15-814 Homework 6

December 6, 2017

1 Modernized Algol

Note: this section should have explicitly referred to the version of MA given in class, where we used `bnd(m;x.m)` and `force(e)` rather than `bnd(e;x.m)`.

Task 1 *Give a definition of the type* `tree*` *in* **MA** *with recursive types and a type* `int` *as presented above. (Hint: your answer should adequately handle null pointers.)*

Solution: `tree`$*$ is a pointer to a tree, which can be a null pointer (represented with an option type). `tree`$* = \mu\alpha.$`unit` $+$ (`int` $\times \alpha \times \alpha$) `ref`

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 its two subtrees.

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

Solution: The immutable tree is given by $\mu\alpha.\tau + \alpha \times \alpha$. The mutability behavior is controlled by placing `ref` in appropriate positions (i.e. at the parts of the tree that can be modified).

(a) τ `tree` $= \mu\alpha.(\tau + \alpha \times \alpha)$ `ref`

(b) τ `tree` $= \mu\alpha.\tau + \alpha$ `ref` $\times \alpha$ `ref`

(c) τ `tree` $= (\mu\alpha.\tau + \alpha \times \alpha)$ `ref`

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. You may use* `fix` *(see Homework 5) in your answer. Briefly explain the intuition behind your answer. You do not have to consider the behavior on* `tmap` *on circular trees.*

Solution: In ML:

```
(* This encodes that we can change an entire branch to a leaf or vice versa *)
datatype 'a tr = Tree of 'a tn ref
     and 'a tn = Leaf of 'a | Branch of ('a tr) * ('a tr);

fun tmap f (Tree e) =
  let val u = !e in
    case u of
      Leaf v => e := Leaf (f v)
    | Branch (l,r) =>
      (tmap f l;tmap f r)
  end;
```

Writing $t = (\tau \to \tau) \to \tau$ `tree` $\to$ `unit cmd`. The following are abbreviations (with free variables) so that the final definition of `tmap` can fit on a line.
On a leaf, we replace the leave with the result of applying f:

$$\texttt{lf} = \texttt{cmd}(\texttt{bnd}(\texttt{unfold}(e)*=\texttt{inl}((f\ v)); \_.\texttt{ret}(\langle\rangle)))$$

On a branch, we **do not** modify the branch, but recurse into its sub-trees:

$$\texttt{br} = \texttt{cmd}(\texttt{bnd}(\texttt{force}(m\ f\ \pi_1 b); \_.\texttt{force}(m\ f\ \pi_2 b)))$$

Putting everything together:

$$\texttt{tmap} = \texttt{fix}\{t\}(m.\lambda f{:}\tau \to \tau.\lambda e{:}\tau\ \texttt{tree.cmd}(\texttt{bnd}(*\texttt{unfold}(e); u.\texttt{force}(\texttt{case}\ u\ \{v.\texttt{lf}; b.\texttt{br}\}))))$$

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) $(\tau \to \tau) \to \tau$ `tree` $\to$ `unit`

(b) $(\tau \to \tau)$ `cmd` $\to \tau$ `tree` $\to$ `unit cmd`

(c) $(\tau \to \tau$ `cmd`$) \to \tau$ `tree` $\to$ `unit cmd`

(d) $(\tau \to \tau) \to (\tau$ `tree` $\to$ `unit`$)$ `cmd`

Solution:

(a) Not possible to do the tree update in place since the type does not have `cmd` anywhere in it. A function of this type cannot contain encapsulated commands, and so cannot modify the existing references. It can, however, recreate a brand new tree with the leaf values updated.

(b) This is possible. Whenever f is called above, we replace it with a bind instead. This gives the function being mapped the chance to produce side effects whenever it is called. Alternatively, we can bind it once at the start to get the side effect only once.

(c) This is also possible. In this case, whenever we have $f\ v$, we replace it with a bind for its side effects. This rules out the second possibility in (b) above. On the other hand, the side effect produced can now depend on v.

(d) Not possible. If we had `tmap` f, then the only thing we can do with it is execute it to get a function of type τ `tree` $\to$ `unit`. However, this execution cannot possibly modify the tree since it has not even been passed the tree yet. Moreover, the resulting function's type does not have `cmd` so we are back to the same reasoning as (a).

