Constructive Logic (15-317), Fall 2012
Assignment 9: Dependent Representations

Carlo Angiuli (`cangiuli@cs`)

Out: Tuesday, November 20, 2012
Due: Thursday, November 29, 2012 (before class)

In this assignment, you'll explore the power of dependently-typed programming and representations in Elf. First, you will write some short programs over dependently-typed data structures whose invariants are enforced by their types. Then, you'll use higher-order representations to code up logical constructs from earlier in the semester, and you will prove adequacy of your encoding.

The written portion of this assignment (in Section 3) should be typeset in LaTeX or written *neatly* by hand.

Your code should be submitted via AFS by copying it to the directory

```
/afs/andrew/course/15/317/submit/<userid>/hw09
```

where `<userid>` is replaced with your Andrew ID. Your solutions should work in the version of Twelf installed in the course directory.

1 Running Twelf

To run Twelf, execute

```
/afs/andrew/course/15/317/bin/twelf-server
```

from any Andrew machine. Alternatively, you may download and install a copy locally following the directions at `http://twelf.org`, but please test your code a final time on an Andrew machine to ensure it works there, as that is what we will use to grade.

You can load a file `foo.elf` at the prompt by typing

```
loadFile foo.elf
```

To issue queries, type `top` and enter predicates at the prompt.

2 Dependent Data (15 points)

The code in this section can be found in `treelist.elf`. Consider the following Elf definition of length-indexed lists and the corresponding append function:

```
nat : type.
0 : nat.
s : nat -> nat.

plus : nat -> nat -> nat -> type.
plus/0 : plus 0 N N.
plus/s : plus (s N) M (s P)
          <- plus N M P.

list : nat -> type.
nil : list 0.
cons : nat -> list N -> list (s N).

append : list N -> list M -> plus N M P -> list P -> type.
%mode append +X1 +X2 +X3 -X4.
append/nil : append nil L plus/0 L.
append/cons : append (cons X L) L' (plus/s Dplus) (cons X L'')
               <- append L L' Dplus L''.
%worlds () (append _ _ _ _).
%total L (append L _ _ _).
```

Now suppose we have an ordinary, simply-typed representation of trees:

```
tree : type.
leaf : tree.
node : nat -> tree -> tree -> tree.
```

Your task is to convert trees to lists, following the specification that the resulting list's length should equal the number of elements in the tree.

Task 1 (5 points). Write a predicate `nelts : tree -> nat -> type.` to count the number of elements in a tree.

Task 2 (10 points). Write a predicate `flatten : {T:tree} nelts T N -> list N -> type.` creating a list from the elements of the tree seen in depth-first order.

3 Sum Types and Adequacy (25 points)

The code in this section can be found in `stlc.elf`. So far, we've seen how to encode the syntax of the simply-typed lambda calculus with a base type `o` with a single inhabitant b:

$$\begin{array}{rcl} \text{types } \tau & ::= & o \mid \tau \to \tau \\ \text{expressions } e & ::= & b \mid e\ e \mid \lambda x{:}\tau.e \end{array}$$

Suppose we add a sum (disjunction) type to the language:

$$\begin{array}{rcl} \text{types } \tau & ::= & o \mid \tau \to \tau \mid \tau + \tau \\ \text{expressions } e & ::= & b \mid e\ e \mid \lambda x{:}\tau.e \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}(e, x.e, x.e) \end{array}$$

Recall the inference rules for sum types:

$$\frac{e : \tau_1}{\mathsf{inl}\ e : \tau_1 + \tau_2}\ {+I_1} \qquad \frac{e : \tau_2}{\mathsf{inr}\ e : \tau_1 + \tau_2}\ {+I_2} \qquad \frac{e : \tau_1 + \tau_2 \quad \begin{array}{c}[x : \tau_1] \\ \vdots \\ e_1 : \tau\end{array} \quad \begin{array}{c}[y : \tau_2] \\ \vdots \\ e_2 : \tau\end{array}}{\mathsf{case}(e, x.e_1, y.e_2) : \tau}\ {+E}$$

(NOTE: These rules are provided to help with intuition while encoding syntax; you *do not* have to encode them.)

Task 3 (10 points)**.** Encode the syntax of the simply-typed lambda calculus with sum types.

Task 4 (15 points)**.** To prove adequacy of syntax, we need an embedding function $\ulcorner - \urcorner$ from first-order logic syntax to LF terms and a de-embedding function $\llcorner - \lrcorner$ in the other direction that compose to the identity. Define these two functions for your encoding.