

## Homework 7

15-814: Type Systems for Programming Languages

TA: Kumar Avijit (kavijit@cs.cmu.edu)

**Out:** Nov. 5, 2009

**Due:** Nov. 12, 2009 (11:59 PM)

### 1 Recap of System $\mathbf{F}$

We recap the static semantics of System  $\mathbf{F}$  (refer to PFPL Chapter 23). Even though  $\text{nat}$  is definable in  $\mathbf{F}$ , we add it explicitly for ease of use:

$$\begin{array}{lcl}
 \text{Types } \tau & ::= & t \mid \tau_1 \rightarrow \tau_2 \mid \forall t. \tau \mid \text{nat} \\
 \text{Terms } m & ::= & x \mid \lambda x: \tau. m \mid \text{ap}(m_1; m_2) \mid \Lambda t. m \mid m[\tau] \\
 & & \mid z \mid \text{s}(m) \mid \text{rec}\{m; m_0; x. m_s\} \\
 \text{Type context } \Delta & ::= & \cdot \mid \Delta, t \text{ type} \\
 \text{Term context } \Gamma & ::= & \cdot \mid \Gamma, x: \tau
 \end{array}$$
  

$$\frac{}{\Delta; \Gamma, x: \tau \vdash x: \tau} \qquad \frac{\Delta; \Gamma, x: \tau \vdash m: \tau'}{\Delta; \Gamma \vdash \lambda x: \tau. m: \tau \rightarrow \tau'}$$
  

$$\frac{\Delta; \Gamma \vdash m_1: \tau_2 \rightarrow \tau_1 \quad \Delta; \Gamma \vdash m_2: \tau_2}{\Delta; \Gamma \vdash \text{ap}(m_1; m_2): \tau_1} \qquad \frac{\Delta, t \text{ type}; \Gamma \vdash m: \tau}{\Delta; \Gamma \vdash \Lambda t. m: \forall t. \tau}$$
  

$$\frac{\Delta \vdash t \text{ type} \quad \Delta; \Gamma \vdash m: \forall t. \tau'}{\Delta; \Gamma \vdash m[\tau]: [\tau/t]\tau'} \qquad \frac{}{\Delta; \Gamma \vdash z: \text{nat}} \qquad \frac{\Delta; \Gamma \vdash m: \text{nat}}{\Delta; \Gamma \vdash \text{s}(m): \text{nat}}$$
  

$$\frac{\Delta; \Gamma \vdash m: \text{nat} \quad \Delta; \Gamma \vdash m_0: \tau \quad \Delta; \Gamma, x: \tau \vdash m_s: \tau}{\Delta; \Gamma \vdash \text{rec}\{m; m_0; x. m_s\}: \tau}$$

Figure 1: Syntax and static semantics of  $\mathbf{F}$

## 1.1 Encoding lists

**Question 1.** Consider the type `list` of lists of natural numbers. The introduction and elimination rules are as follows:

$$\frac{}{\Gamma \vdash \text{empty} : \text{list}} (\text{list-I}_1) \qquad \frac{\Gamma \vdash m_1 : \text{nat} \quad \Gamma \vdash m_2 : \text{list}}{\Gamma \vdash \text{cons}(m_1; m_2) : \text{list}} (\text{list-I}_2)$$

$$\frac{\Gamma \vdash m : \text{list} \quad \Gamma \vdash m_1 : \tau \quad \Gamma, x:\text{nat}, y:\tau \vdash m_2 : \tau}{\Gamma \vdash \text{listrec}(m; m_1; x, y.m_2) : \tau} (\text{list-E})$$

Give an encoding of `list` in  $\mathbf{F}$ . Define `empty`, `cons` and `listrec` according to your encoding. Verify the  $\beta$ -rules for lists:

1.  $\text{listrec}(\text{empty}; m_1; x, y.m_2) \equiv m_1$
2.  $\text{listrec}(\text{cons}(m_h; m_t); m_1; x, y.m_2) \equiv [m_h/x][\text{listrec}(m_t; m_1; x, y.m_2)/y]m_2$

