MAU11602
Lecture 15
2026-02-25

Announcement

Since there was no tutorial this week you can turn in your current assignment along with the next one if you want.

Recursive data structures

A stack is a data structure. The stack is either empty, or consists of a head and a tail, which is also a stack. Pushing an item onto a stack creates a new stack with that item as head and the old stack as tail. Popping an item off a non-empty stack gives you the head of that stack and a new stack, the tail of the old stack.
This is the definition of a stack from Lecture 1.
It's clearly a recursive definition, since stacks are defined in terms of stacks.
It's also polymorphic, since I haven't specified what type the items in a stack have.
Last week we introduced recursive functions into our language. I didn't mention it, but recursive functions are allowed to be polymorphic, just like non-recursive functions.
We don't yet have recursive data types though. It's time to fill that gap.
Just as it's hard to work with recursive anonymous functions it's hard to work with recursive anonymous data types, so we need a way to name data types.
For practical reasons we'd like to have one anyway.

datatype examples

Each language is different, but the SML syntax for datatype definitions is `datatype`, followed by the name we're assigning to that type, then `=`, and then the definition, which is allowed to be recursive, i.e. refer to the name of the data type.

The second line of my calculator programme was

```
datatype token = Int of int | Oper of int * int -> int | Lparen | Rparen
```

This defines `token` to be a (non-binary) tagged union with four possible tags `Int`, `Oper`, Lparen, and `Rparen`.

The tag `Int` has values of type `int`. The tag `Oper` has values of type `int * int -> int`, i.e. functions from a pair of `int`s to an `int`. The tags `Lparen` and `Rparen` have values of `unit` type, but the language doesn't force us to make this explicit.

The default type of any tag is `unit`, so if we really wanted to define booleans as (binary) tagged unions of `unit`s we would write

```
datatype boolean = True | False
```

datatype examples, continued

The tag names are meant to be descriptive. You can probably guess that `Int`, `Oper`, `Lparen`, and `Rparen` refer to integers, (arithmetic) operators, left parentheses, and right parentheses, respectively.

Giving descriptive names is good practice when writing code, as a way to signal intent to (human) readers, including your future self. When reading code though, remember that it signals what the writer intended, which may not be what the code actually does!

The data type definitions

```
datatype token = Int of int | Oper of int * int -> int | Lparen | Rparen
datatype Boolean = True | False
```

are non-recursive. The third line of the calculator contains a recursive definition:

```
datatype 'a stack = Empty | Push of 'a * 'a stack
```

`'a` is just the way SML writes type variables, which are needed here because `stack` is polymorphic. This `'a` is the type of items in our stack.

There are two tags, `Empty` which has the default type of `unit`, and `Push`, whose type is an ordered pair of an item and a stack of items.

Tags and constuctors

Tags double as constructors. For each tag there is a function of the same name which constucts a tagged union with the given tag from a value of the corresponding type. For example, `Int 17` and `Oper (fn (i, j) => i + j)` are two expressions of type `token`, the first tagged `Int` and the second tagged `Oper`, according the the definition `datatype token = Int of int | Oper of int * int -> int | Lparen | Rparen` In theory we ought to write `Lparen ()` or `Rparen ()` to obtain tokens tagged `Lparen` and `Rparen` but the `()` doesn't convey any useful information and so SML allows us to omit it.

Similarly the stack constructors are `Empty` and `Push` according to

`datatype 'a stack = Empty | Push of 'a * 'a stack`

So to obtain a stack of size 2 with 17 on top and 42 below it we would write

`Push (17, Push (42, Empty))`

Functions on recursive data types

To operate on a recursive data type we need recursive functions. For example,

```
fun size Empty = 0
  | size (Push (i, s)) = 1 + size s
```

is a function from stacks to integers. `size (Push (17, Push (42, Empty)))` evaluates to 2, as expected.

This function is inefficient in its memory usage. A more efficient, but less readable, version is

```
fun size_aux acc Empty = acc
  | size_aux acc (Push (i, s)) = size_aux (acc + 1) s
fun size s = size_aux 0 s
```

In lecture I'll generally sacrifice efficiency for readability, unless the efficiency gain is huge, as in the Fibonacci example.

