CSCI 490 problem set 6

Due Tuesday, March 1

Revision 1: compiled Tuesday 23rd February, 2016 at 21:21

Rubric

For full credit, your solutions should demonstrate a proficient understanding of the following topics:

- Simply typed lambda calculus: 4 points
- Y combinator: 4 points
- Type inference: 4 points
- Curry-Howard isomorphism: 4 points

An extra 4 points will be awarded for completeness.

The Simply-Typed Lambda Calculus

This section provides a reference for the simply-typed λ-calculus extended with product and sum types, as discussed in class.

Syntax

Types are denoted by the metavariables φ or ψ and are defined by the following recursive grammar:

or occasionally χ or whatever other lowercase Greek letter I feel like using

$$\varphi, \psi ::= A \mid \varphi \to \psi \mid \varphi \times \psi \mid \varphi + \psi$$

That is, a type is either a base type (denoted by a capital letter like A, B, ...), a function type $\varphi \to \psi$, a product type $\varphi \times \psi$, or a sum type $\varphi + \psi$.

The syntax of terms is given by

$$\begin{aligned} t ::= {} & x \mid \lambda x : \varphi.\, t \mid t_1\ t_2 \\ & \mid (t_1, t_2) \mid \mathsf{fst}\ t \mid \mathsf{snd}\ t \\ & \mid \mathsf{left}\ t \mid \mathsf{right}\ t \mid \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{left}\ x_1 \to t_1;\ \mathsf{right}\ x_2 \to t_2\} \end{aligned}$$

Formally, this syntax requires that the argument of a λ must be annotated with a type ($\lambda x : \varphi.\, t$). However, we will sometimes omit the type annotation ($\lambda x.\, t$), either because the type can be easily understood from the context, or because it is up to you to deduce what its type annotation should be.

Typing

The typing rules for this version of the simply-typed λ-calculus are shown below.

$$\frac{x : \varphi \in \Gamma}{\Gamma \vdash x : \varphi} \text{ Var (Ax)}$$

$$\frac{\Gamma, x : \varphi \vdash t : \psi}{\Gamma \vdash \lambda x{:}\varphi.\, t : \varphi \to \psi} \text{ Lam } (\to\text{I})$$

$$\frac{\Gamma \vdash t_1 : \varphi \to \psi \qquad \Gamma \vdash t_2 : \varphi}{\Gamma \vdash t_1\ t_2 : \psi} \text{ App } (\to\text{E})$$

$$\frac{\Gamma \vdash t_1 : \varphi \qquad \Gamma \vdash t_2 : \psi}{\Gamma \vdash (t_1, t_2) : \varphi \times \psi} \text{ Pair } (\times\text{I})$$

$$\frac{\Gamma \vdash t : \varphi \times \psi}{\Gamma \vdash \mathsf{fst}\ t : \varphi} \text{ Fst } (\times\text{E}_1) \qquad \frac{\Gamma \vdash t : \varphi \times \psi}{\Gamma \vdash \mathsf{snd}\ t : \psi} \text{ Snd } (\times\text{E}_2)$$

$$\frac{\Gamma \vdash t : \varphi}{\Gamma \vdash \mathsf{left}\ t : \varphi + \psi} \text{ Left } (+\text{I}_1) \qquad \frac{\Gamma \vdash t : \psi}{\Gamma \vdash \mathsf{right}\ t : \varphi + \psi} \text{ Right } (+\text{I}_2)$$

$$\frac{\Gamma \vdash t : \varphi + \psi \qquad \Gamma, x_1 : \varphi \vdash t_1 : \chi \qquad \Gamma, x_2 : \psi \vdash t_2 : \chi}{\Gamma \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{left}\ x_1 \to t_1;\ \mathsf{right}\ x_2 \to t_2\} : \chi} \text{ Case } (+\text{E})$$

