Some Hope examples

Ross Paterson

April 18, 2000

1 The Factorial Function

First, the usual recursive version:

```
dec fact : num -> num;
--- fact 0 <= 1;
--- fact n <= n*fact(n-1);
```

Note that Hope uses best-fit pattern matching, so that the second clause is chosen only if `n` is non-zero. Moreover, swapping the clauses doesn't change the behaviour of the program.

```
fact 7;
```

Second, a non-recursive version using some libraries:

```
uses lists, range;

dec fact : num -> num;
--- fact n <= product (1..n);

1..7;
product [1, 2, 3, 4, 5, 6, 7];
fact 7;
```

2 Fibonacci Numbers

First, the usual recursive version:

```
dec fib : num -> num;
--- fib 0 <= 1;
--- fib 1 <= 1;
--- fib(n+2) <= fib n + fib(n+1);

fib 10;
```

```
uses list, range;
1..10;
map fib (0..10);
```

This is notoriously inefficient. A much faster version, using infinite lists, is:

```
uses lists;

dec fibs : list num;
--- fibs <= fs whererec fs == 1::1::map (+) (tail fs||fs);
```

Here `||` is the 'zip' function. The infinite list of factorials `fs` is defined in terms of itself, as a circular structure (by the `whererec`), so that previously calculated values for smaller arguments are reused.

```
front(11, fibs);
```

3 Breadth-first tree traversal

For this example, we shall define trees as 'rose trees':

```
type rose_tree alpha == alpha # list(rose_tree alpha);
```

A tree consists of a label and a list of sub-trees. (Note that this particular implementation permits recursive type synonym definitions.)

We wish to construct the list the labels in breadth-first order. The method used is:

1. Construct an infinite list of levels of the tree:
   - The first level comprises just the input tree.
   - Each other level consists of all the children of the previous level.
2. Truncate the list before the first empty level.
3. Concatenate the levels.
4. Extract the labels of the trees.

This is represented directly, using '`.`' to stand for reversed function application.

```
uses lists, functions, products;

dec bf_list : rose_tree alpha -> list alpha;
--- bf_list t <= [t].
        iterate (concat o map snd).
        front_with (/= []).
        concat.
        map fst;
```

```
bf_list (1, [(2, [(5, []),
                  (6, [(10, [])])
                 ]),
             (3, [(7, [])]),
             (4, [(8, [(11, [])]),
                  (9, [])
                 ])
            ]);
```

4 Symbolic Boolean Expressions

We define a simple variable type:

```
type var == char;
```

We represent Boolean expressions by trees:

```
infixrl IMPLIES: 1;
infix   OR: 2;
infix   AND: 3;

data bexp ==    VAR var ++ bexp AND bexp ++ bexp OR bexp ++
                NOT bexp ++ bexp IMPLIES bexp;
```

We define an *assignment* as a list assigning Boolean values to variables:

```
type assignment == list(var # truval);
```

Now define a function to evaluate a Boolean expression with respect to an assignment of Boolean values to the variables it contains:

```
dec lookup : alpha # list(alpha#beta) -> beta;
--- lookup(a, (key, datum)::rest) <=
        if a = key then datum else lookup(a, rest);

dec evaluate : bexp # assignment -> truval;
--- evaluate(VAR v, a) <= lookup(v, a);
--- evaluate(e1 AND e2, a) <= evaluate(e1, a) and evaluate(e2, a);
--- evaluate(e1 OR e2, a) <= evaluate(e1, a) or evaluate(e2, a);
--- evaluate(NOT e, a) <= not(evaluate(e, a));
--- evaluate(e1 IMPLIES e2, a) <=
        not(evaluate(e1, a)) or evaluate(e2, a);

evaluate (VAR 'a' AND VAR 'b' OR VAR 'c',
          [('a', false), ('b', true), ('c', false)]);

evaluate (VAR 'a' AND VAR 'b' OR VAR 'c',
          [('a', true), ('b', true), ('c', false)]);
```

We intend to use this function in a function to test whether any Boolean expression is a tautology, *i.e.* is true for any assignment of its variables. First, we need a function to extract the list of variables in an expression. The following function merges two ordered lists to form an ordered list without duplicates:

```
dec merge : list alpha # list alpha -> list alpha;
--- merge([], l) <= l;
--- merge(l, []) <= l;
--- merge(x1::l1, x2::l2) <=
        if x1 = x2 then x1::merge(l1, l2)
        else if x1 < x2 then x1::merge(l1, x2::l2)
                        else x2::merge(x1::l1, l2);
```

The next function returns the ordered list of variables in a Boolean expression, without duplicates:

```
dec vars : bexp -> list var;
--- vars(VAR v) <= [v];
--- vars(e1 AND e2) <= merge(vars e1, vars e2);
--- vars(e1 OR e2) <= merge(vars e1, vars e2);
--- vars(NOT e) <= vars e;
--- vars(e1 IMPLIES e2) <= merge(vars e1, vars e2);

vars (VAR 'a' AND VAR 'b' OR VAR 'a');

vars ((VAR 'a' IMPLIES VAR 'b' IMPLIES VAR 'c') IMPLIES
        (VAR 'a' IMPLIES VAR 'b') IMPLIES
        VAR 'a' IMPLIES VAR 'c');
```

Now we want to generate the the list of all possible assignments to a list of variables:

```
uses list;

dec interpretations: list var -> list assignment;
--- interpretations [] <= [[]];
--- interpretations(h::t) <=
        (map ((h, false)::) l <> map ((h, true)::) l)
        where l == interpretations t;

interpretations "a";

interpretations "ab";

interpretations "abc";
```

Finally, we put all these together. An expression is a tautology if it evaluates to true for all possible assignments to the variables it contains:

```
dec tautology: bexp -> truval;
--- tautology e <=
        foldr(true, (and))
                (map evaluate
                        (dist(e, interpretations(vars(e)))));

tautology (VAR 'a' AND VAR 'b' OR VAR 'a');

tautology ((VAR 'a' IMPLIES VAR 'b' IMPLIES VAR 'c') IMPLIES
                (VAR 'a' IMPLIES VAR 'b') IMPLIES
                VAR 'a' IMPLIES VAR 'c');
```