5.1 Peano numbers

Italian mathematician Giuseppe Peano introduced a way of reasoning about whole numbers that includes the following axioms:

Axiom 7 *There exists a whole number 0.*

Axiom 8 *Every whole number n has a successor,* `succ` n. *successor*

Axiom 9 *No whole number has 0 as its successor.*

Axiom 10 *If $a, b \in \mathbb{W}$, then $a = b$ iff* `succ` $a =$ `succ` b.

If we interpret *successor* to mean "one more than," these axioms allow us to define whole numbers recursively (called *Peano numbers*); a whole number is *Peano numbers*

- zero, or
- one more than another whole number.

As with the `fun` form for declaring functions, the `datatype` form may refer to the type being defined.

```
- datatype wholeNumber = Zero | OnePlus of wholeNumber;
```

When we wrote self-referential functions or algorithms, they were defined as processes which contained smaller versions of themselves. They were recursive *verbs*. A self-referential datatype is a recursive *noun*. The recursive part establishes a pattern for generating every possible whole number. For example,

5 is a whole number because it is the successor of
4, which is a whole number because it is the successor of
3, which is a whole number because it is the successor of
2, which is a whole number because it is the successor of
1, which is a whole number because it is the successor of
0, which is a whole number by Axiom 7.

In ML,

```
- val five = OnePlus(OnePlus(OnePlus(OnePlus(OnePlus(Zero)))));

val five = OnePlus (OnePlus (OnePlus (OnePlus (OnePlus Zero)))) : wholeNumber

- val six = OnePlus(five);

val six = OnePlus (OnePlus (OnePlus (OnePlus (OnePlus (OnePlus Zero))))) : wholeNumber
```

Finding the successor of a number is just a matter of tacking "`OnePlus`" to the front, a process easily automated.

```
- fun succ(num) = OnePlus(num);
```

Conversion from an int to a wholeNumber is a recursive process—the base case, 0, can be returned immediately; for any other case, we add one to the wholeNumber representation of the int that comes before the one we are converting. (Negative ints will get us into trouble.)

```
- fun asWholeNumber(0) = Zero
=   | asWholeNumber(n) = OnePlus(asWholeNumber(n-1));

val asWholeNumber = fn : int -> wholeNumber
```

predecessor

Notice how subtracting one from the int and adding one to the resulting wholeNumber balance each other off. Opposite the successor, we define the *predecessor* of a natural number n, `pred`(n), to be the number of which n is the successor. From Axiom 10 we can prove that the predecessor of a number, if it exists, is unique; Axiom 9 says that 0 has no predecessor. Pattern matching makes stripping off a "`OnePlus`" easy:

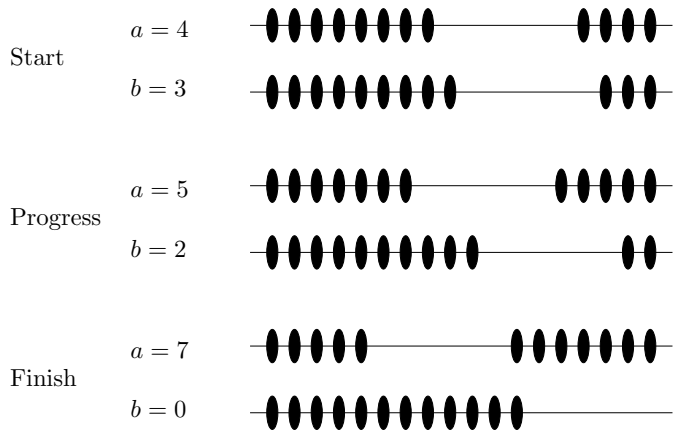
```
- fun pred(OnePlus(num)) = num;

Warning: match nonexhaustive
           OnePlus num => ...

val pred = fn : wholeNumber -> wholeNumber
```

The "match nonexhaustive" warning is okay; we truly want to leave the operation undefined for `Zero`. Using the function on `Zero`, rather than failing to define it for `Zero`, would be the mistake.

```
- val three = asWholeNumber(3);

val three = OnePlus (OnePlus (OnePlus Zero)) : wholeNumber

- pred(three);

val it = OnePlus (OnePlus Zero) : wholeNumber

- pred(Zero);

uncaught exception nonexhaustive match failure
```

Now we can start defining arithmetic recursively. `Zero` will always be our base case; anything we add to `Zero` is just itself. For other numbers, picture an abacus. We have two wires, each with a certain number of beads pushed up. At the end of the computation, we want one of the wires to contain our answer. Thus we push down one bead from the other wire, bring up one bead on the answer wire, and repeat until the other wire has no beads left. In other words, we define addition similarly to our recursive gcd lemmas from Section 3.10.

$$\begin{aligned} 0 + b &= b \\ a + 0 &= a \\ a + b &= (a + 1) + (b - 1) \quad \text{if } b \neq 0 \end{aligned}$$



In ML,

```
- fun plus(Zero, num) = num
=   | plus(num, Zero) = num
=   | plus(num1, OnePlus(num2)) = plus(OnePlus(num1), num2);
```

