MAU11602
Lecture 4
2026-01-29

Die Welt als ob

The rules I've given are highly prescriptive about how to parse and evaluate expressions.
For reasoning about the language this is great. There is no ambiguity. Any expression can be evaluated in at most one way.
For implementing it this is less great. Optimisations are strictly forbidden.
Consider evaluating `(1 - 2 * 3 + 4) * 0` . Our reduction semantics require fully evaluating the expression `1 - 2 * 3 + 4` before multiplying by `0`, even though we know its value can't matter.
The best way to deal with this is to specify unambiguous rules and then say that a conforming implementation is one which evaluates any expression *as if* it followed those rules, rather than one which actually does follow them.
A compiler or interpreter writer who wants to optimise can make optimisations if there's no context in which the behaviour changes observably.
This is harder than it looks. In our current language it's safe to replace expressions of the form e `* 0` with 0 but if we add elements which might make e unevaluable then this is no longer safe.

This happens in maths too!

This is not some weird computer science thing. The same issue happens in mathematics. They inherited their problem from us.

Consider the following five rules for manipulating arithmetic expressions:

- We can safely replace any expression of the form $(w/x) \cdot (x/z)$ with w/z.
- We can safely replace any expression of the form $(w/x) \cdot (y/w)$ with y/x.
- We can safely replace any expression of the form $u \cdot v$ with $v \cdot u$.
- We can safely replace any expression of the form $u \cdot 0$ with 0.
- We can safely replace any expression of the form $0 \cdot v$ with 0.

Which of these do you believe? I hope not all of them!

Consider the case $w = 0$, $x = 1$, $y = 1$, $z = 0$.

On any interpretation where both multiplicands must be evaluated before multiplication the expression $(w/x) \cdot (y/z)$ cannot be evaluated.

If we believe the second rule then we can replace it with $1/1$, which evaluates to 1.

The horror continues.

- We can safely replace any expression of the form $(w/x) \cdot (x/z)$ with w/z.
- We can safely replace any expression of the form $(w/x) \cdot (y/w)$ with y/x.
- We can safely replace any expression of the form $u \cdot v$ with $v \cdot u$.
- We can safely replace any expression of the form $u \cdot 0$ with 0.
- We can safely replace any expression of the form $0 \cdot v$ with 0.

Again $w = 0$, $x = 1$, $y = 1$, $z = 0$.

If we believe the first and third rules we can replace $(0/1) \cdot (1/0)$ with $(1/0) \cdot (0/1)$, then with $1/1$, and then finally 1.

If we believe the fifth rule we can first evaluate $0/1$ to get $0 \cdot v$ where v is $1/0$, we avoid evaluating v by applying the rule and get 0.

If we believe the third and fourth rules we replace $(0/1) \cdot (1/0)$ with $(1/0) \cdot (0/1)$, evaluate $0/1$ to 0, giving $u \cdot 0$, where u is $1/0$. We don't evaluate this but instead apply the fourth rule to get 0.

Which rules are actually valid for our language and its reduction semantics?

The first two are vacuous and the last three are all valid!

Referential transparency, again

Referential transparency is an optimisation strategy.
It allows you to substitute an unevaluated expression somewhere the reduction rules don't explicitly permit.
Like the optimisation rules from the previous slide, it may or may not be valid, and adding things to our language can break it.
You can't just decree that referential transparency is valid in a language, just like can't decree that all arithmetic operators are associative.
If you want to use it safely you have to prove it, and then anytime you modify the language you have to prove you haven't broken it.

Postfix evaluation

The great advantage of postfix was that it was the easiest to write an evaluator for. Our stack based evaluator read input tokens and pushed integers onto the stack and applied operators to the two topmost elements and then pushed the result.
What changes do we need if we add booleans, boolean operators and conditionals?
- Our stack will contain integers *and booleans*.
- When we read an arithmetic operator, *boolean operator, or relation* we pop the top two items, *check their types*, apply the operator *or relation* and push the result.
- *When we read* `ifthenelse` *we pop the top three items, check whether the last popped value is* `true` *or* `false`, *and push one of the other two.*

Does this work?
Yes and no. It gives the right answer, but it evaluates expressions which don't need to be evaluated, like the second operand of a boolean expression when only the first is needed, or the branch not chosen in an `ifthenelse`.
For now this is an inefficiency, but if we introduce unevaluable expressions, e.g. division by zero, it can become an error.

