Homework 6: Algol and Classification

15-814: Types and Programming Languages
Fall 2015
TA: Evan Cavallo (ecavallo@cs.cmu.edu)

Out: 11/22/15
Due: 12/6/15

1 Modernized Algol

In this section, we'll examine an extension to **MA**, the language with a modal separation between expressions and commands described in PFPL 34. First, let's review the general setup by looking at some familiar data structures.

1.1 References for Mutable Data Structures

In order for mutable structures to be useful, it is better to work in the variant of **MA** with free assignables and references. Recall that free assignable declarations are evaluated by extending the signature, which, in contrast to the dynamics for scoped assignables, is tracked as part of the state.

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\Sigma\{\mathtt{dcl}(e;a.m) \parallel \mu\} \mapsto \nu\Sigma, a \sim \tau\{m \parallel \mu \otimes a \hookrightarrow e\}}\ (\textsc{Dcl-I})$$

The reference type τ `ref` internalizes assignable symbols as data. While the functionality of the reference type can be encoded in **MA** using capabilities (PFPL 35.1), a primitive reference type is better-behaved (and more convenient). Since the reference type cannot safely be mobile, these are mostly useful with free assignables.

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} \mathtt{ref}[a] : \tau\ \mathtt{ref}}\ (\textsc{Ref}) \qquad \frac{\Gamma \vdash_\Sigma e : \tau\ \mathtt{ref}}{\Gamma \vdash_\Sigma \mathtt{getref}(e) \sim \tau}\ (\textsc{Getref})$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau\ \mathtt{ref} \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathtt{setref}(e_1; e_2) \sim \tau}\ (\textsc{Setref})$$

In the first homework, we discussed various interpretations of the following C data structure:

```
typedef struct tree_data tree;

struct tree_data {
  int leaf_value;
  tree *left;
  tree *right;
};
```

Now that we have covered recursive types and mutable state, we can give this a proper treatment. (Note that reference cells introduce the possibility of circular data structures, so this is different from the inductive type of trees.)

Task 1 *Give a definition of the type* `tree*` *in* **MA** *with recursive types and a type* `int`*, as presented above (without making any improvements).*

Task 2 *We previously concluded that trees are better encoded using sum types. For each of the following specifications, define a type τ `tree` of mutable trees with the described behavior. In each case, an element of a tree type should consist either of a leaf with a value of type τ or of two subtrees.*

(a) A mutable tree can be changed to a leaf (by supplying a value of type τ) or to a node (by supplying a new pair of mutable subtrees).

(b) A mutable tree is permanently either a leaf or a node with two subtrees. Leaves cannot be updated. However, a node can be mutated by modifying one of the two subtrees.

(c) A mutable tree can only be updated by providing a whole new tree; its subparts cannot be modified in isolation.

Task 3 *For the encoding of τ `tree` you defined in Task 2(a), define a function*

$$\texttt{tmap} : (\tau \to \tau) \to \tau\ \texttt{tree} \to \texttt{unit cmd}$$

so that `tmap` f t *applies the function f to each leaf in t in place. For the sake of convenience, you may use* `fix`*, as well as any derived operators (such as* $\texttt{do}(e)$ *and* $m_1; m_2$*) that we have discussed in class. You don't have to consider the behavior on* `tmap` *on circular trees.*

Task 4 *For each of the following alternate type specifications below, explain informally whether it is possible to define a term of said type with the same or similar behavior as in the previous task. If it is possible, describe any difference in functionality between the two.*

(a) $(\tau \to \tau) \to \tau\ \texttt{tree} \to \texttt{unit}$

(b) $(\tau \to \tau)\ \texttt{cmd} \to \tau\ \texttt{tree} \to \texttt{unit cmd}$

(c) $(\tau \to \tau\ \texttt{cmd}) \to \tau\ \texttt{tree} \to \texttt{unit cmd}$

