Announcements

Graded HW2 back this weekend (look for email from GitHub)

Activity Solution

fun is_older (y,m,d) (y',m',d') =
  y < y'
  orelse (y = y' andalso m < m')
  orelse (y = y' andalso m = m' andalso d < d')

Activity

Write a function `num_before_sum` that takes an int called `sum` (assume `sum` is positive) and an int list (assume all positive) and returns an int. The return value is an n such that the sum of the first n elements is < `sum` and the sum of the n + 1 elements is >= `sum`. Assume that the sum of the entire list > n. Summing goes from left to right.

E.g.,

`num_before_sum 3 [0,1,2,3]` returns 2
Activity

fun num_before_sum sum xs = #1(
    foldl (fn (x,(n,acc)) =>
        if acc + x >= sum then
            (n, acc + x)
        else
            (n + 1, acc + x)
    ) (0,0) xs
)

Recall motivation: type annotations are ugly and hard to get right.

Instead of  fun addOne(x: int) = x + 1

Write  fun addOne x = x + 1

type inference

polymorphic type inference

Type inference should work for “generic” code, too!

fun apply f x = f x

- apply addOne 1;
  val it = 2 : int

fun prependHello x = "Hello " ^ x;

- apply prependHello "Dan";
  val it = "Hello Dan" : string

polymorphic type inference

fun apply f x = f x

same basic procedure as “monomorphic” (i.e., non-generic) code:

1. convert to \( \lambda \) expression
2. label with type variables
3. generate constraints
4. unify
5. rename variables
1. convert to $\lambda$ expression

```latex
fun apply \ f \ x = f \ x
f = \lambda f. \lambda x. f \ x
```

2. label with type variables

```latex
fun apply \ f \ x = f \ x
f = \lambda f. \lambda x. f \ x
```

3. generate constraints

```
M ::= x \ variable
| \lambda x.M \ abstraction
| MM \ function application
```

Variable rule: No constraint.

Abstraction rule: If the type of $x$ is $a$ and the type of $M$ is $b$, and the type of $\lambda x. M$ is $c$, then the constraint is $c = a \rightarrow b$.

Application rule: If the type of $M_1$ is $a$ and the type of $M_2$ is $b$, and the type of $M_1M_2$ is $c$, then the constraint is $a = b \rightarrow c$. 

```
<table>
<thead>
<tr>
<th>subexpression</th>
<th>type</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>r</td>
<td>n/a</td>
</tr>
<tr>
<td>x</td>
<td>s</td>
<td>n/a</td>
</tr>
<tr>
<td>f x</td>
<td>t</td>
<td>r = s \rightarrow t</td>
</tr>
<tr>
<td>\lambda x. f x</td>
<td>u</td>
<td>u = s \rightarrow t</td>
</tr>
<tr>
<td>\lambda f. \lambda x.f x</td>
<td>v</td>
<td>v = r \rightarrow u</td>
</tr>
</tbody>
</table>
```
4. unify

\[ x : a, M : b, c : \lambda x. M \Rightarrow c = a \rightarrow b \]
\[ M_1 : a, M_2 : b, c : M_1 M_2 \Rightarrow a = b \rightarrow c \]

<table>
<thead>
<tr>
<th>subexpression</th>
<th>type</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( s \rightarrow t )</td>
<td>n/a</td>
</tr>
<tr>
<td>( x )</td>
<td>( s )</td>
<td>n/a</td>
</tr>
<tr>
<td>( fx )</td>
<td>( t )</td>
<td></td>
</tr>
<tr>
<td>( \lambda x.f )</td>
<td>( u )</td>
<td>( u = s \rightarrow t )</td>
</tr>
<tr>
<td>( \lambda f.\lambda x.f )</td>
<td>( v )</td>
<td>( v = s \rightarrow t \rightarrow u )</td>
</tr>
</tbody>
</table>

