Before starting, be sure that you understand the course policy on Academic Integrity. Download code07.tar from the course website, which contains Standard ML files for System F interpreter.

1. **Church Encodings in System F**

Recall the encodings of booleans, pairs, and natural numbers in the (untyped) Lambda Calculus:

- \( \text{true} = \lambda x. \lambda y. x \)
- \( \text{false} = \lambda x. \lambda y. y \)
- \( \text{if} = \lambda b. \lambda t. \lambda f. b \, t \, f \)
- \( \text{and} = \lambda b_1. \lambda b_2. \, b_1 \, b_2 \, b_1 \)
- \( \text{or} = \lambda b_1. \lambda b_2. \, b_1 \, b_2 \, b_2 \)
- \( \text{not} = \lambda b. \lambda x. \lambda y. b \, y \, x \)
- \( \text{mkpair} = \lambda x. \lambda y. \lambda z. z \, x \, y \)
- \( \text{fst} = \lambda p. p \, (\lambda x. \lambda y. x) \)
- \( \text{snd} = \lambda p. p \, (\lambda x. \lambda y. y) \)
- \( \text{0} = \lambda s. \lambda z. z \)
- \( \text{1} = \lambda s. \lambda z. s \, z \)
- \( \text{2} = \lambda s. \lambda z. s \, (s \, z) \)
- \( \text{3} = \lambda s. \lambda z. s \, (s \, (s \, z)) \)
- \( \text{isZero} = \lambda n. n \, (\lambda x. \text{false}) \, \text{true} \)
- \( \text{isEven} = \lambda n. n \, \text{not} \, \text{true} \)
- \( \text{succ} = \lambda n. \lambda s. \lambda z. s \, (n \, s \, z) \)
- \( \text{plus} = \lambda m. \lambda n. \text{succ} \, n \)
- \( \text{times} = \lambda m. \lambda n. \text{plus} \, n \, 0 \)

Although these values are well-typed in the Simply-Typed Lambda Calculus, they are not particularly useful, because they must be given a single type. For example, we could give \( \text{if} \) the type \( \text{(int \rightarrow int \rightarrow int)} \rightarrow \text{int \rightarrow int \rightarrow int} \), but this \( \text{if} \) only “works” for conditionals that return an \text{int}.

With System F, we can recover useful, well-typed Church encodings of booleans, pairs, and natural numbers.
(a) We can define the boolean type\(^1\) and operations as follows:

```
bool = \forall \alpha. \alpha \to \alpha \to \alpha
true : bool
true = \Lambda \alpha. \lambda x: \alpha. \lambda y: \alpha. x
false : bool
false = \Lambda \alpha. \lambda x: \alpha. \lambda y: \alpha. y
if : \forall \beta. bool \to \beta \to \beta \to \beta
if = \Lambda \beta. \lambda b: bool. \lambda t: \beta. \lambda f: \beta. b [\beta] t f
```

Show how to write the terms **and**, **or**, and **not** in System F.

```
and : bool \to bool \to bool
or : bool \to bool \to bool
not : bool \to bool
```

(b) We can define the pair type as follows:

```
\tau_1 \ast \tau_2 = \forall \gamma. (\tau_1 \to \tau_2 \to \gamma) \to \gamma
```

Show how to write the terms **mkpair**, **fst**, and **snd** in System F.

```
mkpair : \forall \alpha. \forall \beta. \alpha \to \beta \to \alpha \ast \beta
fst : \forall \alpha. \forall \beta. \alpha \ast \beta \to \alpha
snd : \forall \alpha. \forall \beta. \alpha \ast \beta \to \beta
```

(c) We can define the natural number (Church numeral) type and natural numbers as follows:

```
nat = \forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha
0 : nat
0 = \Lambda \alpha. \lambda s: \alpha \to \alpha. \lambda z: \alpha. z
1 : nat
1 = \Lambda \alpha. \lambda s: \alpha \to \alpha. \lambda z: \alpha. s z
2 : nat
2 = \Lambda \alpha. \lambda s: \alpha \to \alpha. \lambda z: \alpha. s (s z)
3 : nat
3 = \Lambda \alpha. \lambda s: \alpha \to \alpha. \lambda z: \alpha. s (s (s z))
```

