CSCI 365 Module 5

Due Wednesday 9 March, 2022

Specification

- To receive **Level 1** credit for this module:
  - Complete exercises 1 and 4–6.
  - Complete the survey at `https://forms.gle/wj4HWbq9cYrhRGci9` (you should be able to click the above link; it is also linked from the course website). Note the survey is anonymous; once you have submitted the survey, on your PDF submission please include the statement "I solemnly swear I have completed the survey."
  - Your code must adhere to the style guide linked from the course web page.
  - You must complete Quiz 5 for credit, which will test your understanding of the simply typed lambda calculus and the Curry-Howard isomorphism.
- To receive **Level 2** credit for this module:
  - Complete everything required for Level 1.
  - Complete exercises 2 and 3.

The Simply-Typed Lambda Calculus

This section provides a reference for the simply-typed λ-calculus extended with product and sum types (abbreviated STLC$^{\times+}$), as discussed in class.

Syntax

Types are denoted by the metavariables α or β and are defined by the following recursive grammar:

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

$$\alpha, \beta ::= A, B, \cdots \mid \alpha \to \beta \mid \alpha \times \beta \mid \alpha + \beta$$

That is, a type is either a base type (denoted by a capital letter like A, B, ...), a function type $\alpha \to \beta$, a product type $\alpha \times \beta$ (written (α, β) in Haskell), or a sum type $\alpha + \beta$ (written `Either` α β in Haskell).

The syntax of terms is given by

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

Formally, this syntax requires that the argument of a λ must be annotated with a type ($\lambda x : \alpha.\, t$). 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.

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

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

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

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

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

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

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

Reduction

For completeness, the *reduction rules* are as follows. We did not discuss these in class, but they show how terms of STLC$^{\times+}$ actually behave at runtime, by showing individual evaluation steps. "Running" a term consists of applying evaluation steps until it is not possible

to reduce any further. Note that all the "Cong" rules are uninteresting, and just say that we can reduce inside of terms. The β rules are where the interesting computation happens.

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

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

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

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

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

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

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

Exercise 1 Give a valid type for the following term (note there are multiple correct answers). If it is helpful, you can try constructing a formal typing derivation (*i.e.* a proof tree) which assigns it a type; but you may also reason more informally about it.

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

Exercise 2 (Level 2) If $\Gamma \vdash t : \alpha$ and $t \rightsquigarrow t'$ (that is, if t has a valid type α and can take a reduction step, resulting in the new reduced term t'), do you think it will always be the case that $\Gamma \vdash t' : \alpha$ (for the *same* type α)? In other words, does reduction preserve types? Explain how you might go about structuring a proof of this statement. Which parts are easy? Which parts would be more difficult?

Exercise 3 (Level 2) 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 In class we argued that it is not possible to give a valid type to the Y combinator. However, 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 valid 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:

$$\forall n \in \mathbb{N}. \forall t. (\mathsf{size}(t) = n \wedge t \text{ has a valid type}) \implies t \text{ terminates.}$$

Unfortunately, this does not work. Explain why not. (*Hint*: does reducing a well-typed λ-term always decrease its size?)

[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, $\mathsf{size}(\lambda x. (\mathsf{fst}\ (y, z))) = 5$.

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).

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

Consider the following Haskell definitions:

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

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

`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 5 Implement a function

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

which infers the type of a term.

Hint: You will need to make a recursive helper function which takes the current context Γ as an extra argument. You will find the `Data.Map` module useful for representing the context. Look at the Var, Lam, and App rules and try to translate them into code.

For example, given a term such as $\lambda x : A.\, x$, which would be represented by

```
Lam "x" (BaseTy "A") (Var "x"),
```

your function should figure out that it has type $A \to A$, represented as

```
Fun (BaseTy "A") (BaseTy "A").
```

The Curry-Howard isomorphism

Exercise 6 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 STLC$^{\times +}$ 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 example, given the proposition $(\alpha \wedge \gamma) \implies \beta \implies \alpha$, we can prove it by giving a λ-calculus term, or Haskell function, of type $a \times c \to b \to a$, like so:

  ```
  exampleProof :: (a,c) -> b -> a
  exampleProof = \p -> \b -> fst p
  ```

  (Of course this could be written more idiomatically in Haskell as `exampleProof (a,_) _ = a`, but I have written it above to explicitly use only syntax that is present in STLC$^{\times +}$.)

- For those that are not provable, explain why not.

(a) $\alpha \implies \alpha$

(b) $(\alpha \implies \alpha) \implies \alpha$

(c) $((\alpha \implies \alpha) \implies \alpha) \implies \alpha$

(d) $(\alpha \wedge (\beta \vee \gamma)) \implies ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$ Déjà vu?

(e) $(\alpha \implies (\beta \wedge \gamma)) \implies ((\alpha \implies \beta) \wedge (\alpha \implies \gamma))$

References

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