5. rename variables in alpha order

\[ x : a, M : b, c : \lambda x. M \Rightarrow c = a \rightarrow b \]
\[ M_1 : a, M_2 : b, c : M_1 M_2 \Rightarrow a = b \rightarrow c \]

<table>
<thead>
<tr>
<th>subexpression</th>
<th>type</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( 'a \rightarrow t )</td>
<td>n/a</td>
</tr>
<tr>
<td>( x )</td>
<td>( 'a )</td>
<td>n/a</td>
</tr>
<tr>
<td>( fx )</td>
<td>( t )</td>
<td></td>
</tr>
<tr>
<td>( \lambda x.f )</td>
<td>( 'a \rightarrow t )</td>
<td></td>
</tr>
<tr>
<td>( \lambda f.\lambda x.f )</td>
<td>( 'a \rightarrow t \rightarrow 'a \rightarrow t )</td>
<td></td>
</tr>
</tbody>
</table>
5. rename variables in alpha order

\[
\begin{array}{ccc}
\text{subexpression} & \text{type} & \text{constraint} \\
\hline
f & 'a \rightarrow 'b & \text{n/a} \\
x & 'a & \text{n/a} \\
f \ x & 'b & \\
\lambda x. f \ x & 'a \rightarrow 'b & \\
\lambda f. \lambda x. f \ x & 'a \rightarrow 'b \rightarrow 'a \rightarrow 'b & \\
\end{array}
\]

Is this the right answer?

\[- \text{fun apply f x = f x;}
\val apply = fn : ('a -> 'b) -> 'a -> 'b\]

activity

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

polymorphic type inference: a problem

\[
\begin{array}{ccc}
\text{subexpression} & \text{type} & \text{constraint} \\
\hline
f & 'a \rightarrow 'b & \text{n/a} \\
x & 'a & \text{n/a} \\
f \ x & 'b & \\
\lambda x. f \ x & 'a \rightarrow 'b & \\
\lambda f. \lambda x. f \ x & 'a \rightarrow 'b \rightarrow 'a \rightarrow 'b & \\
\end{array}
\]

let val id = fn x => x in
  (id "hi", id true)
1. convert to lambda expression

```
let val id = fn x => x in
  (id "hi", id true)
```

```
let val id = λx.x in
  (id "hi", id true)
```

```
let z = U in V is the same as (λz.V)U
(λid.(id "hi", id true))(λx.x)
tup: 'a → 'b → 'a * 'b
(λid.((tup (id "hi"))(id true))(λx.x))
```

2. label types

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>let val id = fn x =&gt; x in (id &quot;hi&quot;, id true)</td>
<td>expr</td>
<td>type</td>
</tr>
<tr>
<td>2</td>
<td>id</td>
<td>type</td>
<td>constraint</td>
</tr>
<tr>
<td>3</td>
<td>&quot;hi&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>tup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(id &quot;hi&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(tup (id &quot;hi&quot;))</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>TRUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(id true)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>((tup ...)(id true))</td>
<td>(lam id___)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(lam id___)</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>(lam x.x)</td>
<td>(lam x.x)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>(lam id___)(lam x.x)</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>
3. generate constraints

\[
\begin{align*}
&x : a, M : b, c : \lambda x. M \Rightarrow c = a \rightarrow b \\
&M_1 : a, M_2 : b, c : M_1 M_2 \Rightarrow a = b \rightarrow c
\end{align*}
\]
4. unify

