FUNCTIONAL PROGRAMMING (1)

PROF. SIMON PARSONS

- Imperative programming is concerned with “how”.
- *Functional* or *applicative* programming is, by contrast, concerned with “what”.
- It is based on the mathematics of the lambda calculus (Church as opposed to Turing).
- “Programming without variables”.
- It is inherently concise, elegant, and difficult to create subtle bugs in.

Referential transparency

- The main (good) property of functional programming is *referential transparency*.
- Every expression denotes a single value.
- This value cannot be changed by evaluating an expression or by sharing it between different parts of the program.
- There can be no reference to global data.
- (Indeed there is no such thing as global data.)
- There are no *side-effects*, unlike in *referentially opaque* languages.

```
program example(output)
var flag: boolean

function f(n:int): int
begin
  if flag then f:= n
          else f: 2*n
  flag := not flag
end

begin
  flag := true
  writeln(f(1) + f(2))
  writeln(f(2) + f(1))
end
```

- What is the output?

- Okay, so the answer is 5 followed by 4.
- This is odd since if these were mathematical functions,

$$f(1) + f(2) = f(2) + f(1)$$

  for any f.
- But this is because mathematical functions are functions only of their inputs.
- They have no memory.
- We can always tell what the value of a mathematical function will be just from its inputs.

- At the heart of the “problem” is fact that the global data `flag` controls the value of `f`.
- In particular the assignment:

```
flag := not flag
```

  is the thing that gives this behaviour.
- If we eliminate assignment, we eliminate this kind of behaviour.
- Variables are no longer placeholders for values that change.
- (They are much less variable than variables in imperative programs).

Simple functional programming in HOPE

- We start with a function that squares numbers.
- In the rather odd syntax of HOPE this is:

```
dec square: num -> num;
--- square(x) <= x * x;
```

- Since we aren't really interested in HOPE, we won't explain the syntax in any great detail.
- Note though that first line includes a type definition.

- HOPE is strongly typed.
- Other functional languages aren't typed (LISP for example).
- We call the function by:

  ```
  square(3)
  ```
- Which evaluates to `3 * 3` by definition, and then to 9 by the definition of `*`.
- Note only that, it will *always* evaluate to 9.

- More complex functions:

```
dec max : num # num -> num;
--- max(m, n) <= if m > n then m else n;
```

- and:

```
dec max3 : num # num # num -> num
---max3(a, b, c) <= max(a, max(b, c));
```

- The type definitions indicate that the functions take two and three arguments respectively.

Tuples

- Saying that these functions take two and three arguments is slightly misleading.
- Instead they both have one argument— they are both *tuples*.
- One is a two-tuple and one is a three-tuple.
- This has one neat advantage—you can get functions to return a tuple, and thus several values.

```
dec IntDiv : num # num -> num # num;
--- IntDiv(m, n) <= (m div n, m mod n);
```

- And we can the compose `max(IntDiv(11, 4))`, which will give `3`.

- Another function:

```
dec analyse : real -> char # trueval # num;
---analyse(r) <= (if r < 0 then '-' else '+',
                  (r > = -1.0) and (r=< 1.0),
                  round(r));
```

- Applying

```
analyse(-1.04)
```

- will give `('-', false, -1)`
- Note the *overloading* of `>`.

Recursion

- Without variables, we can't write functional programs with loops.
- So to get iteration, we need recursion.

```
dec sum : num -> num;
---sum(n) <= if n = 0 then 0
             else sum(n - 1) + n;
```

- Which works in the same way as recursion normally does.
- Recursion fits in perfectly with the functional approach.
- Each application of the recursive function is referentially transparent and easy to establish the value of.

- Here is a classic recursive function, with a twist.
- We can define functions to be *infix*.
- Here is the power function as an infix function:

```
infix ^ : 7;

dec ^ : num # num -> num;
--- x ^ y <= if y = 0 then 1
             else x * x ^ (y - 1);
```

- Again, HOPE gives us a very elegant way of defining the function.

Qualified expressions

- Because we don't have variables, sometime it seems we have to do unecessary work when evaluating functions:

```
dec f: num -> num;
---f(x) <= g(square(max(x, 4))) +
           (if x <= 1 then 1
            else g(square(max(x, 4))));
```

- Here we have to evaluate `g(square(max(x, 4)))` twice in some situations.
- With variables, of course, we would have to do this just once.

- Once way around this would be to define the repeated bit as a new function:

```
dec f: num -> num;
---f(x) <= f1(g(square(max(x, 4))))

dec f1: num -> num;
---f1(a, b) <= a + (if b =< 1 then 1 else a)
```

- Efficiency here relies on efficient evaluation in the language.
- Another way is to use *qualified expressions*.

- Consider:

```
dec f : num -> num
--- f(x) <= let a == g(square(max(x, 4)))
            in a + (if x =< 1 then 1 else a))
```

- The ľet construct allows us to extend the set of parameters of a function.
- In general:

```
let <name> == <expression1> in <expression2>
```

- The first expression defines <name> and the second uses it.

- We also have:

```
<expression2> where <name> == <expression1>
```

- So we could also write:

```
dec f : num -> num
--- f(x) <= a + (if x =< 1 then 1 else a))
            where a == g(square(max(x, 4)))
```

- Note that == associates a name with an expression, it does not do assignment.

