

## Homework 5

15-814: Type Systems for Programming Languages

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

**Out:** October 18, 2009

**Due:** October 23, 2009 (11:59 PM)

### 1 Inductive types

In this part, we will study the covariant action of inductive type operators. A positive type operator  $X.\mu Y.A(X, Y)$  formed by an inductive type is defined by the rule:

$$\frac{\Delta, Y \vdash X.A(X, Y) \text{ pos}}{\Delta \vdash X.(\mu Y.A(X, Y)) \text{ pos}}$$

where  $A(X, Y)$  denotes that type  $A$  has  $X$  and  $Y$  as free type variables.

Instead of giving a general account, we will work with a specific example involving lists. Consider the operator  $X.(\mu Y.1 + X \times Y)$ . The action of this operator on types (defined by substitution, as before) gives types of lists, e.g.  $\mu Y.1 + \text{nat} \times Y$ : a list of natural numbers,  $\mu Y.1 + \text{string} \times Y$ : a list of strings, etc. We will abbreviate  $\mu Y.1 + A \times Y$  as  $\text{list}[A]$ . The functorial action of the operator acts on maps of type  $B_1 \rightarrow B_2$  to produce a map of type  $\text{list}[B_1] \rightarrow \text{list}[B_2]$ . Let us define this map now.

Informally, the map applies  $m : B_1 \rightarrow B_2$  to all elements of the list. This can be defined using the following recursive function:

$$\begin{aligned} \text{listmap } m \text{ inl}_{1, B_1}(\langle \rangle) &= \text{inl}_{1, B_2}(\langle \rangle) \\ \text{listmap } m \text{ inr}_{1, B_1}(\langle h; t \rangle) &= \text{inr}_{1, B_2}(\langle \text{ap}(m; h); \text{listmap } m \ t \rangle) \end{aligned}$$

**Task 1.** Notice that the above function is structurally-recursive in the list argument. Thus we can use the recursion operator `rec` provided by inductive types in order to define `map`. Let  $m : B_1 \rightarrow B_2$ . Define  $\text{map}[X.(\mu Y.1 + X \times Y)](m)$  using `rec` for lists.

**(Hint)** A way to go from  $\text{list}[B_1]$  to  $\text{list}[B_2]$  is via the following diagram, familiar from the definition of inductive types. Try to come up with terms that

can represent the edges in the diagram such that the diagram commutes.:

$$\begin{array}{ccc}
 1 + B_1 \times \text{list}[B_1] & \rightarrow & 1 + B_1 \times \text{list}[B_2] \\
 \downarrow & & \downarrow \\
 & & 1 + B_2 \times \text{list}[B_2] \\
 \downarrow & & \downarrow \\
 \text{list}[B_1] & \longrightarrow & \text{list}[B_2]
 \end{array}$$

## 2 Reynolds' IA

Refer to Chapter 37 of PFPL for a definition of Idealized Algol.

### 2.1 Lazy IA

We will consider a variation of IA, called  $\text{IA}_{\text{lazy}}$ , that uses lazy natural numbers instead of eager numbers. The term syntax for lazy numbers is the same as that for eager numbers:

$$e ::= z \mid \mathbf{s}(e) \mid \mathbf{ifz}(e; e_1; x.e_2)$$

The difference is in definition of values: a number of the form  $\mathbf{s}(e)$  is a value regardless of whether  $e$  is a value:

$$\overline{z \text{ val}_\Sigma} \qquad \overline{\mathbf{s}(e) \text{ val}_\Sigma}$$

**Task 2.** Show by means of an example, that  $\text{IA}_{\text{lazy}}$  is not type safe. The two ways in which a language fails to be type-safe are, first, by a failure of progress theorem, and second, if preservation does not hold. You should be able to design a well-typed expression/command that either gets stuck, or evaluates to an ill-typed expression/command.

**(Hint)** Lazy numbers allow the possibility of leaking an assignable outside the scope of its declaration. With eager numbers this was not possible since an assignable could not syntactically occur in a value of type  $\text{nat}$ . However, with lazy numbers, one can use the cut-rule to embed arbitrary computations inside numbers.

## 2.2 References

We will now add references to assignable variables to **IA** while still maintaining the stack-implementability of the language. We will refer to this language as **IA**{ref}. Please consult Chapter 37 of PFPL for a discussion. We reproduce the key parts here. The only difference between the two presentations is that we use the judgment form  $\Gamma \vdash_{\Sigma} m \sim A \text{ cmd}$  to type commands instead of  $\Gamma \vdash_{\Sigma} m \text{ ok}$ .

