Recitation 12:
Principles of Imperative Computation

15-312: Foundations of Programming Languages

Charles Yuan, Serena Wang

April 10th, 2018

1 Effects

So far in this course, we have been discussed purely functional programming languages with very limited notions of effects. In the first half of the course we built successively more expressive languages that were all total before arriving at **PCF**, the first language where *divergence* became a possibility. We then introduced *exceptions*, a feature that allows programs to "evaluate" to an exceptional state rather than returning a value. Recently we discussed **MPPCF**, a language in which the cost of evaluation is of great theoretical interest.

All of these concepts: divergence, unhandled expressions, and cost, are *effects* of a program that occur independently of the value to which a program evaluates (if there is one). But arguably the most well-known effect is *mutation.* Mutation allows memory cells to take on different values throughout the execution of a program, and for the evaluation of expressions to depend on a context of memory values. Mutation enables imperative programming, a style of programming that leverages updates to memory cells to encode the data being manipulated, to define memory regions accessed by references that may be shared, and to exert nonlocal changes to the state of the program.

Imperative programming enables many useful programming tools: easy constant-space iterative execution, possibly more efficient algorithms, and circular data structures to name a few. Many of these advantages are also conferred by sufficient optimizations to pure languages such as tail recursion optimization and lazy evaluation under certain circumstances. However, the prevalence and historical dominance of imperative programming means that we want to study it from a typed semantic perspective. The model we use to study imperative languages is **MA**, a language that looks foreign at first glance but essentially binds together defining characteristics of imperative languages ranging from C to Python.

2 Modernized Algol

Modernized Algol, or **MA**, builds on our familiarity with the modal distinction from **MPPCF**. In **MPPCF** we had the notion of expressions and values, which are made distinct because they have different meanings for cost effects. Values are pure, meaning that they incur no cost, while expressions are impure, meaning that they may incur cost to evaluate. Of course, we may treat values as zero-cost expressions using the `ret` construct, but there is no obvious other direction to treat expressions as values, overriding their costs.

In **MA**, we separate the *expressions* and the *commands*. Expressions are pure, meaning they do not mutate shared state, while commands are impure, meaning they may perform mutation. We may treat expressions as trivial commands that simply return that expression, and interestingly we may also treat commands as expressions by encapsulating them. The action of encapsulation corresponds to the lazy suspension from **MPPCF**; only when the command is executed are the effects observed.

The syntax for **MA** is derived from **PCF**, but with the addition of commands at the type and term level:

$$\begin{array}{llcll}
\mathsf{Typ} & \tau & ::= & \texttt{cmd} & \text{command} \\
\mathsf{Exp} & e & ::= & \texttt{cmd}\, m & \text{encapsulation} \\
\mathsf{Cmd} & m & ::= & \texttt{ret}\, e & \text{return} \\
 & & & \texttt{bnd}\, x \leftarrow e \,;\, m & \text{sequence} \\
 & & & \texttt{dcl}\, a := e \,\texttt{in}\, m & \text{declaration} \\
 & & & @\, a & \text{fetch} \\
 & & & a := e & \text{assign}
\end{array}$$

All of the **PCF** constructs remain, but we now also have the ability to define and run commands that declare variables and assign to them. The modal separation ensures that expressions do not directly cause the effects of variable state update, but commands when they are executed may do so.

2.1 Statics

The presentation above is for an untyped command type, which only has a meaningful formation judgment which asserts that uses of variables must be preceded by declarations of those variables. The expression typing judgment is augemented with the encapsulated commands having command type.

When commands are untyped, they are implicitly only allowed to return natural numbers.

$$\frac{\Gamma \vdash_\Sigma m \;\mathsf{ok}}{\Gamma \vdash_\Sigma \texttt{cmd}(m) : \texttt{cmd}}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{nat}}{\Gamma \vdash_\Sigma \texttt{ret}(e) \;\mathsf{ok}}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd} \quad \Gamma, x : \texttt{nat} \vdash_\Sigma m \;\mathsf{ok}}{\Gamma \vdash_\Sigma \texttt{bnd}(e; x.m) \;\mathsf{ok}}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{nat} \quad \Gamma \vdash_{\Sigma,a} m \;\mathsf{ok}}{\Gamma \vdash_\Sigma \texttt{dcl}(e; a.m) \;\mathsf{ok}}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a} \texttt{get}[a] \;\mathsf{ok}}$$

$$\frac{\Gamma \vdash_{\Sigma,a} e : \texttt{nat}}{\Gamma \vdash_{\Sigma,a} \texttt{set}[a](e) \;\mathsf{ok}}$$