- To see this:

```
let x == E1 in
  if (let x == E2 in E3)
     then x
     else 1 + x
```

- The first `let` associates `E1` with `x`.
- The second `let` doesn't change this.
- Instead it renames `E2` as `x` within `E3`.
- Outside `E3` `x` has its original meaning.
- So far we have used qualified expressions to save on evaluation.

- We also use them to clarify functions.
- A third use is to decompose tuples.

```
dec quot : num # num -> num;
--- quot(q, r) <= q;

dec rem : num # num -> num;
--- rem(q, r) < = r;

let pair == IntDiv(x, y) in quot(pair) *
                   y + rem(pair)

let(q, r) == IntDiv(x, y) in q * y + r
```

- This latter expression pattern matches `(q, r)` with the result of calling `IntDiv`.

User defined data

- As in most languages, we can't do much interesting stuff in HOPE without defining data.
- This is way simpler in HOPE than in other languages.
- Consider handling lists.
- In C, we have to use `struct`s, and pointers and worry about memory.
- Even in Java we have to use the right constructors.
- In HOPE we just deal with the recursive definition of a list.

- A list is either empty or an element followed by a list.

```
data NumList == nil ++ cons(num # NumList)
```

- Here `nil` and `cons` are constructors.
- A single element list is then:

```
cons(3, nil)
```

- And the list comprising 1, 2 and 3 is:

```
cons(1, cons(2, cons(3, nil)))
```

- To define another kind of list we just do something similar:

```
data CharList == NilCharList
         ++ ConsChars(char # CharList)
```

- Note that there is nothing special about the names `nil` or `cons`.
- Note also that we don't have to say anything about how these lists are represented internally.
- All we tell HOPE is that the list is either a something or a character followed by a list.

- The similarity of the definitions is intentional.
- All list definitions look like this.
- In fact, we can make a general definition:

```
typevar any

data list(any) == AnyNil
    ++ AnyCons(any # list(any))
```

- This is a *polymorphic* definition.
- We parameterize the list by the kinds of objects contained in it.

- With this definition we can build lists of any type:

```
AnyCons(1, AnyCons(2, nil))

AnyCons('a', AnyCons('b', nil))

AnyCons(AnyNil, AnyCons(AnyCons(1, nil)))

AnyNil
```

- The last two are a list of lists, and a list of unspecificed type.

- Lists are so common that they are built into HOPE.

```
infix :: : 7
data list(alpha) == nil ++ alpha :: list(alpha)
```

- We can also write lists as, for example `[1, 2, 3]`.
- Strings are lists of characters.
- With this information it is easy to write functions to handle lists.

```
dec join : list(alpha) # list(alpha)
                -> list(alpha);
--- join(nil, L) <= L;
--- join(x::y, L) <= x :: join(y, L)

dec rev: list(alpha) -> list(alpha);
--- rev(nil) <= nil;
--- rev(x::l) <= rev(l) join [x];
```

- Note that `join` is predefined in HOPE as the infix function `<>`.

Higher order functions

- Consider

```
dec IncList : list(num) -> list(num);
--- IncList(nil) <= nil;
--- IncList(x::l) <=
            (x + 1)::IncList(l);

dec MakeStrings : list(char)
              -> list(list(char));
--- MakeStrings(nil) <= nil;
--- MakeStrings(c::l) <=
              [c]::MakeStrings(l);
```

- While doing different things, these two functions have the same basic form.

- Both operate on a list and apply a function to every member of the list.
- The two functions are:

```
dec Inc : num -> num
--- Inc(n) <= n + 1

dec Listify : char -> list(char)
--- Listify(c) <= [c]
```

- We can capture this by defining a *higher order function*

- This takes a function and a list as arguments and applies the function to every member of the list.

```
dec map : (alpha -> beta) # list(alpha)
                  -> list(beta);
--- map(f, nil) <= nil;
--- map(f, x :: l) <= f(x) :: map(f, l);
```

- We can then write down the equivalent of our two earlier functions.

```
map(Inc, L)

map(Listify, L)
```

- Of course, this relies on us having defined `Listify` and `Inc`.
- However, we don't even have to do this.
- HOPE provides us with the means to write anonymous function bodies when and where we need them.
- For example:

```
lambda x => x + 1
```

- Here we have to use the word `lambda`.

- In general, we can replace any function with a `lambda` expression.
- We replace:

```
--- f(x) <= E
```

- with

```
lambda x => E
```

- Thus the function `IncList` is the same as:

```
map(lambda x => x + 1, L)
```

- Note that we have problems defining a recursive `lambda` because there is no name to use in the recursion.
- Instead we have to use a `let` or `where`.
- For example:

```
let f == lambda x => if x = 0 then 0
                     else x + f(x - 1)
```

- (which computes the sum of the first 3 numbers.)
- Such constructs are called *recursive let* and *recursive where¿*

- Some functional languages make these separate constructs (eg `letrec`).
- In HOPE `lambda` expressions can also contain a number of parts.

```
--- IsEmpty(nil) <= true;
--- IsEmpty(_::_) <= false;
```

- becomes

```
lambda nil => true | _::_ => false
```