

# Type Theory (15-814) Fall 2006

## Homework 4: Polymorphism and Control Effects

William Lovas (wlovas@cs)

Out: Friday, October 17, 2006

Due: Tuesday, October 28, 2006 (before 1:30 pm)

In this assignment, you'll review polymorphic Church encodings, practice reasoning about control effects, and implement an abstract machine.

### 1 Polymorphism

Recall from class System F, or the polymorphic  $\lambda$ -calculus.

$$\begin{aligned}\tau &::= X \mid \tau_1 \rightarrow \tau_2 \mid \forall X. \tau \\ e &::= x \mid \lambda x:\tau. e \mid e_1 e_2 \mid \Lambda X. e \mid e[\tau]\end{aligned}$$

Despite its simplicity, System F is quite expressive. As we discussed in class, it has sufficient expressive power to be able to encode many datatypes found in other programming languages, including products, sums, and inductive datatypes.

For example, remember that nats may be encoded as follows:

$$\begin{aligned}\text{nat} &:= \forall X. X \rightarrow (X \rightarrow X) \rightarrow X \\ \text{zero} &:= \Lambda X. \lambda z:X. \lambda s:X \rightarrow X. z \\ \text{succ} &:= \lambda n:\text{nat}. \Lambda X. \lambda z:X. \lambda s:X \rightarrow X. s (n [X] z s)\end{aligned}$$

This is just a typed version of the Church encoding of natural numbers. A  $\text{nat } \bar{n}$  is defined by what it can do, which is to compute a function iterated  $n$  times. In the polymorphic encoding above, the result of that iteration can be any type  $X$ , as long as you have a base element  $z : X$  and a function  $s : X \rightarrow X$ .

Conveniently, this encoding "is" its own elimination form, in a sense:

$$\mathbf{rec}(e, e_0, x:\tau. e_1) := e[\tau] e_0 (\lambda x:\tau. e_1)$$

The case analysis is baked into the very definition of the type.

**Exercise 1.** Verify that these encodings typecheck in System F.

Similarly, we can encode  $\tau$  lists, lists of elements of type  $\tau$ <sup>1</sup>.

$$\begin{aligned}\tau \text{ list} &:= \forall X. X \rightarrow (\tau \rightarrow X \rightarrow X) \rightarrow X \\ \text{nil}_\tau &:= \Lambda X. \lambda n:X. \lambda c:\tau \rightarrow X \rightarrow X. n \\ \text{cons}_\tau &:= \lambda h:\tau. \lambda t:\tau \text{ list}. \Lambda X. \lambda n:X. \lambda c:\tau \rightarrow X \rightarrow X. c h (t [X] n c)\end{aligned}$$

---

<sup>1</sup>Note that we simply specify an encoding that works at any type  $\tau$ —to formally be generic in that type, we would need to use *type operators*, the subject of study in an extension of System F called  $F_\omega$ .

As with nats, The  $\tau$  list type's case analyzing elimination form is just application. We can write functions like map:

$$\begin{aligned} \text{map} &: (\sigma \rightarrow \tau) \rightarrow \sigma \text{ list} \rightarrow \tau \text{ list} \\ &:= \lambda f:\sigma \rightarrow \tau. \lambda l:\sigma \text{ list}. l[\tau \text{ list}] \text{ nil}_\tau (\lambda x:\sigma. \lambda y:\tau \text{ list}. \text{cons}_\tau (f x) y) \end{aligned}$$

As mentioned above and in class, System F can express any inductive datatype defined by a *positive operator*.

**Exercise 2.** Consider the following simple binary tree type:

```
datatype 'a tree =
  Leaf
  | Node of 'a tree * 'a * 'a tree
```

- Give a System F encoding of binary trees, including a definition of the type  $\tau$  tree and definitions of the constructors  $\text{leaf} : \tau \text{ tree}$  and  $\text{node} : \tau \text{ tree} \rightarrow \tau \rightarrow \tau \text{ tree} \rightarrow \tau \text{ tree}$ .
- Write a function  $\text{height} : \tau \text{ tree} \rightarrow \text{nat}$ . You may assume the above encoding of nat as well as definitions of the functions  $\text{plus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$  and  $\text{max} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ .
- Write a function  $\text{in-order} : \tau \text{ tree} \rightarrow \tau \text{ list}$  that computes the in-order traversal of a binary tree. You may assume the above encoding of lists; define any auxiliary functions you need.

