Homework 6: Algol and Continuations

15-814: Types and Programming Languages
Fall 2017
Instructor: Karl Crary
TA: Yong Kiam Tan

Out: Nov 15, 2017 (10 pm)
Due: Dec 4, 2017 (11 pm)

Notes:

- Welcome to 15-814's sixth homework assignment!
- Please email your work as a PDF file to `yongkiat@cs.cmu.edu` titled "15-814 Homework 6". Your PDF should be named "<your-name>-hw6-sol.pdf".

1 Modernized Algol

In this section, we will examine an extension to **MA**, the language with a modal separation between expressions and commands described in PFPL 34.

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. As we did in class, we will omit the signature for assignables[1]. The rule for declaring a new (free) assignable is shown below:

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

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.

$$\frac{}{\Gamma \vdash \&a : \tau \ \mathtt{ref}} \ (\textsc{Ref})$$

$$\frac{\Gamma \vdash e : \tau \ \mathtt{ref}}{\Gamma \vdash *e \div \tau} \ (\textsc{Getref}) \qquad \frac{\Gamma \vdash e_1 : \tau \ \mathtt{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 *= e_2 \div \tau} \ (\textsc{Setref})$$

Consider the following C data structure:

```
typedef struct tree_data tree;

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

[1] See PFPL 35.3 for a complete treatment.

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. (Hint: your answer should adequately handle null pointers.)*

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.

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.*

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 will 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 a control stack-style dynamics, which we will specify via states $k \rhd m$, representing execution of a command, $k \lhd v$, representing normal return, and $k \blacktriangleleft v$, representing exceptional return. Each of these will be accompanied by a store μ.

Expressions will be evaluated independently of the store and control stack with an evaluation dynamics $e \Downarrow v$ (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`:

$$\frac{e \Downarrow v}{\mu \parallel k \rhd \mathtt{ret}(e) \mapsto \mu \parallel k \lhd v} \text{ (Ret-I)}$$

$$\frac{e \Downarrow v}{\mu \parallel k \rhd \mathtt{dcl}(e; a.m) \mapsto \mu \otimes a \hookrightarrow v \parallel k \rhd m} \text{ (Decl-I)}$$

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

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

$$\frac{e \Downarrow v}{\mu \parallel k \rhd \mathtt{raise}\{\tau\}(e) \mapsto \mu \parallel k \blacktriangleleft v}$$

$$\frac{}{\mu \parallel k \rhd \mathtt{try}(m_1; x.m_2) \mapsto \mu \parallel k; \mathtt{try}(-; x.m_2) \rhd m_1}$$

$$\frac{}{\mu \parallel k; \mathtt{try}(-; x.m_2) \lhd v \mapsto \mu \parallel k \lhd v}$$

$$\frac{}{\mu \parallel k; \mathtt{try}(-; x.m_2) \blacktriangleleft v \mapsto \mu \parallel k \rhd [v/x]m_2}$$

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

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* $\mathtt{dcl}(v; a.m)$ *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* $k \blacktriangleleft v$*.*

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

(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 $\mathtt{dcl}(e; a.m)$ *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?

2 Continuations

In this section, we will write some programs using continuations and show how to encode continuations in a language without them. We will start with a language with continuations which is evaluated using a control stack.