`size_aux` is actually a curried function of type `int -> 'a stack -> int` so we could write `val size = size_aux 0` instead. There's no need for the `s` on each side.

Appending

What does the following function do?

```
fun append (Empty, t) = t
  | append (Push (i, s), t) = Push (i, append (s, t))
```

It gives a stack with all the items in `s` followed by all the items in `t`, in order.

The corresponding function on lists is generally called append, but stacks are lists.

We prove properties of recursively defined functions on recursive data types by induction. For example, we expect `size (append (s, t))` to be equal to `size s + size t`. Is this true?

It's certainly true when `s` is `Empty` because `append (Empty, t) = t` and `0 + size t = size t`.

Also, if it's true for a given `s` and `t` then `size (append (Push (i, s), t))` is equal to `size Push(i, s) + size t` for any `i`.

This true because `size (Push (i, s)) = 1 + size s` and integer addition is associative.

This is a proof by induction, but it's not ordinary induction on the integers but what's called *structural induction*.

More appending

We expect append to be associative, i.e. that `append (s, append (t, u))` is equal to `append (append (s, t), u)`. Is this true?

It's true if `s` is `Empty`:

```
append (Empty, append (t, u)) = append (t, u)
                              = append (append (Empty, t), u)
```

To complete the structural induction we need to show that if

```
append (s, append (t, u)) = append (append (s, t), u)
```

for some `s`, `t`, and `u` then

```
append (Push (i, s), append (t, u)) = append (append (Push (i, s), t), u)
```

for all `i`.

If you try to prove this directly you run into trouble. You first need to prove some simpler properties, like

```
Push (i, append (s, t)) = append (Push (i, s), t)
```

The integers are redundant!

Once we have recursive data types the original type `int` becomes redundant.
Analogous to how we defined stacks and append we can define

```
datatype Integer = Zero | S of Integer
fun add (Zero, y) = y
  | add (S x, y) = S (add (x, y))
```

Remember, human readable names reflect intent, not necessarily reality! What I've defined here actually mimics the natural numbers, not the integers!
We can prove properties of the natural numbers and addition, like associativity, by structural induction.
In fact the proof is essentially the same. We can think of the natural numbers as a simplified version of unit stacks.
Induction on the natural numbers is really just a special case of structural induction.
Not all properties of natural numbers are inherited from stacks. Addition is associative and commutative, while append is just associative.

A representation of integers

I want something which mimics the integers, not the natural numbers, but we can build it from the naturals, e.g.

```
datatype Natural = Zero | S of Natural
fun addNat (Zero, y) = y
  | addNat (S x, y) = S (addNat (x, y))
datatype Integer = P of Natural | N of Natural
fun addInt (x, y) = ???
```

Here the tags `P` and `N` represent positive, or really non-negative, and negative, or really non-positive. This has the disadvantage that 0 is represented by both `P Zero` and `N Zero`. Also, defining addition is painful.

A better representation

There are ways to avoid the non-uniqueness of the representation of zero, or we could just embrace it and define integers as ordered pairs of naturals, to be interpreted as the left element minus the right element:

```
datatype Natural = Zero | S of Natural
fun addNat (Zero, y) = y
  | addNat (S x, y) = S (addNat (x, y))
type Integer = Natural * Natural
fun addInt ((u, v), (x, y)) = (addNat (u, x), addNat(v, y))
fun subInt ((u, v), (x, y)) = (addNat (u, y), addNat(v, x))
fun eqInt ((u, v), (x, y)) = (addNat (u, y) = addNat (v, x))
```

We need `eqInt` to test whether two `Integers` are equal because ordinary = would test whether the ordered pairs have the same left and right elements, which is not what we want.

Efficiency

Of course no one would really represent integers this way in a real programming language. Computers are designed to deal with integers and using their built-in representation and operations will be much faster.
Similarly, most languages provide a built-in implementation of stacks, typically called lists, and the built-in operations may be faster than the ones given earlier.
The method used to define stacks and stack operations applies more generally though.
If our language provides lists but not trees then we can define (binary) trees by
`datatype 'a tree = Empty | Join of 'a tree * 'a * 'a tree`
and define recursive functions on trees and prove their properties by structural induction.
Your language of choice may provide a built-in implementation of binary trees, which may be more efficient than rolling your own, but there will always be some recursive structure it doesn't provide.