MAU11602
Lecture 9
2026-02-11

Tagged unions

Our stack based evaluator for postfix arithmetic in Lecture 1 used a stack of integers, but the prefix evaluator needed a stack of items which could be either integers or arithmetic operators.

The cleanest way to do this is with what's called a *tagged union*. In our language for types we can represent this as `int + (int * int -> int)` or `int + (int -> int -> int)`, depending on how we want to represent the operators. `+` is a type constructor for tagged unions, just as `*` is for ordered pairs and `->` is for functions.

I've assumed here that it has higher precedence than `->` and lower precedence than `*`. Otherwise I'd need more parentheses, like `int + ((int * int) -> int)` or `int + (int -> (int -> int))`.

Language support

Tagged unions are not as common a language feature as ordered pairs. Weakly typed languages sometimes simulate them by allowing functions to inspect the types of their arguments, but this lacks two important features of real tagged unions:

- A tagged union can *only* contain the types specified, not anything else. The type checker will check this, and check that functions are prepared to handle either.
- The types in a tagged union don't need to be distinct.

To see why you might want the types to be the same, consider points in the plane. You could represent them by Cartesian coordinates (`real * real`), polar coordinates (`real * real`), or a tagged union (`real * real + real * real`).

Sometimes there's no useful information beyond the tag, in which case one or both types in the tagged union might be `unit`.

A parenthesis type might be `unit + unit`, for example, to distinguish left and right. Most languages at least offer enumerated types for such cases.

Assembly and disassembly

For ordered pairs we have a constructor, to assemble an ordered pair of type $\sigma * \tau$ from its components, a left component of type σ and a right component of of type τ, and two destructors, to access the left and right components of a pair.
For a (binary) tagged union there's only one component, tagged either as left or right, and two constructors, to get a left $\sigma + \tau$ from a σ or a right $\sigma + \tau$ from a τ .
How do we access the values of a tagged union though? The destructor for a tagged union is a case statement, specifying how to get a value of some third type from either of the two possibilities.
For the points in plane example, distance might be calculated like

```
case pt of
    (x, y) => sqrt (x * x + y * y)
  | (r, theta) => r
end
```

assuming `pt` is a value of type `real * real + real * real` and `sqrt` is a function of type `real -> real` which computes square roots.

Option types

One popular use of tagged unions is for optional values. We can use `unit + `τ as a return type for functions that ought to return a value of type τ, but might not.
Some languages, e.g. Haskell, use the term maybe rather than option.
We could give division type `int -> int -> (unit + int)`, for example.
Any time you divide you have to handle the exceptional case and the regular case via a `case` statement.
This isn't the best way to handle possible division by zero, but the idea is useful in other contexts, e.g. looking up a value from a table might have type `table -> key -> (unit + value)`.
Here I assume the types `table`, `key` and `value` have already been defined.

Grammar rules

Adding tagged unions requires grammar rules, typing rules, and reduction rules.
For the grammar I'm not following any real language but using something vaguely SMLish:

```
expr ::= ... | "left" expr | "right" expr
         | "case" expr "of" var "=>" expr "|" var "=>" expr "end"
```

As usual ... represents the kinds of expressions we already had.
I've stopped writing whitespace explicitly. Let's assume the lexer strips it.
`left` and `right` are the syntax for tagging a value to make a tagged union.
In a real language we'd probably want more than two possible tags, just as we allow *n*-tuples rather than just ordered pairs, but two is simpler as an illustration.

Typing rules

The typing rules for our constructors are

$$\frac{\vdash e : \tau_1}{\vdash \texttt{left}\ e : \tau_1 + \tau_2} \qquad \frac{\vdash e : \tau_2}{\vdash \texttt{right}\ e : \tau_1 + \tau_2}$$

The typing rule for the destructor, the case statement, is

$$\frac{e_1 : \tau_1 + \tau_2 \qquad x : \tau_1 \vdash e_2 : \tau_3 \qquad y : \tau_2 \vdash e_3 : \tau_3}{\vdash \texttt{case}\ e_1\ \texttt{of}\ x\ \texttt{=>}\ e_2\ \texttt{|}\ y\ \texttt{=>}\ e_3\ \texttt{end} : \tau_3}$$

Once again we have context rules. We expect x to appear in the expresion e_2, so x's type must be known in order to find the the type of e_2, and similarly for y and e_3. It's important that e_2 and e_3 have the same type, so that the full expression has the same type no matter how e_1 was tagged.

Reduction rules

To evaluate `case` e_1 `of` x `=>` e_2 `|` y `=>` e_3 `end` we first need to evaluate e_1, which gives us a tag and a value. The tag is then used to determine whether to evaluate e_2 or e_3.
The value is substituted for all free occurences of the variable, x or y depending on which expression we're evaluating, in the expression.
The other expression is not evaluated, and generally can't be, because the value to be substituted would have the wrong type.

Unit type in tags

Tagged unions always have a value, but if the tag makes the value of unit type then that value is just `()`.
In this case practical languages may not force you to assign it to a variable.
More generally, they may not force you to introduce variables when not needed in the case expression.
Haskell and SML both allow you to write _ for an unused value.
You can even omit that when the type is `unit`.
Taking one of the two types in a tagged union to be `unit` gave us option (maybe) types. What would taking *both* to be `unit` do?
Now the value is never useful. We only have the tag, which gives us one bit of information: two possibilities.
A case statement for such a union allows us to choose which of two expressions to evaluate.
This sounds a lot like booleans and conditional statements. In fact booleans and conditionals are redundant in a language with tagged unions.

Redundant language features

We started with a simple language and gradually added new features. Some new features render old features redundant.

We might still keep the redundant features for convenience, but for proving things about the language they mostly just get in the way.

- Let statements are redundant once we have function definition and application. A let statement effectively just defines a function and then immediately applies it.
- Boolean operators are redundant once we have conditionals. e_1 `andalso` e_2 is just shorthand for `if` e_1 `then` e_2 `else false` and e_1 `orelse` e_2 is just shorthand for `if` e_1 `then true else` e_2.
- Boolean values and conditionals are redundant once we have tagged unions and a unit type. `bool` is shorthand for `unit + unit` and `true` and `false` are shorthand for `left ()` and `right ()`.

`if` e_1 `then` e_2 `else` e_3 is shorthand for `case` e_1 `of _ =>` e_2 `| _ =>` e_3 `end`.

A minimalist language could get by with types `int` and `unit`, the built-in arithmetic operators and relations, (anonymous) function definition and application, ordered pairs, and tagged unions.

Positions vs labels

If we only allow 2-tuples and binary tagged unions then positional notation is manageable. We can just remember which is the left or right component.
If we want to allow n-tuples or non-binary tagged unions then it becomes awkward, and it's convenient to have notation using labels instead.
Languages may provide both, or may regard positions as a special kind of label.
They may force you to name compound types, like `type point = int * int` or just allow you to refer to `int * int` without naming it.
No language I'm aware of handles this nicely, allowing positional or labelled references and named or anonymous types for both tuples and tagged unions.