Constructive Logic (15-317), Fall 2012
Assignment 10: Bracket Abstraction in Twelf

Carlo Angiuli (cangiuli@cs)

Out: Thursday, November 29, 2012
Due: Thursday, December 6, 2012 (before class)

By now, you've used Elf to encode arithmetic functions over dependent data and the syntax of a deductive system, the simply-typed lambda calculus. This assignment will continue on the path of encoding deductive systems. You will encode another such system, the SKI combinator calculus, and give a translation between simply typed lambda calculus terms and combinator terms.

Please write your code starting from the file `bracket.elf` released on the website. Submit the written portions of this assignment as comments in your code.

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

`/afs/andrew/course/15/317/submit/<userid>/hw10`

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

1 Combinators (20 points)

In the lambda calculus, there are two ways to form compound terms: lambda abstraction and application. These correspond to the introduction and elimination rules for $\supset$ in natural deduction. The SKI combinator calculus is a system that can be used to construct the same sorts of programs, but where application is the only term constructor. To make up for lacking λ, it has three constants (called combinators) whose types happen to yield the same expressive power:

- $\mathsf{S} : (\tau \to \sigma \to \rho) \to (\tau \to \sigma) \to (\tau \to \rho)$
- $\mathsf{K} : \tau \to (\sigma \to \tau)$
- $\mathsf{I} : \tau \to \tau$

The file `bracket.elf` contains an encoding of the following grammar for combinator terms.

$$C \quad ::= \quad \mathsf{b} \mid \mathsf{S} \mid \mathsf{K} \mid \mathsf{I} \mid C@C$$

The typing judgment $C : \tau$ on combinator terms is defined as follows.

$$\frac{}{\mathsf{I} : \tau \to \tau}\ \mathsf{of/I} \qquad \frac{}{\mathsf{K} : \tau \to (\sigma \to \tau)}\ \mathsf{of/K}$$

$$\frac{}{\mathsf{S} : (\tau \to \sigma \to \rho) \to (\tau \to \sigma) \to (\tau \to \rho)}\ \mathsf{of/S}$$

Writing the combinator application of function C_1 to argument C_2 as $C_1@C_2$, the typing rule is as usual:

$$\frac{C_1 : \tau \to \tau' \qquad C_2 : \tau}{C_1@C_2 : \tau'}\ \mathsf{cof/app}$$

Finally, we include a combinator base term b to match the base term of the lambda calculus, with the same typing rule.

$$\frac{}{b : o}\ \mathsf{cof/b}$$

Task 1 (5 points)**.** Give combinator terms inhabiting the following types:

1. $\rho \to (\tau \to \sigma \to \tau)$
2. $(\tau \to \rho) \to (\tau \to \sigma \to \rho)$
3. $\tau \to \tau$, without using I.

Task 2 (5 points)**.** Encode the static semantics of combinators in Twelf according to the above rules as a judgment `cof : comb -> tp -> type`.

Combinators, like lambda terms, have a dynamic semantics, but it works a little differently. A lambda term can be partially applied, whereas each combinator must be applied to *all* of its arguments before it fires.

$$\frac{}{\mathsf{I}@C \mapsto C}\ \mathsf{cstep/I} \qquad \frac{}{\mathsf{K}@A@B \mapsto A}\ \mathsf{cstep/K}$$

$$\frac{}{\mathsf{S}@A@B@C \mapsto (A@C)@(B@C)}\ \mathsf{cstep/S}$$

There is one compatibility rule.

$$\frac{A \mapsto A'}{A@B \mapsto A'@B}\ \mathsf{cstep/app}$$

Task 3 (10 points)**.** Encode the dynamic semantics of combinators in Twelf as a judgment `cstep : comb -> comb -> type`.

2 Translation from STLC (20 points)

For this assignment, we will define one direction of correspondence between these systems: lambda terms to combinators. Let $\mathsf{tr}(e) = C$ mean that the lambda term e translates to a combinator term C.

Translating the base term and application is straightforward, since we have those constructs in both systems.

$$\begin{aligned}\mathsf{tr}(\mathsf{b}) &= \mathsf{b} \\ \mathsf{tr}(e_1\ e_2) &= \mathsf{tr}(e1)@\mathsf{tr}(e2)\end{aligned}$$

Translating a lambda term is where all the action is. For this translation, we need to define an auxiliary function $\langle - \rangle$.

$$\mathsf{tr}(\lambda x{:}\tau.e) = \langle x \rangle \mathsf{tr}(e)$$

$\langle - \rangle -$ is the *bracket abstraction* function. We can read $\langle x \rangle C$ as encoding the *abstraction* of combinator variable x from combinator term C. But, of course, the grammar of combinators doesn't include variables – they are purely an auxilliary notion, and the process of bracket abstraction must end with their absence.

Bracket abstraction is defined as follows.

$$\begin{aligned}\langle x \rangle x &= \mathsf{I} \\ \langle x \rangle C &= \mathsf{K}@C && (x \text{ not free in } C) \\ \langle x \rangle (C_1 @ C_2) &= (\mathsf{S}@(\langle x \rangle C_1))@(\langle x \rangle C_2)\end{aligned}$$

Note that, critically, the second rule covers combinator variables different from x.

Task 4 (10 points)**.** Encode $\langle - \rangle -$ as a Twelf judgment

```
bracket : (comb -> comb) -> comb -> type.
```

Task 5 (10 points)**.** Encode $\mathsf{tr}(-) = -$ as a Twelf judgment

```
translate : term -> comb -> type.
```