(d) $(\tau \to \tau) \to (\tau\ \texttt{tree} \to \texttt{unit})\ \texttt{cmd}$

1.2 Exceptions

In this section, we'll consider adding an exception mechanism to **MA** at the level of commands. Assume we have fixed a type τ_{exn} and are working in **MA** with free assignables. In order to deal with control flow, we will need a control stack-style dynamics, which we will specify via states $k \triangleright m$, representing execution of a command, $k \triangleleft v$, representing normal return, and $k \blacktriangleleft v$, representing exceptional return. Each of these states exists in the context of a signature Σ and memory μ. We will evaluate expressions independently of the memory and control stack with an evaluation dynamics $e \Downarrow_\Sigma v$ (the choice of evaluation dynamics is simply to save space on rules, since expressions are not our focus). For example, we use the following rules for `ret` and `dcl`:

$$\frac{e \Downarrow_\Sigma v}{\nu\Sigma\{\mu \parallel k \triangleright_\Sigma \texttt{ret}(e)\} \mapsto \nu\Sigma\{\mu \parallel k \triangleleft v\}}$$

$$\frac{e \Downarrow_\Sigma v}{\nu\Sigma\{\mu \parallel k \triangleright_\Sigma \texttt{dcl}(e; a.m)\} \mapsto \nu\Sigma, a \sim \tau\{\mu \otimes a \hookrightarrow v \parallel k \triangleright_\Sigma m\}}\ (\textsc{Decl-I})$$

Exceptions are implemented via commands `raise` and `try`.

$$\frac{\Gamma \vdash_\Sigma e : \tau_{\mathsf{exn}}}{\Gamma \vdash_\Sigma \texttt{raise}\{\tau\}(e) \sim \tau}\ (\textsc{Raise}) \qquad \frac{\Gamma \vdash_\Sigma m_1 \sim \tau \quad \Gamma, x : \tau_{\mathsf{exn}} \vdash m_2 \sim \tau}{\Gamma \vdash_\Sigma \texttt{try}(m_1; x.m_2) \sim \tau}\ (\textsc{Try})$$

$$\frac{e \Downarrow_\Sigma v}{\nu\Sigma\{\mu \parallel k \triangleright \texttt{raise}\{\tau\}(e)\} \mapsto \nu\Sigma\{\mu \parallel k \blacktriangleleft v\}}$$

$$\frac{}{\nu\Sigma\{\mu \parallel k \triangleright \texttt{try}(m_1; x.m_2)\} \mapsto \nu\Sigma\{\mu \parallel k; \texttt{try}(-; x.m_2) \triangleright m_1\}}$$

$$\frac{}{\nu\Sigma\{\mu \parallel k; \texttt{try}(-; x.m_2) \triangleleft v\} \mapsto \nu\Sigma\{\mu \parallel k \triangleleft v\}}$$

$$\frac{}{\nu\Sigma\{\mu \parallel k; \texttt{try}(-; x.m_2) \blacktriangleleft v\} \mapsto \nu\Sigma\{\mu \parallel k \triangleright [v/x]m_2\}}$$

Task 5 *Give control stack dynamics rules for* `bnd`*. (Remember to handle cases involving exceptions!)*

Task 6 *We can also add exceptions to* **MA** *with* scoped *assignables. In this setup,* (Dcl-I) *is different: rather than reducing a declaration* $\texttt{dcl}(v; a.m)$ *by adding* $a \hookrightarrow v$ *to the memory and deleting the declaration, we push the declaration onto the stack and continue as* m*. As a result, we can do away with the memory completely and instead maintain the values of assignables on the control stack. In this version, we'll use states* $k \triangleright m$*,* $k \triangleleft v$*, and* $k \blacktriangleleft v$*, each in a signature* $\nu\Sigma\{-\}$ *(but without a memory).*

$$\frac{e \Downarrow_\Sigma v}{\nu\Sigma\{k \triangleright_\Sigma \texttt{dcl}(e; a.m)\} \mapsto \nu\Sigma, a \sim \tau\{k; \texttt{dcl}(v; a.-) \triangleright_\Sigma m\}} \text{ (Decl-I)}$$

(a) *What restriction to the exception setup is necessary to ensure type safety if we use scoped assignables? Give an example of how type safety can fail otherwise.*

(b) *Finish the set of dynamics rules for* $\texttt{dcl}(v; a.m)$ *for this setup, and give rules for* $\texttt{get}[a]$ *and* $\texttt{set}[a](e)$*. (You may find it useful to define auxiliary judgments to search for and update assignable values in the control stack.)*

(c) *For which of the rules you gave in (b) is the restriction you described in (a) necessary for preservation? Why?*

2 Exceptions from Fluid Binding

In this section, we implement exceptions in a language with continuations and *fluid binding*, a restricted form of memory which is, incidentally, a principled presentation of *dynamic scope*. We will work in a system with a single fluid memory cell, which is enough for our purposes; a full treatment can be found in PFPL 32. The extension adds the expressions $\texttt{putf}(e_1; e_2)$ and `getf`, with the following typing rules:

$$\frac{\Gamma \vdash e_1 : \tau_{\mathsf{fluid}} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{putf}(e_1; e_2) : \tau} \qquad \frac{}{\Gamma \vdash \texttt{getf} : \tau_{\mathsf{fluid}}}$$