$$\begin{array}{lcl} \tau & ::= & \texttt{nat} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \tau\ \texttt{cont} \\ e & ::= & \cdots \mid \texttt{cont}(k) \mid \texttt{letcc}\{\tau\}(x.e) \mid \texttt{throw}\{\tau\}(e; e) \\ k & ::= & \epsilon \mid k; f \\ f & ::= & \cdots \mid \texttt{throw}\{\tau\}(-; e) \mid \texttt{throw}\{\tau\}(e; -) \end{array}$$

We use an eager evaluation strategy for all of the constructs in the language. Evaluation of continuations is given by the following rules (see PFPL 30.2 for more details and statics rules).

$$\frac{}{k \rhd \texttt{cont}(k') \longmapsto k \lhd \texttt{cont}(k')} \qquad \frac{}{k \rhd \texttt{letcc}\{\tau\}(x.e) \longmapsto k \rhd [\texttt{cont}(k)/x]e}$$

$$\frac{}{k \rhd \texttt{throw}\{\tau\}(e_1; e_2) \longmapsto k; \texttt{throw}\{\tau\}(-; e_2) \rhd e_1}$$

$$\frac{e_1\ \mathsf{val}}{k; \texttt{throw}\{\tau\}(-; e_2) \lhd e_1 \longmapsto k; \texttt{throw}\{\tau\}(e_1; -) \rhd e_2} \qquad \frac{e_1\ \mathsf{val}}{k; \texttt{throw}\{\tau\}(e_1; -) \lhd \texttt{cont}(k') \longmapsto k' \lhd e_1}$$

2.1 Programming with Continuations

To start off, we will write a few basic programs with continuations. An example which may be useful in Task 7, is the function $\texttt{ccwf} : (\tau_1 \to \tau_2) \to (\tau_2\ \texttt{cont} \to \tau_1\ \texttt{cont})$, "compose continuation with function."

$$\begin{array}{l} \texttt{ccwf} \triangleq \lambda f{:}\tau_1 \to \tau_2.\lambda k{:}\tau_2\ \texttt{cont}. \\ \qquad \texttt{letcc}\{\tau_1\ \texttt{cont}\}(ret.\texttt{throw}\{\tau_1\ \texttt{cont}\}(f(\texttt{letcc}\{\tau_1\}(r.\texttt{throw}\{\tau_1\}(r; ret))); k)) \end{array}$$

How does this function work? Given $f : \tau_1 \to \tau_2$ and $k : \tau_2\ \texttt{cont}$, let us consider what would happen if we threw some value $e : \tau_1$ to the continuation $\texttt{ccwf}\ f\ k : \tau_1\ \texttt{cont}$. First, we push the `throw` onto the stack. (For our purposes, it does not matter what type we give the `throw`.)

$$\begin{array}{rcl} & \epsilon & \rhd\ \texttt{throw}\{\ldots\}(e; \texttt{ccwf}\ f\ k) \\ \mapsto^* & \epsilon; \texttt{throw}\{\ldots\}(e; -) & \rhd\ \texttt{ccwf}\ f\ k \end{array}$$

Now we evaluate $\texttt{ccwf}\ f\ k$. After stepping through the application to f and k, we save the current continuation $(\epsilon; \texttt{throw}\{\ldots\}(e; -))$ in the variable ret (which we will continue to write as ret for sake of space). We then begin computing a value of type τ_2 which we will eventually throw to k.

$$\mapsto^* \quad \epsilon; \texttt{throw}\{\ldots\}(e; -); \texttt{throw}\{\tau_2\}(-; k)\ \rhd\ f(\texttt{letcc}\{\tau_1\}(r.\texttt{throw}\{\tau_1\}(r; ret)))$$

Assuming f is a value, we push an application frame onto the stack. We again save the current continuation, which at this point expects a value of type τ_1, in the variable r.

$$\begin{array}{rcl} \mapsto^* & \epsilon; \texttt{throw}\{\ldots\}(e; -); \texttt{throw}\{\tau_2\}(-; k); f(-) & \rhd\ \texttt{letcc}\{\tau_1\}(r.\texttt{throw}\{\tau_1\}(r; ret)) \\ \mapsto^* & \epsilon; \texttt{throw}\{\ldots\}(e; -); \texttt{throw}\{\tau_2\}(-; k); f(-) & \rhd\ \texttt{throw}\{\tau_1\}(r; ret) \end{array}$$

Finally, we throw r to the continuation ret. The current stack is erased and replaced with the stack stored in ret, which was $\epsilon; \texttt{throw}\{\tau_1\}(e; -)$.

$$\mapsto^* \quad \epsilon; \texttt{throw}\{\ldots\}(e; -)\ \rhd\ r$$

Since r is a value, we execute the `throw` at the top of the stack. It replaces the stack with r, which is $\epsilon; \texttt{throw}\{\ldots\}(e; -); \texttt{throw}\{\tau_2\}(-; k); f(-)$, and continues by evaluating e.