Reduction

Finally, the reduction rules are as follows:

$$\frac{}{(\lambda x{:}\varphi.\, t_1)\ t_2 \rightsquigarrow [x \mapsto t_2]t_1}\ \beta$$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{Cong-AppL} \qquad \frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{Cong-AppR}$$

$$\frac{}{\mathsf{fst}\ (t_1, t_2) \rightsquigarrow t_1}\ \beta\text{-Fst} \qquad \frac{}{\mathsf{snd}\ (t_1, t_2) \rightsquigarrow t_2}\ \beta\text{-Snd}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{fst}\ t \rightsquigarrow \mathsf{fst}\ t'}\ \text{Cong-Fst} \qquad \frac{t \rightsquigarrow t'}{\mathsf{snd}\ t \rightsquigarrow \mathsf{snd}\ t'}\ \text{Cong-Snd}$$

$$\frac{}{\mathsf{case}\ (\mathsf{left}\ t)\ \mathsf{of}\ \{\mathsf{left}\ x_1 \to t_1;\ \mathsf{right}\ x_2 \to t_2\} \rightsquigarrow [x_1 \mapsto t]t_1}\ \beta\text{-CaseL}$$

$$\frac{}{\mathsf{case}\ (\mathsf{right}\ t)\ \mathsf{of}\ \{\mathsf{left}\ x_1 \to t_1;\ \mathsf{right}\ x_2 \to t_2\} \rightsquigarrow [x_2 \mapsto t]t_2}\ \beta\text{-CaseR}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{left}\ x_1 \to t_1;\ \mathsf{right}\ x_2 \to t_2\} \rightsquigarrow \mathsf{case}\ t'\ \mathsf{of}\ \{\mathsf{left}\ x_1 \to t_1;\ \mathsf{right}\ x_2 \to t_2\}}\ \text{Cong-Case}$$

Exercise 1 Give a formal typing derivation (*i.e.* a proof tree) which assigns a type to the following term (note there are multiple correct answers):

$$\lambda p.\, \mathsf{case}\ (\mathsf{fst}\ p)\ \mathsf{of}\ \{\mathsf{left}\ x_1 \to \mathsf{right}\ x_1;\ \mathsf{right}\ x_2 \to x_2\ (\lambda z.\, z)\}$$

Exercise 2 If $\Gamma \vdash t : \varphi$ and $t \rightsquigarrow t'$, will it always be the case that $\Gamma \vdash t' : \varphi$ (for the *same* type φ)? Explain how you would go about structuring a proof of this statement. Which parts are easy? Which parts would be more difficult?

Exercise 3 Recall that in the untyped λ-calculus, we can define the Y combinator by

$$Y = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)).$$

Explain why it is not possible to give a type to Y in the simply-typed λ-calculus.

Hint: note that, as with anything defined inductively/recursively, infinite types are not allowed.

Exercise 4 Of course, just showing that Y does not have a type is not a proof that we have ruled out infinite recursion! Maybe there is some other tricky thing we can do that has a type but acts like Y. We

would like to *prove* that any well-typed term will only reduce for a finite number of steps. If a term t does not reduce infinitely we say t *terminates*. Otherwise we say t *diverges*.

One obvious approach would be to prove the following statement by (strong[1]) induction on the size[2] of λ-calculus terms:

$$\forall n \in \mathbb{N}. \forall t. (\mathsf{size}(t) = n) \implies t \text{ terminates.}$$

Unfortunately, this does not work. Explain why not.

This turns out to actually be true, but proving it is nontrivial—much too nontrivial to include on this homework, but not so nontrivial that you would not be able to understand the proof. If you are interested, you can consult Chapter 12 of Pierce[3] (you are welcome to borrow my copy).

Now that we have cut out unrestricted recursion, suppose we want to add it back in, but in a principled way.[4] The idea is to explicitly add a new term constructor Y to the syntax of λ-calculus terms, *i.e.* in addition to λ, application, *etc.*, a term t can now also be of the form $\mathsf{Y}\ t'$ for some other term t'. Note that in contrast to the Y from the untyped λ-calculus, which we just defined to be a certain λ-calculus term, this new Y is just a piece of syntax with no *a priori* meaning.