Here τ_{fluid} is some fixed type. (If we had multiple fluid cells, there would be no reason to restrict to a single type.) The expressions $\texttt{putf}(e_1; e_2)$ sets the value of the fluid cell to e_1 in the evaluation of e_2, while `getf` gets the current value in the fluid cell. The control stack dynamics rules are as follows.

$$\frac{}{k \triangleright \texttt{putf}(e_1; e_2) \mapsto k; \texttt{putf}(-; e_2) \triangleright e_1} \qquad \frac{}{k; \texttt{putf}(-; e_2) \triangleleft v_1 \mapsto k; \texttt{putf}(v_1; -) \triangleright e_2}$$

$$\frac{}{k; \texttt{putf}(v_1; -) \triangleleft v_2 \mapsto k \triangleleft v_2} \qquad \frac{\texttt{fluid}(k) \Rightarrow v}{k \triangleright \texttt{getf} \mapsto k \triangleleft v} \qquad \frac{\texttt{fluid}(k) \not\Rightarrow}{(k \triangleright \texttt{getf})\ \texttt{err}}$$

It is necessary to include an error judgment s err for the case that getf is called before the fluid cell has been set to any value. (The dynamics must then include the obvious error propagation rules, which we will not enumerate.) The judgments $\texttt{fluid}(k) \Rightarrow v$ and $\texttt{fluid}(k) \not\Rightarrow$, which are used to search for the fluid cell's current value in the stack, are defined by the following rules.

$$\frac{}{\texttt{fluid}(k;\texttt{putf}(v;-)) \Rightarrow v} \qquad \frac{f \neq \texttt{putf}(v';-) \quad \texttt{fluid}(k) \Rightarrow v}{\texttt{fluid}(k;f) \Rightarrow v}$$

$$\frac{}{\texttt{fluid}(\epsilon) \not\Rightarrow} \qquad \frac{f \neq \texttt{putf}(v';-) \quad \texttt{fluid}(k) \not\Rightarrow v}{\texttt{fluid}(k;f) \not\Rightarrow v}$$

Observe that the fluid cell behaves like a "dynamically-scoped variable." For example (assuming the necessary language constructs exist), the term

$$\texttt{putf}(10;\texttt{let}\ f = \texttt{putf}(1;\lambda x{:}\texttt{nat}.\ \texttt{getf} + x)\ \texttt{in}\ f(1))$$

should evaluate to 11, even though the value of the fluid cell is 1 at the time the function is defined, because at the time it is called the assignment $\texttt{putf}(1;-)$ has left the control stack.

Task 7 *Using fluid binding and continuations (with* letcc *and* throw*), define encodings of the expressions* $\texttt{raise}\{\tau\}(e)$ *and* $\texttt{try}(e_1;x.e_2)$*. Assume a type* τ_{exn} *is given; you choose* τ_{fluid}*. You can also assume that the fluid cell is not being used for other purposes. If an exception reaches the toplevel, it should result in an* err *state.*

3 Dynamic Classification for Existentials

In this section, we will investigate the possibility of encoding existential types using dynamic classification. (A result in the other direction, implementing clsfd using existentials and symbolic references, is in PFPL 33.3.) The basis for the encoding is the following implementation of the existential type:

$$\exists t.\tau \triangleq [\texttt{clsfd}/t]\tau$$

In order to define the operations on the existential type, however, we will have to restrict the form of the operator $t.\tau$.

Task 8 *Define* pack *and* open *for this encoding, assuming that* $t.\tau$ *is a positive type operator.*

Task 9 *The condition that* $t.\tau$ *is positive is sufficient, but not necessary. Give definitions of* pack *and* open *for the operator* $t.t \times (t \to \texttt{nat})$*, in which the first occurence of* t *is positive but the second is negative.*

Task 10 *In class, we discussed the dual concepts of* integrity *and* confidentiality *with respect to dynamic classification. The integrity of classified values is determined by which parties have access to the constructor* $\texttt{in}[a]$*, while confidentiality concerns access to the destructor* $\texttt{isin}[a]$*. Describe the role these play in your answers to Tasks 8 and 9.*