Prefix evaluation

With prefix our stack is more complicated, but we always see the boolean operator or `ifthenelse` before its arguments.
We can skip evaluating it, although we still need to parse it to see where it ends.
We might also need to type check it, depending on how strict our language is.

Typing rules

Typing rules specify the type of an expression, based on the type of subexpressions.
For example `1 - 2 * 3 + 4` has integer type because both `1 - 2 * 3` and `4` are both of type integer and expressions of the form $e_1 + e_2$ are of integer type if e_1 and e_2 are of of integer type.
This example is in the standard infix notation. Type checking happens after parsing so we already know that the two subexpressions are `1 - 2 * 3` and `4` and not `1 - 2` and `3 + 4` or `1` and `2 * 3 + 4`.
The order is lexing, then parsing, then type checking, then evaluation.
`1 - 2 <= 3 + 4` is of boolean type since `1 - 2` and `3 + 4` are both of type integer and expressions of the form e_1 `<=` e_2 are of boolean type if e_1 and e_2 are of of integer type.
An expression may not be typable. This is indeed the whole point of a type system, to make some expressions untypable.
We don't want `42 <= true` to be typable.

Soundness

Values are expressions which can't be reduced further. All values should have a type.
A *sound* type system is one where each step reduces a typable expression to one of the same type.
Evaluation proceeds by a finite sequence of steps, so every expression evaluates to a value of the same type.
Warning: Not all type systems are sound! Lots of popular languages have unsound type systems.

Typing rules

I wrote a typing rule informally earlier but there is a standard formal notation for them. My example rule was that expressions of the form e_1 + e_2 are of integer type if e_1 and e_2 are of of integer type.
Note that e_1 and e_2 are not part of our language, which still doesn't have variables. They are part of the metalanguage, the language used for describing the language.
We write this rule formally as

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 \ \texttt{+} \ e_2 : \texttt{int}}$$

Ignore the $\vdash$'s for now. The colon : is to be read as has the type, so $e_1 : \texttt{int}$ means e_1 has the type `int`.
The statements above the horizontal line are premises and the one below is the conclusion. The blank space between two statements above the line should therefore be read as and.

More rules

There are simlar rules for the other arithmetic operations. I won't write them down. There are also rules for Boolean operators, like

$$\frac{\vdash e_1 : \texttt{bool} \qquad \vdash e_2 : \texttt{bool}}{\vdash e_1 \ \texttt{orelse} \ e_2 : \texttt{bool}}$$

Rules can involve a mix of types. This happens for the relations:

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 \ \texttt{<=} \ e_2 : \texttt{bool}}$$

We can read these rules from top to bottom or from bottom to top. The one above either says that if we've shown that e_1 and e_2 are of type `int` then we can conclude that e_1 `<=` e_2 is of type `bool` *or* that if we need to show that e_1 `<=` e_2 is of type `bool` then it suffices to show that e_1 and e_2 are of type `int`.

Typing derivations

We can string instances of rules together to get a typing derivation for an expression. The following shows that `1 - 2 * 3 + 4 <= 0` is of Boolean type:

$$\dfrac{\dfrac{\vdash \texttt{1}\colon \texttt{int} \qquad \dfrac{\vdash \texttt{2}\colon \texttt{int} \qquad \vdash \texttt{3}\colon \texttt{int}}{\vdash \texttt{2 * 3}\colon \texttt{int}}}{\vdash \texttt{1 - 2 * 3}\colon \texttt{int}} \qquad \vdash \texttt{4}\colon \texttt{int}}{\dfrac{\vdash \texttt{1 - 2 * 3 + 4}\colon \texttt{int} \qquad \vdash \texttt{0}\colon \texttt{int}}{\vdash \texttt{1 - 2 * 3 + 4 <= 0}\colon \texttt{bool}}}$$

To convince someone you've assigned an expression its correct type you work downwards, so every step applies a rule to facts you've already established.
To construct a typing derivation you work upwards. `1 - 2 * 3 + 4 <= 0` can only be parsed as `1 - 2 * 3 + 4 <= 0`, then `<=`, and then `0`. The only one typing rule for `<=` gives a boolean expression, so that fixes the last line, and therefore the two premises on the line above.
We continue upwards until we reach values, or find an untypable expression.