1.1 Exceptions

Task 5 *Give control stack dynamics rules for* `bnd`*. (Hint: remember to handle cases involving exceptions.)*

Solution:

$$\frac{}{\mu \parallel k \rhd \texttt{bnd}(m_1; x.m_2) \mapsto \mu \parallel k; \texttt{bnd}(-; x.m_2) \rhd m_1} \text{ (Bnd-M)}$$

$$\frac{}{\mu \parallel k; \texttt{bnd}(-; x.m) \lhd v \mapsto \mu \parallel k \rhd [v/x]m} \text{ (Bnd-Ret)}$$

$$\frac{}{\mu \parallel k; \texttt{bnd}(-; x.m) \blacktriangleleft v \mapsto \mu \parallel k \blacktriangleleft v} \text{ (Bnd-Exn)}$$

Task 6 *We can also add exceptions to* **MA** *with* scoped *assignables. Recall that in this setup,* (Dcl-I) *is different: rather than reducing a declaration* $\texttt{dcl}(v; a.m)$ *by adding* $a \hookrightarrow v$ *to the store and deleting the declaration, we push the declaration onto the stack and continue as* m*. As a result, we can do away with the store and instead maintain the values of assignables on the control stack. In this version, we will use states* $k \rhd m$*,* $k \lhd v$*, and* $k \blacktriangleleft v$*.*

$$\frac{e \Downarrow v}{k \rhd \texttt{dcl}(e; a.m) \mapsto k; \texttt{dcl}(v; a.-) \rhd m} \text{ (Decl-I)}$$

(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 $\texttt{dcl}(e; a.m)$ *for this setup. Give rules for getting an assignable (*$@a$*) and setting an assignable (*$a := e$*). (Hint: you may find it useful to define auxiliary judgments to search for and update assignable values in the control stack.)*

(c) Which of the rules you gave in (b) is the restriction you described in (a) necessary for type preservation? Why?

Solution:

(a) The type chosen for τ_{exn} must be a mobile type. Otherwise, it would be possible to leak assignables out via the exception mechanism. As an example, suppose that τ_{exn} were `nat cmd`. Then we could write $\texttt{try}(\texttt{dcl}(e; a.\texttt{raise}\{\tau\}(\texttt{cmd}(@a))); x.m)$. Inside m, we could mention a outside its scope.

(b) Note that the definitions for `read` and `write` implicitly force the assignable being read/written to be on the stack.

$$\frac{}{k; \texttt{dcl}(v; a.-) \lhd v' \mapsto k \lhd v'} \text{ (Decl-Ret)}$$

$$\frac{}{k; \texttt{dcl}(v; a.-) \blacktriangleleft v' \mapsto k \blacktriangleleft v'} \text{ (Decl-Exn)}$$

$$\frac{}{\texttt{read}\ a\ (k; \texttt{dcl}(v; a.-)) \downarrow v} \text{ (Stack-Read1)} \qquad \frac{\texttt{read}\ a\ k \downarrow v \quad f \text{ is not of the form } \texttt{dcl}(v; a.-)}{\texttt{read}\ a\ (k; f) \downarrow v} \text{ (Stack-Read2)}$$

$$\frac{\texttt{read}\ a\ k \downarrow v}{k \rhd @a \mapsto k \lhd v} \text{ (Get)}$$

$$\frac{}{\texttt{write}\ a\ v'\ (k;\texttt{dcl}(v;a.-)) \uparrow (k;\texttt{dcl}(v';a.-))}\ (\textsc{Stack-Write1})$$

$$\frac{\texttt{write}\ a\ v'\ k \uparrow k' \quad f \text{ is not of the form } \texttt{dcl}(v;a.-)}{\texttt{write}\ a\ v'\ (k;f) \uparrow (k';f)}\ (\textsc{Stack-Write2})$$

$$\frac{e \Downarrow v \quad \texttt{write}\ a\ v\ k \uparrow k'}{k \rhd a := e \mapsto k' \lhd v}\ (\textsc{Set})$$

(c) It is necessary for (Decl-Exn). Following the example in (a), if we had an immobile `nat cmd` for the exception type, then in (Decl-Exn) we would be taking a step with the exception value $v' = \texttt{cmd}(\texttt{ret}(a))$. The type is not preserved after the step because a is no longer in the signature for the stack.

