# Homework 2: $\lambda$-Calculus

Due: September 28, 2004

When in doubt, fully parenthesize the $\lambda$-calculus expression and the types you are working with.

## 1 The Simply Typed $\lambda$-calculus

When we discussed type checking in class, we started with a type, and then gave a proof that a term of the simply typed calculus had that type. It is also possible to use the type checking rules of the simply typed lambda calculus for type inference, as in Haskell. One can either guess the type and prove it, or use the rules to figure out the type. For example, to figure out the type of an application, figure out the type of the procedure and the argument, and then use the return type of the procedure's arrow type. To figure out the type of a $\lambda$-expression, figure out the type of the body, and connect that to the type of the argument with an arrow.

For example, if we assume that $g : (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$, then we can derive the type of $g(\lambda x : \mathbf{o} \,.\, x)$ as follows. For the proof, to save space, let $H$ be the type environment $g : (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}$.

$$\dfrac{\dfrac{\dfrac{}{H, x : \mathbf{o} \vdash x : \mathbf{o}\ [\text{var}]}}{H \vdash g : (\mathbf{o} \to \mathbf{o}) \to \mathbf{o}\ [\text{var}]} \quad \dfrac{}{H \vdash (\lambda x : \mathbf{o} \,.\, x) : \mathbf{o} \to \mathbf{o}\ [\to \text{Intro}]}}{H \vdash (g(\lambda x : \mathbf{o} \,.\, x)) : \mathbf{o}\ [\to \text{Elim}]}$$

1. (15 points) Derive and prove the type of

   $$\lambda f : \mathbf{o} \to \mathbf{o} \,.\, \lambda x : \mathbf{o} \,.\, f(x)$$

   in the empty type environment.

2. (30 points) Derive and prove the type of

   $$\lambda x : \mathbf{o} \to \mathbf{o} \to \mathbf{o} \,.\, \lambda y : \mathbf{o} \to \mathbf{o} \,.\, \lambda z : \mathbf{o} \,.\, xz(yz)$$

   in the empty type environment. (If the tree gets too bushy, use lemmas.)

3. (15 points; extra credit) Let ? be the type of the term in the previous problem; does the following term have a type (in the empty type environment)?

   $$(\lambda S :? \,.\, \lambda K : \mathbf{o} \to \mathbf{o} \to \mathbf{o} \,.\, S\ K\ K)(\lambda x : \mathbf{o} \to \mathbf{o} \to \mathbf{o} \,.\, \lambda y : \mathbf{o} \to \mathbf{o} \,.\, \lambda z : \mathbf{o} \,.\, xz(yz))(\lambda x : \mathbf{o} \,.\, \lambda y : \mathbf{o} \,.\, x)$$

   If so state and prove the type. If not, clearly state why it does not check.

## 2 Untyped $\lambda$-calculus

The following is about the untyped $\lambda$-calculus.

The following problems use the combinators defined below.

$$\begin{aligned} I &\stackrel{\text{def}}{=} \lambda x \,.\, x \\ K &\stackrel{\text{def}}{=} \lambda x \,.\, \lambda y \,.\, x \\ S &\stackrel{\text{def}}{=} \lambda x \,.\, \lambda y \,.\, \lambda z \,.\, x\ z\ (y\ z) \\ B &\stackrel{\text{def}}{=} S\ (K\ S)\ K. \end{aligned}$$

4. (10 points) Prove that, for all terms $M$ and $N$, $KMN \stackrel{\beta}{\Rightarrow}^* M$, where $\stackrel{\beta}{\Rightarrow}^*$ is the reflexive-transitive closure of beta-reduction, $\stackrel{\beta}{\Rightarrow}$.

5. Using the equational rules, formally derive the normal forms (if any) for the following. (Hint: it's convenient to use an editor to do this kind of work, as you can write out an expression, copy it, and apply the beta or eta rules without too much hand copying.) For example,

```
 (\f.(\y.z)f) ((\x.xx) (\x.xx))
==> <eta>
 (\y.z) ((\x.xx) (\x.xx))
==> <beta>
 z
```

   (a) (10 points)
   $(S\ K\ K)\ x$

   (b) (20 points)
   $B\ x\ y\ z$

   (c) (25 points)
   $(S(BBS)(KK))xyz$

6. (15 points; extra credit) Prove that application in the lambda calculus is not associative.

7. (100 points; extra credit, problem due to Barendregt) Suppose a symbol of the $\lambda$-calculus alphabet is always 0.5 cm wide. Write down a $\lambda$-calculus term with length less than 20 cm having a normal form with length at least $10^{10^{10}}$ lightyear. (Note: the term must have a normal form.) The speed of light is $c = 3 \times 10^{10}$ cm/sec.

# 3 Type Checking in Haskell

8. (50 points) In Haskell, write the function

```
> infer :: LambdaTerm -> Maybe Type
```

   which infers the types of simply typed lambda calculus terms. You will do this by filling in the missing definition of `infer` (and any needed helping functions) in the file `TypedLambdaCalculus.lhs` found in the course homework directory, where you will also find the modules it uses: `ParserFunctions` and `LexerTools`.

   Here are some examples of an interactive session:

```
TypedLambdaCalculus> read_type_print
lambda-typing> ((\x:(o->o).x) (\x:o.x))
Just (o -> o)
lambda-typing> x
Nothing
lambda-typing> (\x:o.x)
Just (o -> o)
lambda-typing> (\x:o.(\y:o.(\z:o.x)))
Just (o -> (o -> (o -> o)))
lambda-typing> (\x:o.(\y:o.(\z:o.x)))
Just (o -> (o -> (o -> o)))
lambda-typing> ((\x:o.x)(\x:o.x))
```

```
Nothing
lambda-typing> ((\x:o.x)z)
Nothing
lambda-typing> ((\x:o.x)x)
Nothing
lambda-typing> (f(\x:o.x))
Nothing
lambda-typing> (((\x:o.x)(\x:o.x))(\x:o.x))
Nothing
lambda-typing> ((\x:o.x) ((\x:o.x) (\x:o.x)))
Nothing
lambda-typing> ((\x:o.(\y:o.(\z:o.x))) (\i:o.i))
Nothing
lambda-typing> ((\f:(o->o).(\y:o.(f y))) (\i:o.i))
Just (o -> o)
```

Hand in a printout of your code and output from its testing.

9. (50 points; extra credit) Make your solution to the previous problem both type check, and then evaluate the expression to a normal form.