We add a new type **Ref** to **IA** to denote the type of references. A reference to an assignable  $a$  is denoted by  $\text{ref}[a]$ , and is associated with two elimination operations,  $\text{getv}(e)$  and  $\text{setv}(e_1; e_2)$ , which get and set the value of the referenced assignable.

$$\begin{array}{c}
\frac{\Gamma \vdash_{\Sigma} m \sim A \text{ cmd}}{\Gamma \vdash_{\Sigma} \text{cmd}(m) : \text{Cmd}[A]} \qquad \frac{\Gamma \vdash_{\Sigma} e : \text{nat}}{\Gamma \vdash_{\Sigma} \text{ret}(e) \sim \text{nat cmd}} \\
\\
\frac{\Gamma \vdash_{\Sigma} e : \text{Cmd}[A_1] \quad \Gamma, x:A_1 \vdash_{\Sigma} m \sim A_2 \text{ cmd}}{\Gamma \vdash_{\Sigma} \text{seq}(e; x.m) \sim A_2 \text{ cmd}} \\
\\
\frac{\Gamma \vdash_{\Sigma} e : \text{nat} \quad \Gamma \vdash_{\Sigma, a} m \sim A \text{ cmd}}{\Gamma \vdash_{\Sigma} \text{dcl}(e; a.m) \sim A \text{ cmd}} \qquad \frac{\Gamma \vdash_{\Sigma} e : \text{nat}}{\Gamma \vdash_{\Sigma} \text{set}[a](e) \sim \text{nat cmd}} \\
\\
\frac{}{\Gamma \vdash_{\Sigma} \text{get}[a] \sim \text{nat cmd}} \qquad \frac{}{\Gamma \vdash_{\Sigma, a} \text{ref}[a] : \text{Ref}} \text{(Ref-I)} \\
\\
\frac{\Gamma \vdash_{\Sigma} e : \text{Ref}}{\Gamma \vdash_{\Sigma} \text{getv}(e) \sim \text{nat cmd}} \text{(Ref-I}_1) \qquad \frac{\Gamma \vdash_{\Sigma} e_1 : \text{Ref} \quad \Gamma \vdash_{\Sigma} e_2 : \text{nat}}{\Gamma \vdash_{\Sigma} \text{setv}(e_1; e_2) \sim \text{nat cmd}} \text{(Ref-I}_2)
\end{array}$$

The dynamics are given by the following rules (we only give rules for  $\text{getv}$  and  $\text{setv}$  here, others can be found in PFPL, Chapter 37):

$$\begin{array}{c}
\frac{}{\text{ref}[a] \text{val}_{\Sigma, a}} \qquad \frac{e \xrightarrow{\Sigma} e'}{\text{getv}(e); \mu \xrightarrow{\Sigma} \text{getv}(e'); \mu} \qquad \frac{}{\text{getv}(\text{ref}[a]); \mu \xrightarrow{\Sigma} \text{get}[a]; \mu} \\
\\
\frac{e_1 \xrightarrow{\Sigma} e'_1}{\text{setv}(e_1; e_2); \mu \xrightarrow{\Sigma} \text{setv}(e'_1; e'_2); \mu} \qquad \frac{}{\text{setv}(\text{ref}[a]; e); \mu \xrightarrow{\Sigma} \text{set}[a](e); \mu}
\end{array}$$

**Task 3.** References can be used to leak assignables outside their scope of declaration, and cause a runtime error. This is prevented by the static type system by restricting commands to only evaluate to a value of type **nat**, as forced by the following rule:

$$\frac{\Gamma \vdash_{\Sigma} e : \text{nat}}{\Gamma \vdash_{\Sigma} \text{ret}(e) \sim \text{nat cmd}}$$

What goes wrong if the above rule is replaced by the following, to allow commands to

return expressions of arbitrary types:

$$\frac{\Gamma \vdash_{\Sigma} e : A}{\Gamma \vdash_{\Sigma} \text{ret}(e) \sim A \text{ cmd}}$$

**(Hint)** Think about what might happen if a reference is returned by a command. You should be able to construct a command that gets stuck due to a *dangling* reference, i.e., a reference to an assignable that does not exist in the memory.

**Task 4.** Instead of extending the language **IA** with a new type **Ref**, we can code up the behavior of references in **IA** itself. Intuitively, a reference can be thought of as a pair of a getter and a setter function, each of which when called, do the get/set operation on the concerned assignable. Define the intro and elim forms for **Ref** in **IA**:

$$\begin{aligned} \text{Ref} & := \dots\dots\dots \\ \text{ref}[a] & := \dots\dots\dots \\ \text{getv}(e) & := \dots\dots\dots \\ \text{setv}(e_1; e_2) & := \dots\dots\dots \end{aligned}$$

You can use product and sum types for your definition. In order to ensure that your definition correctly simulates references, you may need to check that if  $e \xrightarrow{\Sigma} e'$  in **IA**{**ref**}, then  $\ulcorner e \urcorner \xrightarrow{\Sigma}^* \ulcorner e' \urcorner$  in **IA** (and analogously for commands), where  $\ulcorner \cdot \urcorner$  represents your translation. You do not need to prove this property.