More Lambda Calculus: Arithmetic and Fixed Points

April 8, 2002

CS 131: Programming Languages

Addition and Multiplication

- Find terms **plus** and **times** such that

  plus ⌜m⌝ ⌜n⌝ $\leftrightarrow_\beta^*$ ⌜m+n⌝

  times ⌜m⌝ ⌜n⌝ $\leftrightarrow_\beta^*$ ⌜mn⌝

Predecessor

- Tricky to subtract with Church numerals
- Key idea: consider the sequence

  <⌜0⌝,⌜0⌝**>** **<**⌜0⌝,⌜1⌝**>** **<**⌜1⌝,⌜2⌝**>** **<**⌜2⌝,⌜3⌝**>** ...

```
pred' := λn.(n (λp.<snd p,succ(snd p)>)
               <0,0>)

pred  := λn.(fst (pred' n))
```

Amazing Facts about Fixed Points

1. For *every* λ-calculus term M, there exists N such that

   $M(N) \leftrightarrow_\beta^* N$

   (including M=**not** and M=**succ** **!**)

2. A term N with this property is called a *fixed point* of M.

3. Fixed points can be found *uniformly*. There are l-calculus terms which compute these fixed points.
   - For example, there is a term **Y** such that

     $M(\mathbf{Y}(M)) \leftrightarrow_\beta^* \mathbf{Y}(M)$

     Namely,

     Y := λf.(λx.f(xx))(λx.f(xx))

Exercises

- Show that

  $\mathtt{M}(\mathbf{Y}(\mathtt{M})) \leftrightarrow_{\beta}^{*} \mathbf{Y}(\mathtt{M})$

Definitions

- The definitions we have seen so far have all been *abbreviations* or *macros*
- Convenient shorthand, letting us write

  not tt

  instead of

  $(\lambda b.\lambda x.\lambda y.b\ y\ x)(\lambda w.\lambda z.w).$
- But, we can always get rid of all abbreviations by replacing them with their definitions.

Recursive Definitions

- But what about defining recursive functions, like factorial?
- A definition like

```
fact := λn.(iszero n) ⌜1⌝
                (times n (fact (pred n)))
```

  is not a simple macro, but a circular definition!
- There's no way to expand out `fact ⌜7⌝` without referring to `fact`.
- Why should we believe this defines anything at all?
  - Why isn't this a non-sensical "circular definition" ?

A Higher-Order Function

- Consider the following *non-circular* definition:

```
F :=  λg.λn.(iszero n) ⌜1⌝
              (times n (g (pred n)))
```

  Then

```
F(f) ↔β* λn.(iszero n) ⌜1⌝
              (times n (f (pred n)))
```

- If the argument to **F** was *already* the factorial function, we'd get the same thing back.
  - Thus the factorial function is a fixed point of **F**.
  - We know that **Y**(**F**) is a fixed point of **F**.
  - Hence **Y**(**F**) is the factorial function. (!?)

Applying Factorial

```
F    :=  λf.λn.(iszero n) ⌜1⌝
                  (times n (f (pred n)))
fact := Y(F)

fact(⌜n⌝)
  ↔β* (F(fact))(⌜n⌝)
  ↔β* (iszero ⌜n⌝) ⌜1⌝
          (times ⌜n⌝ (fact (pred ⌜n⌝)))
```

Applying Factorial

$\texttt{fact}(\ulcorner 2 \urcorner) \leftrightarrow_\beta^*$

Recursive Definitions

- Consider the SML definition
  `fun g(n) = ...code involving g...`
- This is convenient syntax for
  `val rec g = (fn n => ...code involving g...)`
- The function we want is a solution to the *equation*
  `g == (fn n => ...code involving g...)`
- Hence the function we want is a fixed point of
  `fn g => (fn n => ...code involving g...)`

Fibonacci Example

```
FIB := λg.λn.(iszero n) ⌜0⌝
              (iszero (pred n) ⌜1⌝
                (plus (g (pred n))
                      (g (pred (pred n)))))
fib := Y(FIB)
```

Fixed Points

- Every term has at least one fixed point.
  - As mentioned earlier, even **succ** and **not**!
- Some terms have many fixed points
  - For example, λn.n
  - Or, **FIB**
    - Recall the fib and intfib functions from Assignment 1!
  - **Y** picks out the unique *least* fixed point.
    - Which turns out to be the one we "expect".
    - Yields answers as infrequently as possible.

Factorial Revisited

F := λf.λn.(**iszero** n) ⌜1⌝
(**times** n (f (**pred** n)))

f_0 := *...whatever you want...*

f_1 := **F**(f_0)
$\leftrightarrow_\beta^*$ λn.(**iszero** n) ⌜1⌝
(**times** n (f_0 (**pred** n)))

f_2 := **F**(f_1)
$\leftrightarrow_\beta^*$ λn.(**iszero** n) ⌜1⌝
(**times** n (f_1 (**pred** n)))

Factorial Revisited

- Every time we apply **F**, we get a better approximation to the factorial function.
- Thus, to find n! all we need to do is compute
  F(**F**(**F**(**F**(...(**F** f_0)...)))) ⌜n⌝
  where there are n+1 applications of **F**; that is,
  (**F**$^{n+1}$(f_0))(⌜n⌝) $\leftrightarrow_\beta^*$ ⌜n!⌝
- But
  fact = **Y F** $\leftrightarrow_\beta^*$ **F**(**Y F**) $\leftrightarrow_\beta^*$ **F**(**F**(**Y F**))
  $\leftrightarrow_\beta^*$ ... $\leftrightarrow_\beta^*$ (**F**$^{n+1}$(**Y F**))
  so
  fact(⌜n⌝) $\leftrightarrow_\beta^*$ (**F**$^{n+1}$(**Y F**))(⌜n⌝) $\leftrightarrow_\beta^*$ ⌜n!⌝