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>let val id = fn x =&gt; x in (id &quot;hi&quot;, id true)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>expr</td>
<td>type</td>
<td>constraint</td>
</tr>
<tr>
<td>3</td>
<td>id</td>
<td>string -&gt; e</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&quot;hi&quot;</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>tup</td>
<td>'a -&gt; 'b -&gt; 'a * 'b</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(id &quot;hi&quot;)</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(tup (id &quot;hi&quot;))</td>
<td>f</td>
<td>'a -&gt; 'b -&gt; 'a * 'b = e -&gt; f</td>
</tr>
<tr>
<td>8</td>
<td>TRUE</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(id true)</td>
<td>g</td>
<td>string -&gt; e = bool -&gt; g</td>
</tr>
<tr>
<td>10</td>
<td>((tup ...)(id true))</td>
<td>h</td>
<td>f = g -&gt; h</td>
</tr>
<tr>
<td>11</td>
<td>(lam id.____)</td>
<td>i</td>
<td>i = string -&gt; e -&gt; h</td>
</tr>
<tr>
<td>12</td>
<td>x</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>(lam x.x)</td>
<td>k</td>
<td>k = j -&gt; j</td>
</tr>
<tr>
<td>14</td>
<td>(lam id.____)(lam x.x)</td>
<td>l</td>
<td>i = k -&gt; l</td>
</tr>
</tbody>
</table>

replace d with string -> e

what went wrong?

let val id = fn x => x in (id "hi", id true)

We incorrectly assume that all uses of id have the same type.

let-polymorphism

When a term with type variables is bound, don’t assume that type is same for every use.
new language

programming is too complicated...

...because there are too many letters in a program!

Williams introduces the Breph programming language

Breph language

only 8 commands:

e
p
h
r
a
i
m
!

(for extra effect)

programming now much simpler!

Breph language

we will be switching CS134 to Breph

we need your help writing tutorial materials

i.e., hello world

(for some reason we can’t figure it out)

Breph abstract machine

| 0 0 0 0 0 0 0 0 ... | 0 byte

1 byte

hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh!

instruction pointer
Breph language

e  move data pointer to the right
p  move data pointer to the left
h  increment byte at data pointer by 1
r  decrement byte at data pointer by 1
a  accept one byte of input; store @ data pointer loc
i  while loop start
m  while loop end
!  print byte @ data pointer loc using ascii table

Breph language

e  move data pointer to the right
p  move data pointer to the left
h  increment byte at data pointer by 1
r  decrement byte at data pointer by 1
a  accept one byte of input; store @ data pointer loc
i  while loop start
m  while loop end
!  print byte @ data pointer loc using ascii table

Breph language

e  move data pointer to the right
p  move data pointer to the left
h  increment byte at data pointer by 1
r  decrement byte at data pointer by 1
a  accept one byte of input; store @ data pointer loc
i  while loop start
m  while loop end
!  print byte @ data pointer loc using ascii table

Breph language

e  move data pointer to the right
p  move data pointer to the left
h  increment byte at data pointer by 1
r  decrement byte at data pointer by 1
a  accept one byte of input; store @ data pointer loc
i  while loop start
m  while loop end
!  print byte @ data pointer loc using ascii table
Breph language

- e: move data pointer to the right
- p: move data pointer to the left
- h: increment byte at data pointer by 1
- r: decrement byte at data pointer by 1
- a: accept one byte of input; store @ data pointer loc
- i: while loop start
- m: while loop end
- !: print byte @ data pointer loc using ascii table
Breph language

- e: move data pointer to the right
- p: move data pointer to the left
- h: increment byte at data pointer by 1
- r: decrement byte at data pointer by 1
- a: accept one byte of input; store @ data pointer loc
- i: while loop start
- m: while loop end
- !: print byte @ data pointer loc using ascii table

All commands advance instruction pointer
(except i and m, which may jump it elsewhere)

Breph abstract machine

```
1 byte

1 0 0 0 0 0 0 0 ...
```

```
1 byte

hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh!!
```

Breph abstract machine

```
2 0 0 0 0 0 0 0 ...
```

```
1 byte

hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh!!
```
Breph abstract machine

```
50 0 0 0 0 0 0 ...
```

1 byte

“2”

```
hhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhh!
```

Challenge

Top 5 people who write the “brephest” program that prints “Hello Dan Barowy” will earn extra points

("brephest" = shortest)

Demo

```
$h . ./breph.rb program.moo$

$h . ./breph.rb program.moo$
```

"brephest" = shortest