2 Continuations

Task 7 *Define programs with the following types. You may use* `ccwf` *in your definitions.*

1. $\texttt{lem} : \tau + \tau\ \texttt{cont}$
2. $\texttt{dne} : \tau\ \texttt{cont}\ \texttt{cont} \to \tau$
3. $\texttt{cps} : (\tau_1 \to \tau_2) \to (\tau_1\ \texttt{cont} + \tau_2)$

(Note: if we interpret these types as propositions by treating $(-)$ `cont` *as logical negation* $\neg(-)$*, then these are tautologies of classical propositional logic.)*

Solution:

1. In the following, we can type $\lambda x{:}\tau.\texttt{inl}(x)$ with $\tau \to (\tau + \tau\ \texttt{cont})$.
   $\texttt{lem} = \texttt{letcc}\{\tau + \tau\ \texttt{cont}\}(x.\texttt{inr}(\texttt{ccwf}\ (\lambda y{:}\tau.\texttt{inl}(y))\ x))$
   LEM lets us do the next two by "casing" on $P \vee \neg P$.
2. $\texttt{dne} = \lambda x{:}\tau\ \texttt{cont}\ \texttt{cont}.\texttt{case}\ \texttt{lem}\ \{l.l; r.\texttt{throw}\{\tau\}(r;x)\}$
3. For this we use `lem` at type τ_2.
   $\texttt{cps} = \lambda f{:}\tau_1 \to \tau_2.\texttt{case}\ \texttt{lem}\ \{l.\texttt{inr}(l); r.\texttt{inl}(\texttt{ccwf}\ f\ r)\}$

Task 8 *Take* $\tau = \texttt{int}$ *in the previous task, and consider the following expression* $e : \texttt{int}$ *defined using your implementation of* `lem`*.*

$$\begin{array}{rcl} e & \triangleq & \texttt{case lem}\ \{ \\ & & \quad \texttt{inl}(x) \hookrightarrow 2 * x; \\ & & \quad \texttt{inr}(c) \hookrightarrow \texttt{throw}\{\texttt{int}\}(6;c) \\ & & \} \end{array}$$

What is the result of evaluating $\epsilon \rhd e$*? You do not have to write out the stack machine steps, but give an informal explanation of the evaluation process.*

Solution:

1. The case expression (with a hole in place of `lem`) is pushed onto the stack and we evaluate `lem`.

2. Evaluating `lem` returns $\texttt{inr}(\texttt{ccwf}\ (\lambda y{:}\tau.\texttt{inl}(y))\ x)$, where x is the stack containing the above-mentioned case expression.

3. We put this value back into the hole in the case expression.

4. Since this is the right of a sum, the case expression takes the right case and so 6 is thrown to the continuation, $\texttt{ccwf}\ (\lambda y{:}\tau.\texttt{inl}(y))\ x$. By the property of `ccwf`, this eventually results in $\texttt{inl}(6)$ being thrown to x.

5. Now, we are again in the case expression (that was captured in x), but now we have the left case instead. Therefore, we get the result 12 in the final stack.

2.1 Translating Continuations

Task 9 *Define translation rules for the product type $\tau_1 \times \tau_2$ (i.e., for pairing and the two projections). Make sure your definitions have the correct types.*

Solution:
Define $\texttt{curry} = \lambda f{:}\tau_1 \times \tau_2 \to \tau.\lambda x{:}\tau_1.\lambda y{:}\tau_2.f\ \langle x, y\rangle$. This makes the continuation k in the conclusion curried, so we can partially apply it.