As you can see, we augment the statics with a symbol context denoted as a subscript on the $\vdash$ symbol. Not only do we have variables in Γ, but also we have names of assignables in the context that are essentially labels of the assignables manipulated by commands.

2.2 Dynamics

The dynamics of **MA** are quite detailed and shown in section 34.1.2 of PFPL. For expressions, we rely on the value judgment $e\ \mathsf{val}_\Sigma$ and the step judgment $e \mapsto_\Sigma e'$. Most of these rules are exactly as in **PCF**, with the additional rule

$$\frac{}{\mathtt{cmd}(m)\ \mathsf{val}_\Sigma}$$

which says that encapsulated commands, being lazy, are values.

We then have judgments for dynamics of commands. Commands are either final, denoted $m||\mu\ \mathsf{final}_\Sigma$, or step, denoted $m\ ||\ \mu \quad \mapsto_\Sigma m'\ ||\ \mu'$. As before, keeping track of the active assignable names is very important, and done throughout the dynamics. Here we also explicitly track the memory of the command, μ, which maps assignable names to values.

3 Assignables

How does an assignable differ from a variable? Recall that variables are given meaning through substitution, and that substitution occurs in certain constructs during evaluation such as function application. We may then claim that variables are indeterminates of a certain type, which eventually are populated with either values (in a call-by-value interpretation), or by general expressions (in a call-by-name interpretation). Another perspective may be that variables are placeholders for any value/expression of a certain type, that a variable binding is a sort of genericism that defines some computation valid for any member of that type. Either way, variables represent potential future expressions; execution on a variable makes no sense.

Assignables are quite different, despite many mainstream programming languages using the term "variable" for both concepts. Instead of receiving meaning through substitution, assignables are essentially labels indexing into memory. Values may be obtained from an assignable by the fetch construct, and stored into it using the set construct. But the assignable itself is just a name, and the program executes without altering the identity of the assignable, only the memory that it refers to. It may as well be a string key into a table, or if we reify memory as a stack or heap-based storage, a pointer into main memory.

It is important to maintain the philosophical difference between variables and assignables in Modernized Algol.

4 Typed Assignables

So far, we have had a simplified representation of commands which are only allowed to return natural numbers and assignables that can only store numbers. We can generalize **MA** to add commands and assignables of arbitrary data types by altering the command type to become $\tau\ \mathtt{cmd}$, and changing the command formation judgment to a command typing judgment $m \mathrel{\dot\sim} \tau$.

This process is largely straightforward, and works for most types like numbers, products, and sums. However, we run into one important issue when we attempt to make commands or assignables of command or function type.

The issue is that of *scope*. In **MA**, assignables do not float within a global space. They are bound to specific memories μ, and in particular declarations expire at the end of a block. If we define a command within a block and return it or assign it into an assignable, then any active assignables inside that command would escape their original declaration scope, which does not make sense.

There are possibly solutions to this problem, such as choosing to allocate assignables within some global region or making the scoping of our language be more dynamic. However, we will enforce a standard stack discipline within our language: at the end of a block, the stack is considered to be unwound and any expired assignables undefined. This means that an assignable is only well-defined within the block it was declared in. Since any expression of a command type can involve an assignable, we thus regard the returning or assignment of a command or any type containing a command type illegal.

That leaves function types. When faced with a function type like $\texttt{nat} \to \texttt{nat}$, it may seem as if such a type is safe and cannot leak references. However, consider this expression:

$$\lambda(x : \texttt{nat})(\lambda(\_ : \tau\,\texttt{cmd})\texttt{z})(\texttt{cmd}(\texttt{get}[a]))$$

This expression has type $\texttt{nat} \to \texttt{nat}$, but it secretly references an assignable a. The type of a function cannot guarantee that it protects our assignables from leaking.