## 2 Control effects

Consider the following language, a simply-typed  $\lambda$ -calculus with natural numbers and control effects.

$$\begin{aligned} \tau &::= \text{nat} \mid \tau_1 \rightarrow \tau_2 \mid \tau \text{ cont} \\ e &::= x \mid \text{zero} \mid \text{succ } e \mid \mathbf{natcase } e \mathbf{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} \mid \mathbf{fix } x:\tau. e \mid \lambda x:\tau. e \mid e_1 e_2 \\ &\quad \mid \text{fail}_\tau \mid \mathbf{try } e_1 \mathbf{ ow } e_2 \mid \mathbf{letcc } x : \tau \text{ cont } \mathbf{in } e \mid \mathbf{throw}_\tau e_1 \mathbf{ to } e_2 \mid \text{cont}(K) \\ v &::= \text{zero} \mid \text{succ } v \mid \lambda x:\tau. e \mid \text{cont}(K) \\ \\ K &::= \epsilon \mid f;K \\ f &::= \text{succ } \square \mid \mathbf{natcase } \square \mathbf{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} \mid \square e_2 \mid v_1 \square \\ &\quad \mid \mathbf{try } \square \mathbf{ ow } e_2 \mid \mathbf{throw}_\tau \square \mathbf{ to } e_2 \mid \mathbf{throw}_\tau v_1 \mathbf{ to } \square \\ \\ S &::= K > e \mid K < v \mid K < \text{fail} \end{aligned}$$

In order to capture the control behavior of simple exceptions and **letcc**, we specify the dynamic semantics in terms of an abstract machine called the C machine. The C machine's states make the stack explicit, and thus enable us to reify the stack as a first-class expression.

The C machine has three states:

- $K > e$ , for evaluating an expression  $e$  on a stack  $K$ ;
- $K < v$ , for returning a value  $v$  to a stack  $K$ ; and
- $K < \text{fail}$ , to propagate simple failure up the stack.

Everytime the machine is in an evaluation state  $K > e$ , it picks out the principal subexpression of  $e$ , considers what work will remain after evaluating the principal expression, adds that work to the control stack  $K$  in the form of a new stack frame  $f$ , and evaluates the subexpression on the augmented stack  $f; K$ . Then, later, when the machine is in a return state  $f; K < v$ , it picks the last bit of work  $f$  off the stack  $K$  and performs that work on the value  $v$ , possibly evaluating the result in the remaining stack  $K$ .

An expression is initially evaluated in the empty stack; that is, the machine always starts out in a state  $\epsilon > e$ .

The complete static and dynamic semantics of this language are shown in Figures 1, 2, 3, and 4.

**Exercise 3.** Suppose we had an implementation of the plus function. What would the expression

$$\text{plus } 1 \text{ (letcc } k : \text{nat cont in plus } 2 \text{ (throw}_{\text{nat}} 3 \text{ to } k))$$

evaluate to? Explain your answer in terms of a sequence of state transitions.

**Exercise 4.** How does type safety change for this language? State Progress and Preservation, and sketch proofs of both, isolating a few interesting cases. How are these proofs different from the Progress and Preservation proofs we've done before?

Consider extending the language with call-by-value products:

$$\begin{aligned} \tau &::= \dots \mid \tau_1 \times \tau_2 \\ e &::= \dots \mid \langle e_1, e_2 \rangle \mid \text{fst } e \mid \text{snd } e \\ v &::= \dots \mid \langle v_1, v_2 \rangle \end{aligned}$$

Note that a pair is only a value if both of its components are values.

**Exercise 5.** Add new frames, frame typing rules, and state transition rules to specify the semantics of pairs. Keep type safety in mind, and ensure that Progress and Preservation will still hold under your extension.

**Exercise 6 (optional).** Extend the language with polymorphism and/or existential types.

### 3 Implementation

For the implementation section, you'll implement the C machine in SML. For the purposes of this implementation, it's convenient to statically distinguish expressions from values. Thus the datatype representing machine states is as follows:

```
datatype state =
  StEvaluate of Syntax.stack * Syntax.exp
  | StReturn of Syntax.stack * Syntax.value
  | StFail of Syntax.stack * unit
```

The type `Syntax.stack` is just an alias for `Syntax.frame list`; the constructors for `Syntax.frame` mirror those for expressions, but with the `unit` type everywhere a hole ( $\square$ ) occurs. These `unit` arguments obviously have no effect on your program's behavior, but they may help you to keep all of the different kinds of frames straight.