Exercise 5 Write down a typing rule for $\mathsf{Y}\ t$, that is, a rule of the form

$$\frac{\ldots}{\Gamma \vdash \mathsf{Y}\ (t) : \ldots}$$

where you fill in the dots appropriately. (*Hint*: think about the type of `fix` in Haskell.)

Exercise 6 Write down an appropriate reduction rule for Y .

Consider the following Haskell definitions:

```
data Ty where
  BaseTy :: String -> Ty
  Arr    :: Ty -> Ty -> Ty
  deriving (Eq, Show)

data Tm where
  Var :: String -> Tm
  Lam :: String -> Ty -> Tm -> Tm
  App :: Tm -> Tm -> Tm
  deriving Show
```

[1] The strong induction principle for natural numbers says that in order to prove that P holds for all natural numbers, we must prove that P holds for an arbitrary number m under the assumption that P holds for *all* numbers $k < m$. Note this also means we must prove $P(0)$ without any assumptions, since there are no $k < 0$. This is equivalent in power to the usual induction principle for natural numbers, but often "feels" more powerful, since the inductive hypothesis lets you assume that P holds for *all* numbers smaller than m instead of just the predecessor.

[2] The size of a λ-calculus term is defined as the number of constructors it contains, *i.e.* each λ contributes 1 to the size, as does each application, pair, and so on. For example, $\mathsf{size}(\lambda x.\, (\mathsf{fst}\ (y, z))) = 5$.

[3] Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002

[4] Why would we go to all the trouble of getting rid of recursion if we are just going to add it back in again? Although adding Y gives us unrestricted recursion again, it's now easy to tell just by looking at a term whether it could possibly be infinitely recursive: just see whether it contains Y or not. If it doesn't, then it will definitely terminate. Also, as we discussed in class, there are many benefits to having a type system besides just getting rid of infinite recursion.

`Ty` represents types of the STLC, and `Tm` represents terms. I have provided you with the file `STLC.hs`, which contains the above definitions along with utilities to parse and pretty-print terms and types (that is, convert between `String` representations and the above algebraic data types).

Exercise 7 Implement a function

```
inferType :: Tm -> Maybe Ty
```

which infers the type of a term. You can use the provided parsers and pretty-printers to test your function. For example,

Hint: you may find the `Data.Map` module useful.

```
ghci> let tm = readTmU "\\y:D -> A -> B. \\z:D.  \\x:(A -> B) -> ((C -> D) -> E). x (y z)"
ghci> fmap ppTy (inferType tm)
Just "(D -> A -> B) -> D -> ((A -> B) -> (C -> D) -> E) -> (C -> D) -> E"
ghci> fmap ppTy (inferType (readTmU "\\x:A. x x"))
Nothing
```

The Curry-Howard isomorphism

Exercise 8 Some of the following propositions are provable in propositional logic, and some are not.

- For those that are provable, demonstrate it *by giving a term of the STLC with a corresponding type* (you need not give a formal typing derivation, though you may find it helpful to do so). Equivalently, you may write a Haskell function with a corresponding type, as long as you are careful not to use recursion or any other Haskell features which are not part of STLC.
- For those that are not provable, explain why not.

(a) $\varphi \implies \varphi$

(b) $(\varphi \implies \varphi) \implies \varphi$

(c) $((\varphi \implies \varphi) \implies \varphi) \implies \varphi$

(d) $(\varphi \wedge (\psi \vee \chi)) \implies ((\varphi \wedge \psi) \vee (\varphi \wedge \chi))$ Déjà vu?

(e) $(\varphi \implies (\psi \wedge \chi)) \implies ((\varphi \implies \psi) \wedge (\varphi \implies \chi))$

References

Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002.