MAU11602
Lecture 6
2026-02-04

A simple language

Before complicating our language further let's first simplify it.

```
sexpr ::= sval | bfun ws sexpr ws sexpr
          | "ifthenelse" ws sexpr ws sexpr we sexpr
 bfun ::= "*" | "+" | "-" | "<" | "<=" | "=" | ">" | ">="
          | "orelse" | "andalso"
 sval ::= int | "false" | "true"
 expr ::= sexpr | bfun
```

`sval`'s are simple values, the kind we are hoping for as our final answer, and an `sexpr` is an expression we hope evaluates to an `sval`. These are distinct from functional expressions, which we now also have.

- `false` and `true` are of type `bool` while integer values are of type `int`.
- +, −, and ∗ are of type `int -> int -> int`.
- `orelse` and `andalso` are of type `bool -> bool -> bool`.
- < <=, =, >=, and > are of type `int -> int -> bool`.

A simple language, continued

We've taken care of the types of values, but we also need typing rules for more complicated expressions.

$$\frac{\vdash e_1 : \texttt{bool} \qquad \vdash e_2 : \tau \qquad \vdash e_3 : \tau}{\vdash \texttt{ifthenelse}\ e_1\, e_2\, e_3 : \tau}$$

$$\frac{\vdash e_1 : \tau_1\ \texttt{->}\ \tau_2\ \texttt{->}\ \tau_3 \qquad \vdash e_2 : \tau_1 \qquad \vdash e_3 : \tau_2}{\vdash e_1\, e_2\, e_3 : \tau_3}$$

These are similar to the typing rules given earlier.
The reduction rules are as follows.
- `ifthenelse false` e_2 e_3 reduces to e_3 and `ifthenelse true` e_2 e_3 reduces to e_2.
- `orelse false` e and `andalso true` e both reduce to e while `andalso false` e and `orelse true` e reduce to `false` and `true`, respectively.
- If f is an arithmetic operator or relation then $f\ v_2\ v_3$ reduces to what you would expect. If e_3 reduces to e_5 then $f\ v_2\ e_3$ reduces to $f\ v_2\ e_5$. If e_2 reduces to e_4 then $f\ e_2\ e_3$ reduces to $f\ e_4\ e_3$.

Properties of this language and type system

The grammar is unambiguous, the order of evaluation is fully deterministic, all typable `sexpr`'s evaluate to an `sval`, and the type system is sound.
None of these facts is obvious, except maybe determinism.
An input string can fail to be evaluated to an integer or Boolean in three different ways:
- The lexer could fail to convert it into a sequence of tokens, as happens with `<= + - 1 * 2 "three" 4 0`, because `"three"` is a string, not one of the allowed token types.
- The parser could fail to parse that sequence as an expression, as happens with `<= + - 1 * 2 3 4 0 5`.
- The parsed expression could fail to typecheck, as happens with `<= + - 1 * 2 3 false 0`. It fails at the subexpression `+ - 1 * 2 3 false`, consisting of three subexpressions `+`, of type `int -> int -> int`, `- 1 * 2 3` , of type `int`, and `false`, of type `bool`. If we try to match this against the typing rule for expressions of the form $e_1\, e_2\, e_3$ it doesn't work.

The one stage where it *can't* go wrong is evaluation.

Type checking example

$$\dfrac{\vdash \texttt{-: i->i->i} \qquad \vdash \texttt{1: i} \qquad \dfrac{\vdash \texttt{*: i->i->i} \qquad \vdash \texttt{2: i} \qquad \vdash \texttt{3: i}}{\vdash \texttt{* 2 3: i}}}{\vdash \texttt{- 1 * 2 3: i}}$$

$$\dfrac{\vdash \texttt{<=: i->i->b} \qquad \dfrac{\vdash \texttt{+: i->i->i} \qquad \vdash \texttt{- 1 * 2 3: i} \qquad \vdash \texttt{4: i}}{\vdash \texttt{+ - 1 * 2 3 4: i}} \qquad \vdash \texttt{0: i}}{\vdash \texttt{<= + - 1 * 2 3 4 0: b}}$$

To make this fit I had to abbreviate the type names and split the diagram into two pieces.

