Recitation 9: Control Flow and Exceptions

15-312: Foundations of Programming Languages

Charles Yuan, Jeanne Luning Prak

March 21, 2018

1 Control Flow

So far in this course, we have been using *structural dynamics* to study the evaluation of terms in a language. After an expression is deemed well-formed by the statics, the transition rules in a dynamics system tell us what happens at every step of evaluation, and the valuation rules tell us when evaluation ceases.

We've been able to encode some notion of control flow into this system: if-then-else clauses, function calls and recursive functions, as well as laziness in evaluation. But all of these are *local* forms of control, solely determined by the two sides of the $\longmapsto$ judgment. Advanced languages have sophisticated methods of *nonlocal control flow*, among which are the ideas of *exceptional control flow*, *continuations*, nonlocal jumps (`setjmp`, `longjmp`), etc. Such constructs must be modeled with a more sophisticated consideration of control.

The first half of this course largely focuses on logical fundamentals, safety, semantics, and expressiveness of the languages we develop. In the second half, we begin to examine interesting characteristics of languages like scope, mutable assignables, and call stacks. Control stacks in particular allow us to model nonlocal control, and the logical system that corresponds with our understanding of call stacks is called a **K** machine.

1.1 K Machines

A **K** machine is an **abstract machine** that lets us model computation. (A Turing machine is also an abstract machine of sorts.) It's not a language with syntax like we have studied so far, but has the same computational capacity as **PCF**.

A **K** machine possesses an expression language, the same as **PCF**. We also introduce the notion of stack frames f, which are defined inductively and look like this:

$$\mathtt{s}(-)$$

$$\mathtt{ifz}\ e_0\ \{\mathtt{z} \hookrightarrow x \mid \mathtt{s}(e_1) \hookrightarrow -\}$$

$$e_1(-)$$

Intuitively, we use a placeholder $-$ for one spot within the expression where we would normally expect a subexpression. When the frame is placed on the call stack, we mean that the stack is now trying to compute a value that should be placed at the $-$.

After we define the stack frames f, we can define the stacks k:

$$k ::= \epsilon \mid k; f$$

As you can see, stacks themselves are just lists of stack frames. ϵ is the empty stack.

The semantics of **K** machines are defined by the following judgments:

$k \triangleright e$ means that we are evaluating e on the stack k, and
$k \triangleleft v$ means that we are returning the value v from the stack k.

There is also a transition judgment $\longmapsto$, which relates the previous two judgments. We use the transitions to model **PCF** steps, so we want the following rules:

$$\frac{}{k \triangleright \mathtt{z} \longmapsto k \triangleleft \mathtt{z}}$$

This rule says "To evaluate zero on a stack, we just return it." Note that this is true since zero is a value in **PCF**.

$$\frac{}{k \triangleright \mathtt{s}(e) \longmapsto k; \mathtt{s}(-) \triangleright e}$$

This rule says "To evaluate the successor of e, we add a stack frame for the successor and evaluate e". Eventually e becomes fully evaluated, and we have

$$\frac{}{k; \mathtt{s}(-) \triangleleft e \longmapsto k \triangleleft \mathtt{s}(e)}$$

which says "Once e is a value, if it was waited on by a successor, we return the successor of e". Note the direction of the arrows; a right arrow always means "evaluate more", and a left arrow always means "return a value". We always keep k around as the remaining stack. It's sort of like a context for the dynamics, full of the computations that are waiting on this evaluation.

There are more of these rules, one for each transition in **PCF**, and they are given in the textbook.

A valid **K** machine is an explicit representation of the control flow of a **PCF** program. As an exercise, try to evaluate $(\lambda\,(x : \mathtt{nat})\,\mathtt{s}(x))(\mathtt{s}(\mathtt{z}))$ on a **K** machine. You need only to write out all the states that take

$$\epsilon \triangleright (\lambda\,(x : \mathtt{nat})\,\mathtt{s}(x))(\mathtt{s}(\mathtt{z}))$$

to

$$\epsilon \triangleleft \mathtt{s}(\mathtt{s}(\mathtt{z})).$$

1.1.1 Safety

K machines have a notion of correctness which should be shown to make the system fully logically sound and complete. While we won't focus on it, the crux is to define and prove the property that **K** machines are well-formed in that they accept values of certain type, then transform them into values of some other type. Once we have this, we may prove that **K** machines yield exactly the same behavior as **PCF** structural dynamics. The treatment is fully fleshed-out in the textbook.

2 Exceptions

So far, we have devised an alternative scheme to **PCF** structural dynamics which is guaranteed to give the same results as the original system. That's not very interesting, but now we can extend **K** machines to do what **PCF** could not: handle **exceptions** with nonlocal handlers. This will be the first example of how the stack machine extends our understanding of evaluation, and we will soon see more.

2.1 Exceptions vs. Errors

Our treatment of exceptions is modeled off the behavior of exceptions in ML. Exceptions are one of the frequently controversial aspects of high-level programming languages, and much debate occurs over whether they should be used,[1] what information they should carry, how they should be checked statically,[2] the importance of optimizing exceptional control flow,[3] and even whether they're necessary at all![4]

In a functional setting, exceptional control flow circumvents the type system. If a type is a theorem and a function is an implication according to Curry-Howard, then the presence of exceptions undermines the notion of proof since `raise` can be used to prove any claim. It also makes evaluation "dangerous" in that well-typed expressions may now not only evaluate to a value, or diverge, but also now raise an exception. Why, then, do we still use exceptions?