Since values are supposed to be a subset of expressions, there's an explicit injection constructor `ExValue` : `value -> exp` taking a value and creating an expression. This allows us to substitute values into expressions; see the function `Evaluate.substValue`.

**Program 1.** Modify and extend your typechecker from homework 2 (our language here is no longer bidirectional) to handle this new language. You'll have to remove the `rec(...)` cases and add cases for `natcase`, `fix`, `fail`, `try`, `letcc`, and `throw` expressions. (You need not implement continuation typing  $\Gamma \vdash K \sim \tau$ , since evaluation stacks only arise at run-time. You need not typecheck any `ExValue` expressions for similar reasons—you may simply raise an exception to cover the case.)

---

$$\frac{}{\Gamma, x:\tau \vdash x : \tau} \text{(V)} \qquad \frac{\Gamma, x:\tau \vdash e : \tau}{\Gamma \vdash \mathbf{fix} x:\tau. e : \tau} \text{(F)}$$
$$\frac{}{\Gamma \vdash \mathbf{zero} : \mathbf{nat}} \text{(N -I-Z)} \qquad \frac{\Gamma \vdash e : \mathbf{nat}}{\Gamma \vdash \mathbf{succ} e : \mathbf{nat}} \text{(N -I-S)}$$
$$\frac{\Gamma \vdash e : \mathbf{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x:\mathbf{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathbf{natcase} e \mathbf{of} \{ \mathbf{zero} \Rightarrow e_0, \mathbf{succ} x \Rightarrow e_1 \} : \tau} \text{(N -E)}$$
$$\frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} \text{(\(\rightarrow\)-I)} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \text{(\(\rightarrow\)-E)}$$
$$\frac{}{\Gamma \vdash \mathbf{fail}_\tau : \tau} \text{(F)} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{try} e_1 \mathbf{ow} e_2 : \tau} \text{(T)}$$
$$\frac{\Gamma, x:\tau \mathbf{cont} \vdash e : \tau}{\Gamma \vdash \mathbf{letcc} x : \tau \mathbf{cont} \mathbf{in} e : \tau} \text{(L)} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \mathbf{cont}}{\Gamma \vdash \mathbf{throw}_{\tau'} e_1 \mathbf{to} e_2 : \tau'} \text{(T)}$$
$$\frac{\Gamma \vdash K \sim \tau}{\Gamma \vdash \mathbf{cont}(K) : \tau \mathbf{cont}} \text{(C)}$$

---

Figure 1: Static semantics of expressions

**Program 2.** Add frame constructors for pairs and projections in `syntax-sig.sml` and `syntax.sml`. You'll have to modify several minor functions that operate on frames; all the places that require modifications are marked with `TODO` in `syntax.sml` and `syntax-sig.sml`.

**Program 3.** Implement state evaluation `Evaluate.evaluate : Evaluate.state -> Evaluate.state`. You may implement this either in a big-step style or by applying repeated single-step state transitions, but it should return a final state, i.e. either an `StReturn` or an `StFail` with an empty stack.

You can test your programs using the `Top` structure as usual; the concrete syntax is similar to that of previous assignments. (There is no concrete syntax for stacks since they only arise during evaluation.) Some simple examples are included in `examples.txt`.

---

$\Gamma \vdash K \sim \tau$

$$\frac{}{\Gamma \vdash \epsilon \sim \tau} \text{ (S -E )} \quad \frac{\Gamma \vdash f \sim \tau \rightsquigarrow \tau' \quad \Gamma \vdash K \sim \tau'}{\Gamma \vdash f; K \sim \tau} \text{ (S -C )}$$

$\Gamma \vdash f \sim \tau_1 \rightsquigarrow \tau_2$

$$\frac{}{\Gamma \vdash \text{succ } \square \sim \text{nat} \rightsquigarrow \text{nat}} \text{ (F-S )} \quad \frac{\Gamma \vdash e_0 : \tau \quad \Gamma, x:\text{nat} \vdash e_1 : \tau}{\Gamma \vdash \text{natcase } \square \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} \sim \text{nat} \rightsquigarrow \tau} \text{ (F-N )}$$

