MAU11602
Lecture 8
2026-02-05

Functions

It's hard to programme in a language which doesn't allow you to define functions. We want to be able to write things like

```
(define (square x) (* x x ))
```

or

```
fun square x = x * x
```

and then refer to `square` later.

Two distinct things are happening here, creating a function and naming it. We often want to do these in combination, but sometimes separately.

There are various notations for anonymous functions.

Mathematically, we might refer to the function x^2, for example. This is problematic if the definition depends on other variables though.

Is x^2y the function which takes its argument, squares it, and then multiplies by y or the function which takes its argument and multiplies it by x^2, or is it a function of two variables which squares its first argument and then multiplies by its second argument?

A better notation uses $\mapsto$. The three functions above would be $x \mapsto x^2y$, $y \mapsto x^2y$, and $(x, y) \mapsto x^2y$ or $x \mapsto (y \mapsto x^2y)$ in this notation.

Functions, continued

Another notation for anonymous functions is λ expressions. The functions $x \mapsto x^2y$, $y \mapsto x^2y$, and $x \mapsto (y \mapsto x^2y)$ become $\lambda x.x^2y$, $\lambda y.x^2y$ and $\lambda xy.x^2y$ in lambda notation.

We should be able to apply these anonymous functions to arguments, just like we apply named functions, so

$$(\lambda x.x^2)\,7 = 49.$$

There is an entire theory of computation built around anonymous functions and their application to arguments, called the lambda calculus.

From this perspective, what we mean when we define a function by an equation like $f(x) = x^2$ is $f = \lambda x.x^2$.

We define the anonymous function $\lambda x.x^2$ and then deanonymise it by assigning it the name f.

What are functions?

There are really two notions of function: *intensional* and *extensional*.
These terms come from mathematical logic. Unlike mathematics or computer science it has a reasonable terminology. It has weird spelling though.
Intensional functions are defined in terms of how they're computed.
Extensional functions are defined in terms of their values.
More precisely, an extensional function is a set of ordered pairs with the property that no two distinct members of the set have the same left element. If we think of the left element as the argument as the argument and right element as the value, and the set of left elements of such pairs as the domain, then this means for each choice of argument in the domain of the function there's exactly one value for the function at that argument.
The same extensional function can correspond to multiple intensional functions, or none at all.

What are functions? (continued)

The same extensional function can correspond to multiple intensional functions, or none at all.
Lambda notation, or equivalents, are used to specify intensional functions.
$\lambda x.x + x$ and $\lambda x.2x$ are extensionally the same, but intensionally different.
More precisely, these are two different intensional functions. There is a single extensional function which we can think of as the set of ordered pairs of the form $(x, x + x)$ or of the form $(x, 2x)$.
It's possible to show, by various arguments, that there are extensional functions which don't correspond to any intensional function.
We need both notions in this module, so we need to distinguish them.
Until further notice, function means intensional function.
The standard mathematics convention is that function means extensional function.

Adding functions to our language

Following SML I'll use the syntax `fn` x `=>` e for anonymous functions, so the function which takes its argument and squares it is `fn x => x * x`.

If we want to use it multiple times in some expression then we give it a name the same way we give anything else a name, with a let expression, as in

```
let val square = fn x => x * x
in square 3 + square 4
end
```

which evaluates to 25.

We assign names to functions in the same way we assign names to variables because *functions are values*.

Conceptually two distinct things are happening here, creating a function and assigning it a name, but there's a convenient shorthand for doing both at the same time:

```
let fun square x = x * x
in square 3 + square 4
end
```

Functions

By now we know what adding something to our language means:
- Adding grammar rules, so it can be parsed.
- Adding typing rules, so it can be assigned the correct type.
- Adding reduction rules, so it can be evaluated.

We need grammar rules for function definition and function application, like

```
expr ::= ... | "fn" ws var ws "=>" expr | expr expr
```

... indicates the other ways to form an expression, which we leave unchanged.
I am lying of course. We also need to worry about associativity and precedence.
You might worry that `expr expr` is too general. Shouldn't we limit the first `expr` to functional expressions?
No. That's taken care of by the typing rule

$$\frac{\vdash e_1 : \tau_1 \texttt{ -> } \tau_2 \qquad \vdash e_2 : \tau_1}{\vdash e_1 e_2 : \tau_2}$$

which prevents us from applying things which aren't functions.

Typing function definitions

We also have a typing rule for (anonymous) function definition:

$$\frac{x : \sigma \vdash e : \tau}{\vdash \texttt{fn } x \texttt{ => } e : \sigma \to \tau}$$

This says that if, in a context where x has type σ, e has type τ, then `fn` x `=>` e has type σ `->` τ.

This is the second time we've seen a context-dependent typing rule. The first time was in the case of let expressions, which had the typing rule

$$\frac{\vdash e_1 : \sigma \qquad x : \sigma \vdash e_2 : \tau}{\vdash \texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end } : \tau}$$

Reduction rules

We also need reduction rules for functions. We never attempt to evaluate expressions which fail type checking so any function application expression is a functional expression, i.e. an expression of the form `fn` x `=>` e_1, followed by some other expression e_2. In case e_2 is a value the whole thing reduces to e_1, with that value substituted for all free occurrences of x.
As usual there's also a congruence rule saying that if e_2 reduces to e_3 then $e_0\ e_2$ reduces to $e_0\ e_3$.
Together these mean that we evaluate a function application by fully evaluating the function argument and then substituting the value in the body of the body of the function definition.

Functions and let expressions

What is the type of an expression of the form $(\texttt{fn}\ x\ \texttt{=>}\ e_1)\,e_2$?

$$\dfrac{\dfrac{x : \sigma \vdash e_1 : \tau}{\vdash \texttt{fn}\ x\ \texttt{=>}\ e_1 : \sigma\ \texttt{->}\ \tau} \qquad \vdash e_2 : \sigma}{\vdash (\texttt{fn}\ x\ \texttt{=>}\ e_1)\,e_2 : \tau}$$

As far as typing is concerned this is the same as an expression of the form $\texttt{let val}\ x\ \texttt{=}\ e_2\ \texttt{in}\ e_1\ \texttt{end}$.

They also evaluate in the same way, i.e. by reducing e_2 to a value and then substituting that value for free occurrences of x in e_1.

Once we have functions let expressions are redundant, but they can aid readability.

It's also possible to reverse this equivalence, and interpret function evaluation in terms of let expressions. In a context where the name `square` has been assigned to the functional value `fn x => x * x` the expression `square (y + z)` will be equivalent to `let val x = y + z in x * x end`, not just in the usual sense of observational equivalence but in the details of which expressions are evaluated how many times and when.

Ordered pairs

The rules for functions on the previous slides are all for unary functions, i.e. functions of one argument. As discussed before, we can simulate functions of more arguments either by currying or with tuples.

No further work is required to introduce currying; it works automatically in any language in which functions are values.

Ordered pairs are introduced as before. Once we have them we can simulate n-tuples for any $n \geq 2$. 1-tuples are only a notational convenience.

0-tuples are actually useful so we'll also add a `unit` type to replace 0-tuples. There is only one value of type `unit`, which we'll write as if it were a 0-tuple: `()`.

Nearly all computer languages have some form of pairs or tuples although it took Fortran 34 years to add them.