Continuations (= Stacks)

Stacks record what we should continue doing after we're done evaluating the current expression. So, they're sometimes referred to as a "continuation".

If we have a stack in the language, then we might consider making it a first-class thing that the programmer can manipulate, just like functions or integers. In other words, we might add stacks to the class of values:

$$v ::= \ldots \mid S$$

In addition, we might add mechanisms for getting our current stack/continuation (letcc) and for installing a given stack as the current continuation (throwcc):

$$e ::= \ldots \mid \mathsf{letcc}\ x\ \mathsf{in}\ e \mid \mathsf{throwcc}\ e_1\ e_2$$

For throwcc, we'll also need two new stack frames to allow us to evaluate the nested subexpressions:

$$F ::= \ldots \mid \mathsf{throwcc}\ [\,]\ e \mid \mathsf{throwcc}\ v\ [\,]$$

The rewriting rules for these two new mechanisms are:

$$(S, \mathsf{letcc}\ x\ \mathsf{in}\ e) \rightarrow (S, e[S/x])$$

$$(S, \mathsf{throwcc}\ S'\ v) \rightarrow (S', v)$$

$$(S, \mathsf{throwcc}\ v\ e) \rightarrow ((\mathsf{throwcc}\ v\ [\,]) :: S, e) \qquad (e \text{ not a value})$$

$$(S, \mathsf{throwcc}\ e_1\ e_2) \rightarrow ((\mathsf{throwcc}\ [\,]\ e_2) :: S, e_1) \qquad (e_1 \text{ not a value})$$

Notice that letcc makes a copy of the current stack and substitutes it for x within the body of the letcc. The throwcc operation throws away the current stack and installs its first argument as the current continuation/stack.

If we have letcc and throwcc, then we can (more or less) code up exceptions: try/catch corresponds to doing a letcc, and throw corresponds to doing a throwcc to that continuation.

Homework

Due Wed, 26 Nov before class.

1. Implement an interpreter for a language with continuations. You should use the following datatype definitions:

```
type var = string

datatype value = Int of int | Fn of var * exp | Stack of stack

datatype opn = Plus | Times | Minus

and exp = Var of var | Val of Value | Opn of exp * opn * exp |
   App of exp * exp | Letcc of var * exp | Throw of exp * exp

and frame = ...  (* to be filled in by you *)

withtype stack = frame list
```

   Note that you'll need to fill in appropriate constructors for the definition of the frame datatype. You should write two functions for your interpreter:

```
val step : stack * exp -> stack * exp

val evaluate : exp -> value
```

   evaluate will call step until a terminal configuration is achieved.

2. SML/NJ doesn't provide letcc and throwcc, but it does provide two closely related constructs in the SMLofNJ.Cont library:

```
callcc : ('a cont -> 'a) -> 'a
throw : 'a cont -> 'a -> 'b
```

   A "τ cont" is a continuation/stack that expects to be thrown a τ value. You can encode "letcc x in e" as "callcc (fn x => e)".

   Suppose you have the following datatype:

```
datatype 'a tree = Leaf of 'a | Node of {left:'a tree,right:'a tree}
```

   Your goal is to write a function:

```
val exists : ('a -> bool) -> 'a tree -> bool
```

with the propery that (exists p t) should return true iff t has a Leaf(v) such that $p(v)$ returns true.

Furthermore, and this is the interesting part, assuming that throw is a constant time operation, your function should return its answer within a constant number of steps of finding the first Leaf value that satisfies the given predicate. Your function should require no auxiliary data structures, nor should it require multiple passes over the tree.

Note that a standard recursive procedure does not have this property, as you'll end up "unwinding" the stack when you have a deeply nested Leaf that satisfies the predicate.

Extra Credit: Write exists2 so that it has the same type, does not use exceptions, callcc, or an auxilliary data structure, but performs the same task in the same number of steps.

3. Suppose we wish to model a multi-threaded programming language such as Java. In particular, suppose we have a new expression form "fork e_1 e_2" which starts a new thread running the function e_1 applied to the argument e_2. That is, suppose we have the following expressions:

$$e ::= x \mid i \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{fork}\ e_1\ e_2$$

Give a small-step, call-by-value operational semantics for this language. This means definining configurations and transition rules for the expression forms.