$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \square e_2 \sim \tau_2 \rightarrow \tau \rightsquigarrow \tau} \text{ (F-A 1)} \quad \frac{\Gamma \vdash v_1 : \tau_2 \rightarrow \tau}{\Gamma \vdash v_1 \square \sim \tau_2 \rightsquigarrow \tau} \text{ (F-A 2)}$$

$$\frac{\Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{try } \square \text{ ow } e_2 \sim \tau \rightsquigarrow \tau} \text{ (F-T )}$$

$$\frac{\Gamma \vdash e_2 : \tau \text{ cont}}{\Gamma \vdash \text{throw}_{\tau'} \square \text{ to } e_2 \sim \tau \rightsquigarrow \tau'} \text{ (F-T 1)} \quad \frac{\Gamma \vdash v_1 : \tau}{\Gamma \vdash \text{throw}_{\tau'} v_1 \text{ to } \square \sim \tau \text{ cont} \rightsquigarrow \tau'} \text{ (F-T 2)}$$

$\Gamma \vdash S \text{ ok}$

$$\frac{\Gamma \vdash K \sim \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash K > e \text{ ok}} \text{ (S -E )} \quad \frac{\Gamma \vdash K \sim \tau \quad \Gamma \vdash v : \tau \quad v \text{ value}}{\Gamma \vdash K < v \text{ ok}} \text{ (S -R )} \quad \frac{\Gamma \vdash K \sim \tau}{\Gamma \vdash K < \text{fail ok}} \text{ (S -F )}$$


---

Figure 2: Static semantics of stacks, frames, and states

---

 $S_1 \mapsto S_2$ 

$$\begin{aligned} K > \text{zero} &\mapsto K < \text{zero} \\ K > \text{succ } e &\mapsto \text{succ } \square; K > e \\ K > \text{natcase } e \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \} &\mapsto \text{natcase } \square \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \}; K > e \\ \text{succ } \square; K < v &\mapsto K < \text{succ } v \\ \text{natcase } \square \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \}; K < \text{zero} &\mapsto K > e_0 \\ \text{natcase } \square \text{ of } \{ \text{zero} \Rightarrow e_0, \text{succ } x \Rightarrow e_1 \}; K < \text{succ } v &\mapsto K > [v/x] e_1 \\ K > \text{fix } x:\tau. e &\mapsto K > [\text{fix } x:\tau. e/x] e \\ K > \lambda x:\tau. e &\mapsto K < \lambda x:\tau. e \\ K > e_1 e_2 &\mapsto \square e_2; K > e_1 \\ \square e_2; K < v_1 &\mapsto v_1 \square; K > e_2 \\ (\lambda x:\tau. e) \square; K < v_2 &\mapsto K > [v_2/x] e \\ \\ K > \text{fail}_\tau &\mapsto K < \text{fail} \\ K > \text{try } e_1 \text{ ow } e_2 &\mapsto \text{try } \square \text{ ow } e_2; K > e_1 \\ \text{try } \square \text{ ow } e_2; K < v_1 &\mapsto K < v_1 \\ \text{try } \square \text{ ow } e_2; K < \text{fail} &\mapsto K > e_2 \\ f; K < \text{fail} &\mapsto K < \text{fail} \quad (f \neq \text{try } \square \text{ ow } e_2) \\ \\ K > \text{letcc } x : \tau \text{ cont in } e &\mapsto K > [\text{cont}(K)/x] e \\ K > \text{cont}(K') &\mapsto K < \text{cont}(K') \\ K > \text{throw}_\tau e_1 \text{ to } e_2 &\mapsto \text{throw}_\tau \square \text{ to } e_2; K > e_1 \\ \text{throw}_\tau \square \text{ to } e_2; K < v_1 &\mapsto \text{throw}_\tau v_1 \text{ to } \square; K > e_2 \\ \text{throw}_\tau v_1 \text{ to } \square; K < \text{cont}(K') &\mapsto K' < v_1 \end{aligned}$$

---

Figure 3: Dynamic semantics

---

$$\begin{array}{c} \frac{}{\text{zero value}} \quad \frac{e \text{ value}}{\text{succ } e \text{ value}} \quad \frac{}{\lambda x:\tau. e \text{ value}} \quad \frac{}{\text{cont}(K) \text{ value}} \\ \\ \frac{e \text{ value}}{\epsilon < e \text{ final}} \quad \frac{}{\epsilon < \text{fail final}} \end{array}$$

---

 $S \text{ final}$

Figure 4: Values and final states