$$\mapsto^* \quad \epsilon; \texttt{throw}\{\ldots\}(e; -); \texttt{throw}\{\tau_2\}(-; k); f(-) \quad \triangleright \quad e$$

If $f(e)$ evaluates to some value v, then after said evaluation takes place, we execute the `throw` at the top of the stack, which throws v to k, leaving us in the state

$$\mapsto^* \quad k \quad \triangleright \quad v$$

In the end, throwing a value e to $(\texttt{ccwf}\ f\ k)$ amounts to throwing $f(e)$ to k.

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 $(-)\ \texttt{cont}$ *as logical negation* $\neg(-)$*, then these are tautologies of classical propositional logic.)*

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

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

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

2.2 Translating Continuations

In this section, we will show how to translate programs in a language **K** with continuations into a language **I** without them. The theorem we prove will have the following form:

Theorem 1 *Fix a type* ρ *in* **I**. *There are translations* $||-||$, $|-|$ *from types in* **K** *to types in* **I** *and a translation* $\hat{-}$ *from well-typed terms in* **K** *to terms in* **I**, *all defined in terms of* ρ. *If* $\Gamma \vdash_{\mathbf{K}} e : \tau$, *then* $||\Gamma|| \vdash_{\mathbf{I}} \hat{e} : |\tau|$. *(Here,* $||-||$ *is extended to a translation of contexts by translating the type of each variable.)*

Of course, the value of this theorem depends on how we define $||-||$, $|-|$ and $\hat{-}$. (Our definition will satisfy a correctness theorem comparing the results of evaluating e and $\hat{e}$, but we will not discuss this here.) We start by defining the translations of types $||\tau||$ and $|\tau|$. These are defined by mutual recursion. First, we have

$$|\tau| \quad \triangleq \quad (||\tau|| \to \rho) \to \rho$$

The type ρ represents the type of the "final result." A value of type $|\tau|$ accepts a "$||\tau||$ continuation," a term which computes a ρ from a $||\tau||$, and uses it to compute such a ρ. $||\tau||$ is defined inductively on the structure of τ. Assume that $\mathbf{K}$ and $\mathbf{I}$ support natural numbers, products, and functions. We define

$$\begin{array}{rcl} ||\texttt{nat}|| & \triangleq & \texttt{nat} \\ ||\tau_1 \times \tau_2|| & \triangleq & ||\tau_1|| \times ||\tau_2|| \\ ||\tau_2 \to \tau_1|| & \triangleq & ||\tau_2|| \to |\tau_1| \\ ||\tau\ \texttt{cont}|| & \triangleq & ||\tau|| \to \rho \end{array}$$

The translation of context $|\Gamma|$ is simply given by translating the type of each variable. Now, we will given the translation of expressions. This is a *type-directed translation* – the translation of some expression e with $\Gamma \vdash_{\mathbf{K}} e : \tau$ is defined by induction on its typing derivation. We will specify it by defining a judgment $\Gamma \vdash e : \tau \rightsquigarrow \hat{e}$, which expresses that $\Gamma \vdash_{\mathbf{K}} e : \tau$ translates to $\hat{e}$ and has a case for each typing rule in $\mathbf{K}$. After giving the definition, we will show that $||\Gamma|| \vdash_{\mathbf{I}} \hat{e} : |\tau|$. We start with the rules for variables and natural numbers.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow \lambda k.\ k(x)}\ (\textsc{Tr-Var})$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat} \rightsquigarrow \lambda k.\ k(\texttt{z})}\ (\textsc{Tr-Z}) \qquad \frac{\Gamma \vdash e : \texttt{nat} \rightsquigarrow \hat{e}}{\Gamma \vdash \texttt{s}(e) : \texttt{nat} \rightsquigarrow \lambda k.\ \hat{e}(\lambda x.\ k(\texttt{s}(x)))}\ (\textsc{Tr-S})$$