## 1.2 Encoding streams

Consider the type `stream` of streams of natural numbers. The introduction and elimination forms are as follows:

$$\frac{\Gamma \vdash m : \text{stream}}{\Gamma \vdash \text{head}(m) : \text{nat}} (\text{stream-E}_1) \qquad \frac{\Gamma \vdash m : \text{stream}}{\Gamma \vdash \text{tail}(m) : \text{stream}} (\text{stream-E}_2)$$

$$\frac{\Gamma \vdash m_s : \tau \quad \Gamma \vdash m_h : \tau \rightarrow \text{nat} \quad \Gamma \vdash m_t : \tau \rightarrow \tau}{\Gamma \vdash \text{corec } m_s \{\text{hd} \Rightarrow m_h \ \& \ \text{tl} \Rightarrow m_t\} : \text{stream}} (\text{stream-I})$$

In order to encode streams, it is helpful to first add existential types and product types to  $\mathbf{F}$ . This addition does not increase the expressiveness of  $\mathbf{F}$  since both products and existentials are already definable in it. Let us call this extension  $\mathbf{F}^{\times\exists}$  (the “...” stand for the syntax given in Figure 1).

$$\begin{array}{ll} \text{Types } \tau & ::= \dots \mid \tau_1 \times \tau_2 \mid \exists t. \tau \\ \text{Terms } m & ::= \dots \mid \langle m_1; m_2 \rangle \mid \text{fst}(m) \mid \text{snd}(m) \mid \\ & \quad \text{pack } \tau \text{ with } m \text{ as } \exists t. \tau' \mid \text{open } m_1 \text{ as } t \text{ with } x:\tau \text{ in } m_2 \end{array}$$

Given existential types, we can think of streams in terms of their introduction form: a stream is composed of three parts: a seed element, a “head” function that maps the seed to a natural number, and a “tail” function that generates another seed element from the current seed. However notice that given any type  $\tau$ , and terms  $m_s, h$  and  $t$  of types  $\tau, \tau \rightarrow \text{nat}$  and  $\tau \rightarrow \tau$  resp., one can form a stream. This suggests that `corec` can be used to define a function of the following type<sup>1</sup>

<sup>1</sup>The type  $\tau_1 \times \tau_2 \times \tau_3$  should be read as  $\tau_1 \times (\tau_2 \times \tau_3)$

$$\exists t.(t \times (t \rightarrow \text{nat}) \times (t \rightarrow t)) \rightarrow \text{stream}$$

Moreover, given a term of type `stream`, one can form a package of type  $\exists t.(t \times (t \rightarrow \text{nat}) \times (t \rightarrow t))$  by abstracting over the type `stream` itself. This suggests that `stream` can be identified with the existential type  $\exists t.(t \times (t \rightarrow \text{nat}) \times (t \rightarrow t))$ .

**Question 2.** Define `head`, `tail` and `corec` according to the above encoding of `stream`. Show that the following definitional equalities hold:

1.  $\text{head}(\text{corec } m_s \{\text{hd} \Rightarrow m_h \ \& \ \text{tl} \Rightarrow m_t\}) \equiv \text{ap}(m_h; m_s)$
2.  $\text{tail}(\text{corec } m_s \{\text{hd} \Rightarrow m_h \ \& \ \text{tl} \Rightarrow m_t\}) \equiv \text{corec } \text{ap}(m_t; m_s) \{\text{hd} \Rightarrow m_h \ \& \ \text{tl} \Rightarrow m_t\}$

