

1 That power function: notes from last lecture

This class centered around the power function and the analogous function for complex number. As a reminder, the function is

```
let rec power (n, x) =  
  if n = 0 then 1  
    else if even n  
      then sqr(power (n/2, x))  
      else x * power (n-1, x)  
  
and sqr x = x * x
```

There were three ways to stage it. The most direct way produced code like

```
1) <fun x -> x * sqr(x * 1)>
```

The second required us to stage the square function like so

```
let sqr x = <~x * ~x>
```

and produced code with duplication.

```
2) fun x -> x * (x * 1) * (x * 1)
```

In the third we used the following square function

```
let sqr x = <let y = ~x in y * y>
```

and got the following code

```
3) fun x -> x * let y = x * 1 in y * y
```

2 Complex multiplication

So now we want to consider the problem of the power function for complex numbers. We can define complex numbers as

```
type complex = float * float
```

and the multiply and square function over complex numbers as

```
let (**) (r1, i1) (r2, i2) = (r1*r2 - i1*i2, r1*i2 + r2*i1)
(**) : float * float -> float * float -> float * float
```

```
let sqr x = x ** x
sqr : float * float -> float * float
```

```
let rec power' (n, x) =
  if n = 0 then (1,0)
  else if even n
    then sqr(power (n/2, x))
    else x ** power (n-1, x)
power : int * (float * float) -> float * float
```

If we only stage the power function we get following code, which corresponds to (1) above.

```
1c) <fun r,i -> mult (r, i) (csqr (mult (r, i) one))>
```

This has both function calls and tuples. Ick. So lets try to stage **. Since sqr is just a wrapper around **, we can escape it without needing to stage sqr explicitly.

```
let (**) (r1, i1) (r2, i2) =
  (<~r1*~r2 - ~i1*~i2>,
   <~r1*~i2 + ~r2 * ~i1>)
```

Using this, however, duplicates code in a number of places. Thus if we write it up in OCaml like so

```
let mult (r1, i1) (r2, i2) =
  (.<~r1 * ~r2 - ~i1* ~i2>.,
   .<~r1* ~i2 + ~r2* ~i1>.)

let sqr x = mult x x

let even x = (x mod 2) == 0

let rec power (n, (a,b)) =
  if n = 0 then (.<1>.,.<0>.)
  else if even n
    then (sqr (power (n/2, (a,b))))
    else (mult (a,b) (power (n-1, (a,b))))

let pull (a,b) = .<~a,~b>.;;
```

Where pull is the isomorphism function we defined between tuples of code and code of tuples, we get output with code explosion like so.

```
# let f = .<fun (x,y) -> ~>(pull (power (3, (.<x>.,.<y>..))))>.;;
val f : ('a, int * int -> int * int) code =
.<fun (x_1, y_2) ->
  (((x_1 *
    (((((x_1 * 1) - (y_2 * 0)) * ((x_1 * 1) - (y_2 * 0))) -
      ((x_1 * 0) + (1 * y_2)) * ((x_1 * 0) + (1 * y_2)))))) -
  (y_2 *
    (((((x_1 * 1) - (y_2 * 0)) * ((x_1 * 0) + (1 * y_2))) +
      (((x_1 * 1) - (y_2 * 0)) * ((x_1 * 0) + (1 * y_2)))))),
  ((x_1 *
    (((((x_1 * 1) - (y_2 * 0)) * ((x_1 * 0) + (1 * y_2))) +
      (((x_1 * 1) - (y_2 * 0)) * ((x_1 * 0) + (1 * y_2)))))) +
    (((((x_1 * 1) - (y_2 * 0)) * ((x_1 * 1) - (y_2 * 0))) -
      ((x_1 * 0) + (1 * y_2)) * ((x_1 * 0) + (1 * y_2)))) * y_2))>.
```

Once again the duplicated code is embedded in two separate pieces of code, and thus our let strategy will not work. So we have to use CPS: but we cannot prove why. We have a first-stage continuation that returns a piece of code. Using CPS we hope to generate code analogues to 3) above.

3 CPS and Staging

When you convert a direct-style function to CPS, you change the type signature. A direct-style function of type $A \rightarrow B$ becomes a CPS function of type $A \rightarrow (B \rightarrow \alpha) \rightarrow \alpha$

When you then stage this code, the goal is to always escape the continuation so that it fully evaluates at the first stage. This means the continuation is of type code and the function is of type $A \rightarrow (B \rightarrow \langle \alpha \rangle) \rightarrow \langle \alpha \rangle$

Now that the return type is of type code, we can do a lot more with it than we can with a generic return type. Namely, we can escape it. This is what we intended, but it is also a little strange. The continuation returning is no longer the last thing that happens, even in the first stage. We now evaluate the continuation and splice in the resultant code. This is a distinct change from the usual idea of continuations. None the less, it does seem to be a coherent concept and it is certainly useful.