Show how to write the terms **isZero**, **isEven**, **succ**, **plus**, and **times** in System F.

```
isZero : nat \to bool
isEven : nat \to bool
succ : nat \to nat
plus : nat \to nat \to nat
times : nat \to nat \to nat
```

---

\(^1\)Think of the definition of **bool** as a “macro”: writing **bool** means writing \(\forall \alpha. \alpha \to \alpha \to \alpha\).
2. **Implementing Type Substitution in System F**

The code provided defines an abstract syntax and a scanner/parser for System F with integers, addition, multiplication, greater-than, integer-based conditionals (0 is false, other integers are true), a mutable heap, units, pairs, sums, and roll/unrolls. (It is based closely upon the code from Homework 05.)

Implement the `typ_substitute` function (in `ast.sml`), which is used to type check type application, rolls, and unrolls. Note that you must implement *capture-avoiding* substitution. Review the notes from Lecture 08 (Lambda Calculus (cont’d)) on defining substitution; your solution should closely follow the (correct) Attempt 4. Also examine the provided `typ_equal` function, which uses type substitution to compare types (with bound type variables) for equality.

The reference solution is only 17 lines long; ask for help if you find yourself attempting to write significantly more than this.
3. Parametricity (and lack thereof)

(a) Give 4 values \(v\) in System F (the language from the previous problem without reference and recursive types) such that:

- \(\vdash v : \forall \alpha. (\alpha \times \alpha) \to (\alpha \times \alpha)\)

- Each \(v\) is not equivalent to the other three (i.e., given the same arguments, it may return different results).

(b) Consider System F with references (the language from the previous problem without recursive types).

Unsurprisingly, if \(v\) is a closed value of type \(\forall \alpha. (\alpha \to \text{bool} \to \text{bool})\), then \(v [\tau] x z\) and \(v [\tau] y z\) always produce the same result in an environment where \(x\) and \(y\) are bound to values of type \(\tau\) and \(z\) is bound to a value of type \text{bool}.

Surprisingly, there exists a closed value \(v\) of type \(\forall \alpha. (\text{ref }\alpha \to \text{ref }\text{bool} \to \text{bool})\) such that in some environment \(v [\tau] x z\) evaluates to \text{true} but \(v [\tau] y z\) evaluates to \text{false}. Write down one such \(v\) and explain how to call \(v\) (i.e., what \(x\), \(y\), and \(z\) should be bound to) to get this surprising behavior. (Note: although this value behaves differently when applied to different arguments, it should always behave the same when applied to the same arguments.)

Hint: You can solve this problem in SML (i.e., you do not need any System F features not found in SML). In fact, here is a template that you might follow:

```sml
(* Define 'v'. 'v' is a closed value, in the sense that it does not mention 'x', 'y', or 'z'. *
 * (Technically, it is not completely closed because it will mention the operations '!', ':=', *
 * and possibly boolean constants and operations, but the important part is that it is not *
 * defined in terms of the 'x', 'y', or 'z' values.) *
 val v : 'a ref -> bool ref -> bool = ...;

(* Define 'x', 'y', and 'z' values. *)
val z = ...
val y = ...
val x = ...

(* Evaluate 'v' with different arguments. The repeated evaluations are to demonstrate *
 * that the answer depends upon the values of *
 * the arguments, not only on side effects. *
 * In particular, 'v x z' always returns 'true' and 'v y z' always returns 'false'. *)
val ans1 = v x z
val ans2 = v y z
val ans3 = v y z
val ans4 = v x z
val ans5 = v x z
val ans6 = v y z

(* Check answers. *)
val [true, false, false, true, true, false] = [ans1, ans2, ans3, ans4, ans5, ans6];
```
4. Debriefing
   • How many hours did you spend on this assignment?
   • Would you rate it as easy, moderate, or difficult?
   • How deeply do you feel you understand the material it covers (0% – 100%)?
   • If you have any other comments about the assignment, then please include them with your submission or send email to mtf@cs.rit.edu.

Submission

Programming components of the assignment:
   • problem 2
must be submitted to the Homework 7 Assignment on MyCourses by the due date.

Written components of the assignment:
   • problems 1a, 1b, 1c
   • problems 3a, 3b
   • debriefing
must either be submitted to the instructor in class or submitted as a PDF file to the Homework 7 Assignment on MyCourses by the due date.

Document History

November 18, 2019
   Original version