## 2 Abstraction using existential types

Existential types can be used to define abstractions. The introduction form `pack  $\rho$  with  $m$  as  $\exists t.\tau$`  implements the abstraction using the representation type  $\rho$  and the implementation  $m$ . The type variable  $t$  stands for that component of  $m$ 's type that a client is not allowed to depend on, and hence must treat generically. This is evidenced in the typing of the elimination form:

$$\frac{\Delta; \Gamma \vdash m_1 : \exists t.\tau_1 \quad \Delta, t \text{ type}; \Gamma, x:\tau_1 \vdash m_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type}}{\Delta; \Gamma \vdash \text{open } m_1 \text{ as } t \text{ with } x:\tau_1 \text{ in } m_2 : \tau_2}$$

where the client  $m_2$  is typed generic in  $t$ . This genericity allows  $m_2$  to be linked safely with different implementations  $m_1$  of the same specification  $\exists t.\tau_1$ .

We would like to define a type `complex` to represent complex numbers. For the sake of this example, let us suppose we wish to have only the following operations defined on complex numbers:

- `i : complex`
- `realToComplex : real → complex`
- `magnitude : complex → real`
- `add : complex → complex → complex`

Figure 2: Signature of an abstract type for complex numbers

We add the following syntax to our language:

Terms  $m ::= \text{abstype } t \text{ with } x_1:\sigma_1, \dots, x_n:\sigma_n \text{ is rep}(\sigma; m_1, \dots, m_n) \text{ in } m$

The syntax introduces both the *implementor* as well as the *client* of an abstract type  $t$  at the same time. It defines an abstract type with available operations of types  $\sigma_1, \dots, \sigma_n$ , where  $\sigma_i$ 's can mention the type variable  $t$ . The client code  $m$  can refer to the operations using variables  $x_1$  through  $x_n$ . We will call  $x_1:\sigma_1, \dots, x_n:\sigma_n$  the *signature* of the abstract type  $t$ . The part  $\text{rep}(\sigma; m_1, \dots, m_n)$  implements all the supported operations as per the specification signature. All the implementation pieces  $m_i$ 's are written for a concrete type  $\sigma$  in place of  $t$ . The  $m_i$ 's each implement the corresponding  $x_i$ 's. The static semantics is as follows:

$$\frac{\Delta; \Gamma \vdash m_i : [\sigma/t]\sigma_i \quad (i \in [1..n]) \quad \Delta, t \text{ type}; \Gamma, x_1:\sigma_1, \dots, x_n:\sigma_n \vdash m : \tau \quad \Delta \vdash \tau \text{ type}}{\Delta; \Gamma \vdash \text{abstype } t \text{ with } x_1:\sigma_1, \dots, x_n:\sigma_n \text{ is } \text{rep}(\sigma; m_1, \dots, m_n) \text{ in } m : \tau}$$

Dynamically, the implementation code  $m_1, \dots, m_n$  is simply substituted for  $x_1, \dots, x_n$  in  $m$ .

$$\text{abstype } t \text{ with } x_1:\sigma_1, \dots, x_n:\sigma_n \text{ is } \text{rep}(\sigma; m_1, \dots, m_n) \text{ in } m \mapsto [\sigma/t]([m_1, \dots, m_n/x_1, \dots, x_n]m)$$

Using the abstype framework, the interface to complex numbers can be written as

```
abstype complex with
  i : complex,
  realToComplex : real -> complex,
  magnitude : complex -> real,
  add : complex -> complex -> complex

is
  rep ....

in
  ....
```

**Question 3.** You will now implement the abstract type `complex` in two different ways, adhering to the signature in Figure 2 each time.

1. For the first concrete representation, use the rectangular form  $a + bi$  of complex numbers, i.e., use the representation type `real × real`, with the two components of the pair being the real and the imaginary parts resp. You can assume that the following functions on the type `real` are available:

- `sqrt : real → real`
- `plus : real → real`
- `mult : real → real`

Moreover you can use real numbers, e.g.,  $1, 2, \pi, \pi/2$  etc. as terms of type `real`.

2. Now implement `complex` in the polar form  $re^{i\theta}$ . This implementation again uses a representation type `real × real`, but in this case, the two components of representation type correspond to  $r$  and  $\theta$  resp.