It turns out that ML's exception features are highly nuanced largely thanks to the `exn` type.[5] Exceptions in ML serve many more purposes than meets the eye:

- To indicate that an error has occurred. A generic "failure" would also suffice to show this, as we saw with dynamically checked languages, or perhaps a sum (option) type.
- To deliberately invoke nonlocal control flow, such as in a backtracking algorithm. This pattern is much more common in ML than in e.g. Java. Regular control flow could of course be written to achieve the same effect, but might be convoluted by comparison. Continuations, which we will see soon, also serve this purpose.
- To *share data* with the handler of the exception according to a set of programmer- defined tags.

The last point distinguishes ML-style exceptions from other languages, and merits some more discussion. For now we will appeal to our understanding of the `exn` type from ML to see how an exception is able to transmit a packet of data to a handler, who can only unwrap the data if they understand the exception. This is a rudimentary form of secrecy in data transmission! Consider the following snippet:

```
structure Alice :> sig
  exception Message of string
  val tell : string -> unit
end =
struct
  exception Message of string
```

[1]The Google C++ style guide prohibits using C++ exceptions altogether! Seems like a bit of a waste...
[2]Some believe Java's notorious checked exceptions to be a mistake.
[3]See benchmarks. One "pro" is that explicit checks are not necessary at every stack frame. One "con" is that stack unwinding is slow.
[4]Like all things, exceptions are monadic and therefore perfectly expressible within the Haskell type system...
[5]Largely known as "exception" or "extensible" depending on who you ask.

```
  fun tell s = raise Message "secret"
end
```

Now we may communicate with `Alice` by sending using `tell` and receiving by handling `Alice.Message`:

```
fun tell_and_listen s = (Alice.tell s; NONE) handle Alice.Message s => SOME s
val SOME "secret" = tell_and_listen "hi"
```

Now consider sealing the exception away:

```
signature SHH = sig
  val tell : string -> unit
end
structure SecretAlice = Alice :> SHH
```

Now we can still send messages with `tell`, but there is no way to receive a message, because the exception is completely opaque to the sender. A handler could try to catch the exception, but has no way to unwrap the internals since there is no exception `Message` visible to the outside. Only someone who had access to the exception `Message` could read `Alice`'s message, in a typesafe system!

We will discuss `exn` in much more detail when we discuss *dynamic classification* later in the course. For now, this is a taste of how interesting ML exceptions really are, with a control component that is already more ergonomic than other languages, and a data component that is innovative. Despite some drawbacks, using exceptions judiciously enables concise and nuanced code. Few other inventions could suffice to serve all the purposes of exceptions.

2.2 Defining Exceptions

Now that we have motivated exceptions, we can define the syntax and semantics for exceptions. To add exceptions to a language (in our case, we will be adding them to PCF), we add two new operators:

$$\begin{array}{rcl} e & ::= & \ldots \\ & & \texttt{raise}(e) \\ & & \texttt{try}(e_1; x.e_2) \end{array}$$

The $\texttt{try}(e_1; x.e_2)$ creates an exception handler, which either steps to e_1 if no exception is raised, or it binds the exception value to x and evaluates e_2 if an exception is raised. The $\texttt{raise}(e)$ raises an exception with value e.

The statics for `try` and `raise` are given below:

$$\frac{\Gamma \vdash e : \tau_{\texttt{exn}}}{\Gamma \vdash \texttt{raise}(e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{\texttt{exn}} \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}(e_1; x.e_2) : \tau}$$

Note that we have not yet defined the $\tau_{\texttt{exn}}$ type: the type of exception values. There are many types that we could choose to be $\tau_{\texttt{exn}}$, including:

- Strings, to contain error messages.
- Ints, to contain error codes.
- A sum of many different types, to represent a variety of possible errors.

However, all of these have the shortcoming that they constrain us to storing one particular type (or a fixed number of types) of information in an exception. However, there are many different conditions under which we might want to raise an exception, and in each of them, we might want to associate a different type of data with the exception. We will see a way of dealing with this problem later in the class, so for now we won't define exactly what $\tau_{\texttt{exn}}$ is.

To define the dynamics, we add a third judgment to the **K** machine semantics:

$k \blacktriangleleft e$ means that we are passing the exception value e to stack k

We then add judgments for `try` and `raise`:

$$\frac{}{k \triangleright \texttt{raise}(e) \longmapsto k; \texttt{raise}(-) \triangleright e}$$

$$\frac{}{k; \texttt{raise}(e) \triangleleft e \longmapsto k \blacktriangleleft e}$$

$$\frac{}{k \triangleright \texttt{try}(e_1; x.e_2) \longmapsto k; \texttt{try}(-; x.e_2) \triangleright e_1}$$

$$\frac{}{k; \texttt{try}(-; x.e_2) \triangleleft e \longmapsto k \triangleleft e}$$

$$\frac{}{k; \texttt{try}(-; x.e_2) \blacktriangleleft e \longmapsto k \triangleright [e/x]e_2}$$

$$\frac{(f \neq \texttt{try}(-; x.e_2))}{k; f \blacktriangleleft e \longmapsto k \blacktriangleleft e}$$

The last rule states that a raised exception is propagated up the stack, until it reaches a handler or an empty stack. In the second to last and last rules, whether we see a $\blacktriangleleft$ or $\triangleleft$ judgment allows us to determine whether to go into the e_1 or the e_2 of the `try`.