Therefore, in our extension to typed assignables, we cannot generalize to all types, but rather only to so-called mobile types, which are exactly the types except the bad cases above.

5 Capabilities and References

We need a to be in scope for us to ever get to use $\texttt{get}[a]$ and $\texttt{set}[a](e)$. However, what if we want to write procedures that operate on assignables? We would then need a way to take in a way to get and set the contents of the assignable provided as the argument of the procedure. Specifically, we need a capability, which is a pair of a getter and a setter for an assignable. Thus, we have the following type for a capability for an assignable a containing a value of type τ:

$$\tau\,\texttt{cap} \triangleq \tau\,\texttt{cmd} \times \tau \rightharpoonup \tau\,\texttt{cmd}$$

A capability for getting and setting an assignmable a containing a value of type τ may then look like

$$\langle \texttt{cmd}(\texttt{get}[a]), \texttt{proc}\,(x : \tau)\, a := x \rangle$$

However, with such a capability, we cannot be sure whether the getter and setter in the tuple actually refer to the same assignable. The type system does not give us such a guarantee, so we need to introduce the concept of references. A reference is a value from which we can get a capability for an assignable. The syntax for dealing with expressions of type $\tau\,\texttt{ref}$, the type of references to assignables of type τ, is shown below.

Typ	τ	::=	$\tau\,\texttt{ref}$	assignable
Exp	e	::=	$\&a$	reference
Cmd	m	::=	$*\,e$	getting
			$e_1 *= e_2$	setting

We can use references like so to increment the value stored in an assignable:

$$\texttt{proc}\,(r : \texttt{nat}\,\texttt{ref})\,\texttt{bnd}\,x \leftarrow *\,r\,;\,\texttt{bnd}\,\_ \leftarrow r *\!= x + 1\,;\,\texttt{ret}\,x$$

Since the procedure takes in a reference, we know that the getter and setter for that reference must refer to the same assignable.

References are also compatible with our stack discipline for scoped assignables. Since references give us capabilities and capabilities give us getters and setters, references cannot have mobile types. Thus, since the reference type is immobile and we only allow values of mobile types to be stored in assignables or returned from commands, we will still maintain our safety property of not leaking assignables.

6 Benign Effects

MA enforces a rigid separation between effectful and effect-free components of programs in the commands and expressions respectively. This is analogous to the design of Haskell, a pure functional language with constructs for monadic composition that represent the manipulation of state. The claim that Haskell is pure rests on the fact that commands are not evaluated for effects except at the top level, and commands (as well as other expressions) are treated lazily in Haskell and so cannot cause any effects when manipulated. The Haskell type system enforces that even constructs designed to produce side effects, such as I/O, may only take place within special command-like regions known as monads.

However, Standard ML does not make such a distinction. Standard ML has the concept of reference cells that may be accessed and assigned at the expression level, meaning that evaluation of SML expressions may cause side effects. This can be analyzed as a system of free assignables, detailed in section 35.5 of PFPL. When such a system is used, we may take advantage of benign effects. Benign effects are everywhere in SML, ranging from random number generator state updates to self-adjusting balanced search trees. Such programs use mutation and effectful code to provide useful features and performance characteristics to the programmer, without violating certain assumptions. Of course, it falls on the programmer to ensure that the effects are truly benign, as we lose the protection of the modal separation.

7 Backpatching

A particularly clever use of effects in a language with benign effects like SML is to implement recursion through backpatching. It turns out that with free assignables, we do not need fixpoints or recursive types to encode general recursive functions. We may use a trick where we assign to a function reference and access that reference within itself, just like a circular data structure.

Consider the following SML program, which demonstrates how to compute a factorial without any direct recursive constructs:

```
let
  val a = ref (fn (n : int) => n)
in
  (a := (fn (n : int) => if n = 0 then 1 else n * (!a)(n-1)); !a)
end
```

All we need to do is allocate a ref cell that is updated to become the definition of factorial, and refers to itself to obtain the definition of the self-referential call. This demonstrates that adding mutable state to a language has a significant impact on its expressiveness. Even without fixed points or recursive types, we have obtained general recursion.