$$\frac{\Gamma \vdash e : \texttt{nat} \rightsquigarrow \hat{e} \quad \Gamma \vdash e_0 : \tau \rightsquigarrow \hat{e}_0 \quad \Gamma, x : \texttt{nat} \vdash e_1 : \tau \rightsquigarrow \hat{e}_1}{\Gamma \vdash \texttt{ifz}(e; e_0; x.e_1) : \tau \rightsquigarrow \lambda k.\ \hat{e}(\lambda y.\ \texttt{ifz}(y; \hat{e}_0 k; x.\hat{e}_1 k))}\ (\textsc{Tr-Ifz})$$

To understand these rules, remember that each translation $\hat{e}$ should be an expression of type $|\tau| = (||\tau|| \to \rho) \to \rho$, a function which takes a "continuation" and returns a result (a term of type ρ). It is important to distinguish $||\tau||$ from $(||\tau|| \to \rho) \to \rho$: although there is an obvious embedding from the former into the latter, taking e to $\lambda k.\ k(e)$, such a transformation is not reversible in general. In other words, while we might think of $\hat{e}$ as a function that, given a "continuation" $k : ||\tau|| \to \rho$, supplies "e" to k, this is not always accurate (although it is true in the case of the rules (Tr-Var) and (Tr-Z)). For one thing, this would require translating e into a $\mathbf{I}$ term of type $||\tau||$, which is not always possible. Moreover, $\hat{e}$ can use k in other ways. We can more accurately say that $\hat{e} : |\tau|$ will compute a ρ *if we tell it how we would compute a* ρ *given a* $||\tau||$, but it may accomplish this in ways other than "feeding in e."

With this in mind, we can go through the rules for $\texttt{nat}$. The translation for $\texttt{z}$ is simple: given a "continuation" $k : \texttt{nat} \to \rho$, it simply invokes k on $\texttt{z}$. For $\texttt{s}(e)$, we start by translating e into $\hat{e}$. After taking an argument "continuation" $k : \texttt{nat} \to \rho$, we apply $\hat{e}$ to $\lambda x.\ k(\texttt{s}(x))$, the composition $k \circ \texttt{s}$. Put informally, we tell $\hat{e}$ that we want to "use e" by first taking its successor and then continuing with k. Finally, the rule for $\texttt{ifz}(e; e_0; x.e_1)$ supplies the translation $\hat{e}$ with the continuation $\lambda y.\ \texttt{ifz}(y; \hat{e}_0 k; x.\hat{e}_1 k)$, which uses $y : \texttt{nat}$ by checking if it is zero and continuing with either $\hat{e}_0 k$ or $\hat{e}_1 k$ as appropriate.

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.*

The translation becomes more interesting at the function type.

$$\frac{\Gamma, x : \tau_2 \vdash e : \tau_1 \rightsquigarrow \hat{e}}{\Gamma \vdash (\lambda x{:}\tau_2.e) : \tau_2 \to \tau_1 \rightsquigarrow \lambda k.\ k(\lambda x{:}||\tau_2||.\hat{e})}\ (\textsc{Tr-Lam})$$

Recall that $||\tau_2 \to \tau_1|| \triangleq ||\tau_2|| \to |\tau_1|$. If our translation has the right type behavior, then $||\Gamma||, x : ||\tau_2|| \vdash \hat{e} : |\tau_1|$. Thus, $\lambda x.\ \hat{e}$ has type $||\tau_2 \to \tau_1||$, so we can supply it to the continuation k :

$||\tau_2 \to \tau_1|| \to \rho$. The key here is that hypotheses are translated with $||-||$ and conclusions with $|-|$, so it is natural to translate functions, which internalize a hypothetical, with $||\tau_2 \to \tau_1|| \triangleq ||\tau_2|| \to |\tau_1|$.

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* $||\tau_1||$*.*

Finally, we come to continuations.

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

This is enough to cover any program we would write in **K**, but there is a form of expression arising in execution which we have not handled – the term $\texttt{cont}(k)$ where k is a stack. In order to handle this case (and in order to state a correctness of translation result with respect to evaluation), we would have to define a translation of stack frames. We will not pursue this here.