Explicitly Parallel Fibonacci

C code

```c
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*
  - 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*

- **Non-Deterministic Extensions**
  - M-structures

- **Deterministic Extensions**
  - I-structures

- **Purely Functional** *(Haskell)*
  - 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. \[ \text{plus } (2\times3) \ (\text{plus } 4 \ 5) \rightarrow \text{plus } 6 \ (4+5) \rightarrow \text{plus } 6 \ 9 \rightarrow 6 + 9 \rightarrow 15 \]

Confluence

All Functional pH programs (right or wrong) have \textit{repeatable behavior}
Blocks

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

- 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

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

is the same as

\[
\begin{align*}
&\text{let} \\
&\quad \{ x = a \ast a ; \\
&\quad \quad y = b \ast b ; \} \\
&\text{in} \\
&\quad (x - y)/(x + y)
\end{align*}
\]
Lexical Scoping

\[
\begin{align*}
\text{let} \\
y &= 2 \times 2 \\
x &= 3 + 4 \\
z &= \text{let} \\
x &= 5 \times 5 \\
w &= x + y \times x \\
in \\
w \\
in \\
x + y + z
\end{align*}
\]

Lexically closest definition of a variable prevails.

Renaming Bound Identifiers
(\(\alpha\)-renaming)

\[
\begin{align*}
\text{let} \\
y &= 2 \times 2 \\
x &= 3 + 4 \\
z &= \text{let} \\
x &= 5 \times 5 \\
w &= x + y \times x \\
in \\
w \\
in \\
x + y + z
\end{align*}
\]

\[
\begin{align*}
\text{let} \\
y &= 2 \times 2 \\
x &= 3 + 4 \\
z &= \text{let} \\
x' &= 5 \times 5 \\
w &= x' + y \times x' \\
in \\
w \\
in \\
x + y + z
\end{align*}
\]
Lexical Scoping and $\alpha$-renaming

\[ \text{plus} \ x \ y = x + y \]
\[ \text{plus'} \ a \ b = a + b \]

\text{plus} and \text{plus'} are the same because \text{plus'}
can be obtained by \textit{systematic renaming of bound identifiers} of \text{plus}

Capture of Free Variables

\[ f \ x = \ldots \]
\[ g \ x = \ldots \]
\[ \text{foo} \ f \ x = f \ (g \ x) \]

Suppose we rename the bound identifier $f$ to $g$
in the definition of $\text{foo}$

\[ \text{foo'} \ g \ x = g \ (g \ x) \]

\[ \text{foo} \ \equiv \ \text{foo'} \ \ ? \]
Curried functions

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

\[
\begin{aligned}
\text{let} \\
\ f &= \text{plus } 1 \\
\text{in} \\
\ f \ 3 \\
\rightarrow \ (\text{plus } 1) \ 3 \rightarrow 1 + 3 \rightarrow 4
\end{aligned}
\]

Local Function Definitions

\[
\text{integrate } dx \ a \ b \ f = \\
\begin{aligned}
\text{let} \\
\sum x \ \text{tot} &= \\
\quad \text{if } x > b \text{ then } \text{tot} \\
\quad \text{else } \sum (x+dx) \ (\text{tot}+(f \ x)) \\
\text{in} \\
\ (\sum (a+dx/2) \ 0) \ * \ dx
\end{aligned}
\]

Free variables of sum?

\[
\text{Integral}(a,b) = \int\! f(x) \ dx = f(a) + \int\! f(a + 3dx/2) + \ldots \ dx
\]

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

```haskell
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

\[
\begin{align*}
(sq \ 529) & \quad :: \ Int \\
\text{sq} & \quad :: \ Int \to \ Int
\end{align*}
\]

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

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

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

Type of a Curried Function

\[
\begin{align*}
\text{plus} \ x \ y & = \ x + y \\
(\text{plus} \ 1) \ 3 & \quad :: \ Int \\
(\text{plus} \ 1) & \quad :: \ Int \to \ Int \\
\text{plus} & \quad :: \ ?
\end{align*}
\]
\[\lambda\text{-Abstraction}\]

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

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

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

---

\[\text{Parentheses Convention}\]

\(f \ e_1 \ e_2 \equiv ((f \ e_1) \ e_2)\)

\(f \ e_1 \ e_2 \ e_3 \equiv (((f \ e_1) \ e_2) \ e_3)\)

Application is \textit{left associative}

---

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

Type constructor “\(\rightarrow\)” is \textit{right associative}
Type of a Block

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

Type of a Conditional

\[
(\text{if } e \text{ then } e_1 \text{ else } e_2 ) :: t
\]
provided
\[
e :: \text{Bool} \\
e_1 :: t \\
e_2 :: t
\]
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
   \[ x :: t_0 \quad f :: t_1 \]
   \[ f \ x :: t_2 \quad f \ (f \ x) :: t_3 \]
   \[ \Rightarrow \quad \text{twice} :: t_1 \rightarrow (t_0 \rightarrow t_3) \]

2. Set up the constraints
   \[ t_1 = t_0 \rightarrow t_2 \quad \text{because of } (f \ x) \]
   \[ t_1 = \quad \text{because of } f \ (f \ x) \]

3. Resolve the constraints
Another Example: *Compose*

\[
\text{compose } f \ g \ x = f \ (g \ x)
\]

What is the type of `compose`?

1. Assign types to every subexpression
   
   \[
   \begin{align*}
   x &: t_0 & f &: t_1 & g &: t_2 \\
   g \ x &: t_3 & f \ (g \ x) &: t_4
   \end{align*}
   \]
   
   \[\Rightarrow \ \text{compose} &: t_1 \to t_2 \to t_0 \to t_4\]

2. Set up the constraints
   
   \[
   \begin{align*}
   t_1 &= t_3 \to t_4 & \text{ because of } f \ (g \ x) \\
   t_2 &= t_0 \to t_3 & \text{ because of } (g \ x)
   \end{align*}
   \]

3. Resolve the constraints
   
   \[\Rightarrow \ \text{compose} : : \]

---

Now for some fun

\[
\text{twice } f \ x = f \ (f \ x)
\]

1. `twice_1 (twice_2 \ succ) 4`

   \[
   \begin{align*}
   \text{twice}_1 &: : \\
   \text{twice}_2 &: :
   \end{align*}
   \]

   \[\text{same?}\]

2. `twice_3 twice_4 \ succ 4`

   \[
   \begin{align*}
   \text{twice}_3 &: : \\
   \text{twice}_4 &: :
   \end{align*}
   \]

   \[\text{same?}\]

   *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 ...*