Note that there is an isomorphism between $a \rightarrow (b \rightarrow c)$ and $b \rightarrow (a \rightarrow c)$. Like the other isomorphisms discussed in class, these have one-line functions to convert between the two types. In this case the function is its own inverse.

```
let arg_swap f = fun a b -> f b a
```

So we can consider the staged-cps type $A \rightarrow (B \rightarrow \langle \alpha \rangle) \rightarrow \langle \alpha \rangle$ to be analogous to the type $(B \rightarrow \langle \alpha \rangle) \rightarrow (A \rightarrow \langle \alpha \rangle)$. This second type is a transformer. It uses a code generator to construct another code generator. In this sense a continuation does not represent a current computation that we give the result to, but a code generator that we build on top of.

So now to convert our program to CPS. First we need to convert the multiply function.

```
let (**) (r1, i1) (r2, i2) k =
  (< let a = ~r1
    and b = ~r2
    and c = ~i1
    and d = ~i2 in
    ~(k (<a*b - c*d>, <a*c + b*d>))
```

The return type used to be $\langle float \rangle * \langle float \rangle$, a type that has proven very difficult to work with. Now we get to treat this as an argument type, and use the bracket and escape constructs to construct the return value in an efficient, duplication-avoiding fashion.

If we attempt to stage the squaring function in the simple fashion, we get

```
let sqr x k = x ** x k
```

But that sets `**`'s `r1` to the same piece of code as `r2`, and likewise `i1` and `i2`, and thus creates code duplication. Thus we need another let statement.

```
let sqr (r,i) k = <let (a,b) = (~r,~i) in ~((<a>,<b>) ** (<a>,<b>) k)>
```

Our staged CPS power function is thus

```
let rec power' (n, (r,i)) k =  
  if n = 0 then k (<1>,<0>)  
    else if even n  
      then power (n/2, x) (fun r -> sqr r k)  
      else power (n-1, x) (fun r -> x ** r k)
```

Remember pull, the isomorphism function between tuples of code and code of tuples? This works perfectly as a base continuation. For normal continuations, the identity function is the simplest continuation. For this type, pull is the simplest continuation. All it does is convert between the types. The OCaml code is thus

```
let mult (r1, i1) (r2, i2) k =  
  .< let a = .~r1  
    and b = .~r2  
    and c = .~i1  
    and d = .~i2 in  
    .~(k (.<a*b - c*d>., .<a*c + b*d>..))>.  
  
let sqr (r,i) k = .<let (a,b) = (.~r,~i) in .~(mult (.<a>.,.<b>.) (.<a>.,.<b>.) k)>.  
  
let even x = (x mod 2) == 0  
  
let rec power (n, x) k =  
  if n = 0 then k (.<1>.,.<0>.)  
    else if even n  
      then power (n/2, x) (fun r -> sqr r k)  
      else power (n-1, x) (fun r -> mult x r k)  
  
let pull (a,b) = .<.~a, .~b>.
```

And the generated code looks like this.

```
let f = .<fun (x,y) -> .~(power (3, (.<x>.,.<y>..)) pull)>.;;
val f : ('a, int * int -> int * int) code =
.<fun (x_1, y_2) ->
  let a_3 = x_1
  and b_4 = 1
  and c_5 = y_2
  and d_6 = 0 in
  let (a_7, b_8) =
    (((a_3 * b_4) - (c_5 * d_6)), ((a_3 * c_5) + (b_4 * d_6))) in
  let a_9 = a_7
  and b_10 = a_7
  and c_11 = b_8
  and d_12 = b_8 in
  let a_13 = x_1
  and b_14 = ((a_9 * b_10) - (c_11 * d_12))
  and c_15 = y_2
  and d_16 = ((a_9 * c_11) + (b_10 * d_12)) in
  (((a_13 * b_14) - (c_15 * d_16)), ((a_13 * c_15) + (b_14 * d_16)))>.
```

Note that there are duplicated let statements. a_9 and b_{10} are both a_7 .

4 Notes and comments

1. Note in MetaOCaml you have not have arbitrary mixfix operators. so you need `let mult (r1, i1) (r2,i2) k...` like you see above.
2. If you don't know how to stage the function, write down the code you want to generate and see if you can think of a way to generate it directly.
3. Consider if we can prove the only way to do this is to use a continuation.
4. The let statements above are redundant. Oftentimes the pieces of code we are binding are names which can be safely duplicated, not complex computations that we need to give a name to. Is there a way to avoid this?