MAU11602
Lecture 5
2026-01-29

Conditionals and type variables

The method used to generate the derivation on the last slide is more or less how languages with type inference do it, but there are complications.
Consider the rule for conditionals:

$$\frac{\vdash e_1 : \texttt{bool} \qquad \vdash e_2 : \tau \qquad \vdash e_3 : \tau}{\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

There are now three premises, one for each subexpression.
More interestingly, there is also a type variable τ. It's needed because conditionals should work with any type for the last two expressions, but they should be of the same type. The full expression then has that type.
We say that conditionals are *polymorphic*, meaning not all the types are fixed.
Polymorphism complicates type inference but as long as the derivation ends with values of fixed type we can always determine all the type variables.

Function types

Polymorphic types also come up when dealing with functions. We haven't added functions to our language yet (?) but suppose we do, and write unary functions, i.e. functions of a single argument, like `f ( x )`. What is the typing rule?

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

This has two type variables and also the type constructor `->`. τ_1 `->` τ_2 is the type of functions taking an argment of type τ_1 and producing a result of type τ_2, so e_1 is an expression of functional type.
Suppose we had a function `isZero` such that `isZero 0` reduces to `true` and `isZero` v reduces to `false` otherwise.
This function takes an `int` and produces a `bool` so it should have type `int -> bool`.
If it doesn't then our type system will be unsound.
In fact we also need a congruence rule, saying that if e_1 reduces to e_2 then `isZero` e_1 reduces to `isZero` e_2.

Arithmetic operators as functions

Instead of introducing grammatical rules, typing rules and reduction rules for `isZero` it's better to wait until we've created a way to define new functions within the language and then define it.
I said we didn't have functions yet, but really we do have functions. That's what `+`, `-`, and code `*` are.
We just can't define new functions within the language yet.
For historical reasons these particular functions are written in a weird notation.
My rule was for unary functions, but these appear to be functions of two arguments.
What should we do?
- We could introduce new rules for binary functions, then presumably for ternary functions, etc.
- We could introduce a new type constructor for ordered pairs, so that `+`, for example, will be a function from ordered pairs of `int`s to `int`s.
- We could consider `+` as a function from `int`s to functions from `int`s to `int`s.

Ordered pairs

One option is to introduce a notation for assembling and disassembling ordered pairs.
We can write `(17, 42)`, for example, for an ordered pair whose left element is 17 and whose right element is 42.
Although we don't need it now, we can consider ordered pairs with inhomogeneous types, like `(17, false)`.
The typing rule for pairs is

$$\frac{\vdash e_1 : \tau_1 \qquad \vdash e_2 : \tau_2}{\vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} : \tau_1 \texttt{*} \tau_2}$$

This is a polymorphic rule. Just as we had a type constructor `->` for functional types, we now have a constructor `*` for pairs.
Parentheses and commas are the standard notation for ordered pairs in mathematics, and also in most programming languages which have ordered pairs.
Note that parentheses are used in at least three ways: constructing ordered pairs, indicating order of operations, and for function evaluation, e.g. $f(x)$.

Ordered pairs, continued

We also need a way to access the elements of a pair. We can use the (much less standard notation) #1 and #2 for the left and right elements, with typing rules

$$\frac{\vdash (e_1, e_2) : \tau_1 * \tau_2}{\vdash \#1\ (e_1, e_2) : \tau_1}$$

$$\frac{\vdash (e_1, e_2) : \tau_1 * \tau_2}{\vdash \#2\ (e_1, e_2) : \tau_2}$$

We can now think of addition, for example, as a function from ordered pairs of `int`s to `int`s, i.e. as something of type `int * int -> int`.
There is one notational awkwardness though. If we use parentheses for pairing and for function arguments then applying a function f to the pair (x, y) should look like $f((x, y))$.
So let's not use parentheses for function arguments.

Tuples

If we have pairs, why not triples, quadruples, etc.
We could simulate a triple with a pair, the second element of which is a pair, etc., but it's convenient to have them natively and most programming languages provide them.
More generally, we can consider n-tuples for any natural number n.
Any? How about 1-tuples?
The only real use for them is to make unary functions look like we expect, i.e. $f(x)$ can be understood f as f applied to the 1-tuple x, just as $f(x, y)$ is f applied to the ordered pair, i.e. 2-tuple, (x, y).
Then how about 0-tuples?
These are more useful than you might expect!
Eventually I will add ordered pairs to our language, but not just yet.

Currying

The last of our three options, considering `+` as a function from `ints` to functions from `ints` to `ints`, is probably the least familiar.

It works especially well once we drop the parentheses from function evaluation, i.e. if we write `f x` instead of `f ( x )`.

Now $+ m$ will be the function which takes an integer and adds m to it. This produces an integer, so is of type `int -> int`. For example `+ 1` is the increment function.

`+` is therefore a function which takes an integer and produces one of these functions, so it is of type `int -> (int -> int)`.

The parentheses show that this is a function from integers to functions from integers to integers. `( int -> int ) -> int` would be the type of a function from functions from integers to integers to integers.

With this definition of `+` we write the sum of 1 and 2 as `+ 1 2`. In other words, this is prefix notation, but reinterpreted. Now it's the increment function applied to 2.

This trick is known as currying, after Haskell Curry.

Pairs and currying in practice

In the evaluator from last week I typed in things like

```
eval pref "+ - 1 * 2 3 4";
eval inf "( ( 1 - ( 2 * 3 ) ) + 4 )";
eval postf "1 2 3 * - 4 +";
```

`eval` is the evaluator. It needs and evaluation order and a string to evaluate.

SML gives you information about types. When it started it printed some lines, including

```
val pref = fn : token * token stack -> token stack
val postf = fn : token * token stack -> token stack
val inf = fn : token * token stack -> token stack
val eval = fn : (token * token stack -> token stack) -> string -> int
```

What determines the evaluation order is what happens to the stack when we read the next token from input, so I represented them by a function which takes a pair, consisting of a token and a stack, and gives you the updated stack.

I could also have curried these, i.e. represented them as `token -> (token stack -> token stack)`, but chose not to.

Pairs and currying in practice, continued

The `eval` function is curried. It has type
`(token * token stack -> token stack) -> string -> int`.
The convention that `->` is right associative, so this means
`(token * token stack -> token stack) -> (string -> int)`.
I could instead have made this a function from pairs of evaluation order and string to `ints`, i.e. a `(token * token stack -> token stack) * string -> int`.
But if we curry it then `eval pref` by itself, for example, is a function of type `string -> int`, a prefix evaluator.
If I'm going to do a lot of prefix evaluation I can apply `eval` to `pref` once, and the apply the resulting function to many different strings.
This is what's called partial evaluation or staged evaluation.