**Question 4.** Exhibit a relation  $R : (\text{real} \times \text{real}) \leftrightarrow (\text{real} \times \text{real})$  which is bisimulation with respect to the operations in Figure 2. Let us call the above two implementations of `complex` as `comp1` and `comp2` resp., with  $i_i$ , `realToComplexi`, etc. referring to the operations in the  $i$ 'th implementation, for  $i = 1, 2$ . Clearly state what it means for the two implementations to be bisimilar. You do not need to prove that your relation is the required bisimulation.

**Question 5.** Express the `abstype` construct in  $\mathbf{F}^{\times\exists}$ .

### 3 Equational reasoning

In this section, we will extend the definition of extensional equivalence to product and sum types for Godel's **T**. We briefly recall the syntax here:

Types	$\tau ::=$	<code>nat</code>   $\tau_1 \rightarrow \tau_2$   $\tau_1 \times \tau_2$   $\tau_1 + \tau_2$
Terms	$e ::=$	$x$   $\lambda x:\tau.e$   <code>ap</code> ( $e_1; e_2$ )
		$\langle e_1; e_2 \rangle$   <code>fst</code> ( $e$ )   <code>snd</code> ( $e$ )
		<code>inl</code> <sub><math>\tau_1, \tau_2</math></sub> $e$   <code>inr</code> <sub><math>\tau_1, \tau_2</math></sub> $e$   <code>case</code> $e$ of { <code>inl</code> $x \Rightarrow e_1$   <code>inr</code> $x \Rightarrow e_2$ }
		<code>z</code>   <code>s</code> ( $e$ )   <code>rec</code> { $e; e_0; x.e_1$ }

Recall that the definition of extensional equivalence is by induction on the type, and is defined in the same way (as a logical relation) as we did earlier for hereditary termination, the only difference being that hereditary termination was a unary predicate, whereas extensional equivalence is a binary relation.

We first define equivalence for closed terms:

**Products**  $e_1 \sim e_2 : \tau_1 \times \tau_2$  iff `fst`( $e_1$ )  $\sim$  `fst`( $e_2$ ) :  $\tau_1$ , and, `snd`( $e_1$ )  $\sim$  `snd`( $e_2$ ) :  $\tau_2$

**Sums**  $e_1 \sim e_2 : \tau_1 + \tau_2$  iff

- either  $e_1 \mapsto^* \text{inl}_{\tau_1, \tau_2} e'_1$ ,  $e_2 \mapsto^* \text{inl}_{\tau_1, \tau_2} e'_2$  and  $e'_1 \sim e'_2 : \tau_1$ ,
- or  $e_1 \mapsto^* \text{inr}_{\tau_1, \tau_2} e'_1$ ,  $e_2 \mapsto^* \text{inr}_{\tau_1, \tau_2} e'_2$  and  $e'_1 \sim e'_2 : \tau_2$ ,

This definition is extended to open terms as: If  $\Gamma \vdash e_i : \tau$ , then  $e_1 \sim e_2 : \tau[\Gamma]$  iff for all  $\gamma_1, \gamma_2$  such that  $\gamma_1 \sim \gamma_2 : \Gamma$ ,  $\widehat{\gamma}_1(e_1) \sim \widehat{\gamma}_2(e_2) : \tau$

**Question 6.** Show that  $\sim$  is reflexive, i.e., if  $\Gamma \vdash e : \tau$ , then  $e \sim e : \tau[\Gamma]$ . The proof proceeds by induction on the derivation of  $\Gamma \vdash e : \tau$ . We proved this in class for cases related to types  $\tau_1 \rightarrow \tau_2$ , and `nat`. For this question, you only need to show the cases for product and sum types – do the cases for  $\times$  introduction and  $+$  elimination, and any one each of the two cases of  $\times$  elimination and  $+$  introduction. You can assume without proof the head-expansion lemma, but you should state it clearly.