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow \hat{e_1} \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \hat{e_2}}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2 \rightsquigarrow \lambda k{:}||\tau_1 \times \tau_2|| \to \rho.\hat{e_1}(\lambda p{:}||\tau_1||.\hat{e_2}((\texttt{curry}\ k)\ p))}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow \hat{e}}{\Gamma \vdash \pi_1 e : \tau_1 \rightsquigarrow \lambda k{:}||\tau_1|| \to \rho.\hat{e}(\lambda p{:}||\tau_1|| \times ||\tau_2||.k\ (\pi_1 p))}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow \hat{e}}{\Gamma \vdash \pi_2 e : \tau_2 \rightsquigarrow \lambda k{:}||\tau_2|| \to \rho.\hat{e}(\lambda p{:}||\tau_1|| \times ||\tau_2||.k\ (\pi_2 p))}$$

Task 10 *Give a translation rule for function application. For sake of intuition, you may find it useful to note the following: if we expand the definition of $||\tau_2 \to \tau_1||$ another step as $||\tau_2|| \to (||\tau_1|| \to \rho) \to \rho$, we see that a translated function takes an* argument *of type $||\tau_2||$ and a* "return address" continuation *which specifies how to continue with the result of type $||\tau_1||$.*

Solution:

$$\overbrace{||\tau_2 \to \tau_1|| = ||\tau_2|| \to \overbrace{(||\tau_1|| \to \rho) \to \rho}^{|\tau_1|}}^{||\tau_2 \to \tau_1|| = ||\tau_2|| \to (||\tau_1|| \to \rho) \to \rho}$$

Under the translation, we have $\hat{e_1} : \big(\big((||\tau_2|| \to \overbrace{(||\tau_1|| \to \rho) \to \rho}^{|\tau_1|}\big)\big) \to \rho\big) \to \rho$, and $\hat{e_2} : (||\tau_2|| \to \rho) \to \rho$.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow \hat{e_1} \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \hat{e_2}}{\Gamma \vdash e_1\ e_2 : \tau_1 \rightsquigarrow \lambda k{:}||\tau_1|| \to \rho.\hat{e_1}\big(\lambda x{:}||\tau_2 \to \tau_1||.\hat{e_2}(\lambda y{:}||\tau_2||.x\ y\ k)\big)}$$

I also accepted the following answer with reversed evaluation order for e_1, e_2 although strictly speaking, this would have different operational behavior:

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow \hat{e_1} \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \hat{e_2}}{\Gamma \vdash e_1\ e_2 : \tau_1 \rightsquigarrow \lambda k{:}||\tau_1|| \to \rho.\hat{e_2}\big(\lambda x{:}||\tau_2||.\hat{e_1}(\lambda y{:}||\tau_2 \to \tau_1||.y\ x\ k)\big)}$$

Task 11 *Give translation rules for $\texttt{letcc}\{\tau\}(x.e)$ and $\texttt{throw}\{\tau'\}(e_1; e_2)$. These are actually quite simple.*

Solution:

$$\frac{\Gamma, x : \tau\ \texttt{cont} \vdash e : \tau \rightsquigarrow \hat{e}}{\Gamma \vdash \texttt{letcc}\{\tau\}(x.e) : \tau \rightsquigarrow \lambda x{:}||\tau|| \to \rho.\hat{e}\ x}$$

The following translation for `throw` is the straightforward answer (it is also well-typed) but it has the wrong order of evaluation:

$$\frac{\Gamma \vdash e_1 : \tau \rightsquigarrow \hat{e_1} \quad \Gamma \vdash e_2 : \tau \texttt{ cont} \rightsquigarrow \hat{e_2}}{\Gamma \vdash \texttt{throw}\{\tau'\}(e_1; e_2) : \tau' \rightsquigarrow \lambda\_{:}||\tau'|| \to \rho.\hat{e_2}\ \hat{e_1}}$$

We really want to evaluate $\hat{e_1}$ first:

$$\frac{\Gamma \vdash e_1 : \tau \rightsquigarrow \hat{e_1} \quad \Gamma \vdash e_2 : \tau \texttt{ cont} \rightsquigarrow \hat{e_2}}{\Gamma \vdash \texttt{throw}\{\tau'\}(e_1; e_2) : \tau' \rightsquigarrow \lambda\_{:}||\tau'|| \to \rho.e_1(\lambda x{:}||\tau||.\hat{e_2}(\lambda y{:}||\tau|| \to \rho.y\ x))}$$

Note that in both cases, we discard the input continuation.