Let expressions

What else would we like in our language?
How about the ability to name expressions and then refer to them by name?
So we could write something like
`let val x = - 1 2 val y = + 3 4 in * x y end` instead of `* - 1 2 + 3 4`.
This is a bit wordier than necessary, but the `val` and `end` keywords simplify parsing, and keep us (mostly) compatible with SML syntax.
Now we have variables!
We may regret this.
The idea is the we evaluate everything between the `let` and `in` and then substitute the values obtained in for the corresponding variables in between the `in` and the `end`.
Question: Are the following equivalent?

```
let val x = * 1 * 2 * 3 * 4 5 in * x x end
* * 1 * 2 * 3 * 4 5 1 * 2 * 3 * 4 5
```

Answer: observationally yes, both evaluate to $(5!)^2 = 14400$, but the first is more efficient because only calculate 5! once. In practice this may not matter though.

Parsing let expressions

Adding let expressions to our grammar is easy. We just add two more possibilities to the rule for `sexpr`'s:

```
var | "let val" ws var ws "=" ws sexpr ws "in" ws sexpr ws "end"
```

and make a rule for names of variables.

We'll also allow more types of whitespace so we can write let expressions more readably, e.g

```
let
  val y = true
in
  let
    x = 0
  in
    < 1 x y
  end
end
```

Typing let expressions

The typing rule for let expressions is

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

For the first time we have something to the left of the $\vdash$ sign. $x : \sigma \vdash e_2 : \tau$ is to be interpreted as e_2 has the type τ when x has type σ.
Just as σ and τ are type variables, x is a variable variable. It stands in for an arbitrary variable.
In general what we have to the left of the $\vdash$ is a context, i.e. a set of assignments of types to variables. It just happens the context has been empty until now. What we have to the right is a type assignment which is understood to hold in that context.
We'll need some rules for manipulating context, but we'll worry about that later.

An example

$$\dfrac{\dfrac{\vdash \texttt{<}\colon \texttt{int -> int -> bool}}{\texttt{x}\colon \texttt{int} \vdash \texttt{<}\colon \texttt{int -> int -> bool}} \quad \dfrac{\vdash \texttt{1}\colon \texttt{int}}{\texttt{x}\colon \texttt{int} \vdash \texttt{1}\colon \texttt{int}} \quad \texttt{x}\colon \texttt{int} \vdash \texttt{x}\colon \texttt{int}}{\texttt{x}\colon \texttt{int} \vdash \texttt{< 1 x}\colon \texttt{bool}}$$

$$\dfrac{\dfrac{\dfrac{\vdash \texttt{andalso}\colon \texttt{bool -> bool -> bool}}{\texttt{y}\colon \texttt{bool} \vdash \texttt{andalso}\colon \texttt{bool -> bool -> bool}}}{\texttt{y}\colon \texttt{bool}, \texttt{x}\colon \texttt{int} \vdash \texttt{andalso}\colon \texttt{bool -> bool -> bool}} \quad \dfrac{\texttt{x}\colon \texttt{int} \vdash \texttt{< 1 x}\colon \texttt{bool}}{\texttt{y}\colon \texttt{bool}, \texttt{x}\colon \texttt{int} \vdash \texttt{< 1 x}\colon \texttt{bool}}}{\texttt{y}\colon \texttt{bool}, \texttt{x}\colon \texttt{int} \vdash \texttt{andalso < 1 x y}\colon \texttt{bool}}$$

$$\dfrac{\vdash \texttt{true}\colon \texttt{bool} \quad \dfrac{\dfrac{\vdash \texttt{0}\colon \texttt{int}}{\texttt{y}\colon \texttt{bool} \vdash \texttt{0}\colon \texttt{int}} \quad \texttt{y}\colon \texttt{bool}, \texttt{x}\colon \texttt{int} \vdash \texttt{andalso < 1 x y}\colon \texttt{bool}}{\texttt{y}\colon \texttt{bool} \vdash \texttt{let val x = 0 in andalso < 1 x y end}\colon \texttt{bool}}}{\vdash \texttt{let val y = true in let val x = 0 in andalso < 1 x y end end}\colon \texttt{bool}}$$

Context rules

Steps in a type derivation can add context, remove context, or preserve context. What's called the weakening rule says that if a type inference is valid then it remains valid when we add more context to the premises, like

$$\frac{\mathtt{x} : \mathtt{int} \vdash \mathtt{<\ 1\ \ x} : \mathtt{bool}}{\mathtt{y} : \mathtt{bool}, \mathtt{x} : \mathtt{int} \vdash \mathtt{<\ 1\ \ x} : \mathtt{bool}}$$

We can only remove context when a rule specifically allows it. The only example so far is our rule for let expressions, which removes some of the context from the second premise:

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

Otherwise we just carry the context along. Maybe all of our other rules should include context above and below the line, but the context would be the same on both sides, so we don't write it.