\Rightarrow twice &:: t_1 \rightarrow (t_0 \rightarrow t_3)
\end{align*}
\]

2. Set up the constraints

\[
\begin{align*}
t_1 &= t_0 \rightarrow t_2 &\text{because of } (f \ x) \\
t_1 &= t_2 \rightarrow t_3 &\text{because of } f \ (f \ x)
\end{align*}
\]

3. Resolve the constraints

\[
\begin{align*}
t_0 \rightarrow t_2 &= t_2 \rightarrow t_3 \\
\Rightarrow t_0 &= t_2 &\text{and } t_2 = t_3 \\
\Rightarrow t_0 &= t_2 = t_3 \\
\Rightarrow twice &:: (t_0 \rightarrow t_0) \rightarrow (t_0 \rightarrow t_0)
\end{align*}
\]
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 ::` 

---

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* ...