

CMCS 321: Programming Languages

Homework #6

Guest Instructor: Derek Dreyer (dreyer@tti-c.org)

Assigned: Thursday, November 9, 2006

Due: Tuesday, November 21, 2006

1 Type Soundness for System F

How do the new polymorphic constructs of System F—namely, type abstraction $(\Lambda\alpha.e)$ and type application $(e[\tau])$ —affect the proof of type soundness? In particular, are there any new lemmas you will need? If so, state them and explain where they are needed in the proof. *You do not need to show any new cases in the proof.*

2 Infix Traversal of Church Trees

Consider the following SML code:

```
datatype  $\alpha$  tree = Empty | Node of  $\alpha$  *  $\alpha$  tree *  $\alpha$  tree

fun infix (T :  $\alpha$  tree) :  $\alpha$  list =
  case T of Empty => nil
         | Node(n,left,right) => (infix left) @ [n] @ (infix right)
```

(Here, `@` is the SML append function on lists.) Show how to Church-encode the type `α tree` and the function `infix` in System F. Your solution may make reference to the `nil` and `cons` values defined in class but should not make use of product types (primitive or Church-encoded).

3 An Alternative Version of Existential Unpack

Suppose we extend System F with primitive existentials, with the following small-step evaluation rules:

Types $\sigma, \tau ::= \dots \mid \exists\alpha. \tau$
Terms $e, f ::= \dots \mid \text{pack } [\sigma, e] \text{ as } \exists\alpha. \tau \mid \text{let } [\alpha, x] = \text{unpack } e \text{ in } e'$
Values $v, w ::= \dots \mid \text{pack } [\sigma, v] \text{ as } \exists\alpha. \tau$

$$\frac{e \rightsquigarrow e'}{\text{pack } [\sigma, e] \text{ as } \exists\alpha. \tau \rightsquigarrow \text{pack } [\sigma, e'] \text{ as } \exists\alpha. \tau}$$
$$\frac{e \rightsquigarrow e'}{\text{let } [\alpha, x] = \text{unpack } e \text{ in } f \rightsquigarrow \text{let } [\alpha, x] = \text{unpack } e' \text{ in } f}$$
$$\frac{}{\text{let } [\alpha, x] = \text{unpack } (\text{pack } [\sigma, v] \text{ as } \exists\alpha. \tau) \text{ in } e \rightsquigarrow [\sigma/\alpha][v/x]e}$$

The standard typing rules for existentials are as follows:

$$\frac{\Delta \vdash \sigma \text{ type} \quad \Delta; \Gamma \vdash e : [\sigma/\alpha]\tau}{\Delta; \Gamma \vdash \text{pack } [\sigma, e] \text{ as } \exists\alpha. \tau : \exists\alpha. \tau}$$

$$\frac{\Delta; \Gamma \vdash e : \exists\alpha. \tau \quad \Delta, \alpha; \Gamma, x : \tau \vdash e' : \tau' \quad \alpha \notin \text{FV}(\tau')}{\Delta; \Gamma \vdash \text{let } [\alpha, x] = \text{unpack } e \text{ in } e' : \tau'}$$

Notice the rule for `unpack` requires that the result type τ' not mention α . Suppose we were to change the `unpack` rule to

$$\frac{\Delta; \Gamma \vdash e : \exists\alpha. \tau \quad \Delta, \alpha; \Gamma, x : \tau \vdash e' : \tau'}{\Delta; \Gamma \vdash \text{let } [\alpha, x] = \text{unpack } e \text{ in } e' : \exists\alpha. \tau'}$$

and suppose further that we were to treat τ and $\exists\alpha. \tau$ as interchangeable type expressions in the case that $\alpha \notin \text{FV}(\tau)$. This second rule has the advantage of avoiding the side condition $\alpha \notin \text{FV}(\tau')$.

- (a). Can you mimic the new rule using the original static semantics for `unpack`? That is, if under the original semantics $\Delta; \Gamma \vdash e : \exists\alpha. \tau$ and $\Delta, \alpha; \Gamma, x : \tau \vdash e' : \tau'$, can you construct a term e'' such that $\Delta; \Gamma \vdash e'' : \exists\alpha. \tau'$, also under the original semantics?
- (b). How about vice versa? That is, if under the *new* semantics $\Delta; \Gamma \vdash e : \exists\alpha. \tau$ and $\Delta, \alpha; \Gamma, x : \tau \vdash e' : \tau'$ and $\alpha \notin \text{FV}(\tau')$, can you construct a term e'' such that $\Delta; \Gamma \vdash e'' : \tau'$, also under the new semantics?
- (c). Is the new version of the static semantics type-safe? If so, prove it by showing the new cases of the type soundness proof. If not, give a counterexample to type safety.

4 Using Girard's \mathcal{J} to Implement Recursion

For this problem, we are working in System F with full call-by-name β -reduction, extended with Girard's 0 and \mathcal{J} operators. Recall the semantics of Girard's \mathcal{J} operator:

$$\overline{\Delta; \Gamma \vdash \mathcal{J} : \forall\alpha. \forall\beta. \alpha \rightarrow \beta}$$

$$\frac{\sigma = \tau}{\mathcal{J}[\sigma][\tau](e) \rightsquigarrow e} \quad \frac{\sigma \neq \tau \quad \sigma \text{ and } \tau \text{ are closed}}{\mathcal{J}[\sigma][\tau](e) \rightsquigarrow 0[\tau]}$$

Let us say that a closed term Y “encodes the fixed-point combinator at type τ ” if: (1) Y has type $(\tau \rightarrow \tau) \rightarrow \tau$, and (2) for any closed term f of type $\tau \rightarrow \tau$, there exists a term e such that $Y(f) \rightsquigarrow^* e$ and $e \rightsquigarrow^* f(e)$.

Your task is to use Girard's \mathcal{J} operator to define a closed term `fix` such that for all closed types τ , it is the case that `fix` $[\tau]$ encodes the fixed-point combinator at type τ .

Hint: $Y = \lambda f. (\lambda x. f(x x))(\lambda x. f(x x))$ is the fixed-point combinator in the classical untyped λ -calculus. At least in my solution to this problem, the untyped erasure of my `fix` is precisely the classical Y combinator.