Examine for yourself the similar structure of `isLessThanOrEqualTo`. Keep in mind that recursively-defined predicates have two base cases, one true and one false. Here the first and second parameters are in a survival contest; they repeatedly shed a `OnePlus`, and the first one reduced to `Zero` loses.

```
- fun isLessThanOrEq(Zero, num) = true
=   | isLessThanOrEq(num, Zero) = false
=   | isLessThanOrEq(OnePlus(num1), OnePlus(num2)) = isLessThanOrEq(num1, num2);
```

What happens if the two numbers we are comparing are equal? In other words, what if they both strip down to zero simultaneously? The result should be `true`, since it is true that $7 \leq 7$. The expression `isLessThanOrEq(Zero, Zero)` will match to the first pattern, so it will indeed return `true`. Notice how important the order of the first two patterns is. If we were to swap them, we would have

```
- fun isLessThan(num, Zero) = false
=   | isLessThan(Zero, num) = true
=   | isLessThan(OnePlus(num1), OnePlus(num2)) = isLessThan(num1, num2);
```

For subtraction, we observe

$$\begin{aligned} a - 0 &= a \\ a - b &= (a-1) - (b-1) \quad \text{if } a \neq 0 \text{ and } b \neq 0 \end{aligned}$$

In ML,

```
- fun minus(num, Zero) = num
=   | minus(OnePlus(num1),  OnePlus(num2)) = minus(num1, num2);


Warning: match nonexhaustive
          (num,Zero) => ...
          (OnePlus num1,OnePlus num2) => ...

val minus = fn : wholeNumber * wholeNumber -> wholeNumber
```

This rightly leaves the pattern `minus(Zero, OnePlus(num))` undefined. Finally, conversion back to `int` is just a literal interpretation of the identifiers we gave to the constructor expressions.

```
- fun asInt(Zero) = 0
=   | asInt(OnePlus(num)) = 1 + asInt(num);
```

This section received much inspiration from Felleisen and Friedman[7].

Exercises

5.1.1 Write a function `isEven` for the wholeNumber type.

5.1.2 Write a function `multiply` for the wholeNumber type.

5.1.3 Write a function `divide`, performing integer division, for the wholeNumber type. (Hint: The Division Algorithm from Exercise 3.10.5 is a good place to start.)

5.1.4 Write a function `modulo`, performing the `mod` operation, for the wholeNumber type.

5.1.5 Write a function `gcd`, computing the greatest common divisor, for the wholeNumber type.

Natural numbers that are powers of 2 can be defined as

- 1, or
- 2 times a power of 2.

We can represent this in ML as

```
- datatype powerOfTwo = One | TwoTimes of powerOfTwo;
```

5.1.6 Write a function `multiply`, multiplying two powerOfTwos to get another powerOfTwo.

5.1.7 Write a function `asPowerOfTwo`, converting an int to the nearest powerOfTwo less than or equal to it.

5.1.8 Write a function `asInteger`, converting a powerOfTwo to an equivalent int

5.1.9 Write a function `logBase2`, computing an int base 2 logarithm from a powerOfTwo (that is, the type of this function should be powerOfTwo -> int).

If ML did not come with a list construct, we could define our own using the datatype

```
- datatype  homemadeList =
=          Null | Cons of int * homemadeList;
```

5.1.10 Write a function `head` for homemadeList, equivalent to the ML primitive `hd`.

5.1.11 Write a function `tail` for homemadeList, equivalent to the ML primitive `tl`.

5.1.12 Write a function `cat` to concatenate two homemadeLists. equivalent to the ML primitive `@`.

5.1.13 Write a function `isElementOf` for homemadeList.

5.1.14 Write a function `makeNoRepeats` for homemadeList.

5.1.15 Write a function `sum` for homemadeList.

5.1.16 Write a function `map` for homemadeList, like the `map` function of Section 6.5.

5.2 Trees

Trees are useful mechanisms for organizing information. They are used in many contexts and fields of study. You all have seen such common uses as genealogical trees and the phone trees used by customer service numbers.

tree
node
link
root
leaf
internal node
parent
child
full binary tree

A *tree* is a collection of *nodes* and *links*. We will take *node* to be a primitive. A link is an ordered pair of nodes, and each node is the second node in exactly one link, except for one special node which is not the second node in any link; we call this special node the *root* of the tree. Nodes that are not the first node in any link are called *leaves*; all other nodes (nodes which are the first node in at least one link) are called *internal nodes*. To avoid the awkward phrasing of "being a the first node in a link," we will say that there exists a link *from* one node *to* another, and we will call the first node the *parent* and the other the *child*.

Our interest is in a specific kind of tree, a *full binary tree* which is either

- a single node with no links, or
- a node together with links to two other full binary trees

(Technically, the designation that it is a *binary* tree means that any node may have at most two children; *full* eliminates the case of a node having exactly one child, that is, it requires internal nodes to have the full two children possible or none at all.)

We can model this in ML with this datatype: