CPS Semantics: Non-local Control

13 Nov 2002

Scribe: Ben Viglietta and Mehmet Fidanboylu

Lecturer: Radu Rugina

1 Previously...

With Continuation Passing Style, we had the following kind of semantics:

$$\mathcal{C}[\![e]\!]\rho\kappa$$

This evaluates e, then passes result value to the continuation κ, which models the rest of the computation. Here ρ can be thought as the naming environment and κ as the control environment.

Just to remind you here is the CPS semantics for addition:

$$\begin{aligned}\mathcal{C}[\![e_0 + e_1]\!]\rho\kappa = \quad & \mathcal{C}[\![e_0]\!]\rho\ (\mathit{check\_int}\,(\lambda n_0.\\ & \mathcal{C}[\![e_1]\!]\rho\ (\mathit{check\_int}\,(\lambda n_1.\kappa(n_0 + n_1))))\end{aligned}$$

This function evaluates e_0 and e_1, then passes the result of their addition (since we have the addition operator) to the continuation. The higher order function function *check_int* is a filter that only allows integer values to pass through, returning error otherwise. For a precise definition of this function, please refer to the previous lecture.

2 References

Recall that for CPS semantics we had the denotation defined as:

$$\mathcal{C}[\![e]\!] : \mathit{Env} \to \mathit{Cont} \to \mathit{Result}$$

and for the F_{ref} language we had:

$$\mathcal{C}[\![e]\!] : \mathit{Env} \to \mathit{State} \to \mathit{Result}$$

To add references with continuation style passing, we need to combine these into:

$$\mathcal{C}[\![e]\!] : \mathit{Env} \to \mathit{State} \to \mathit{Cont} \to \mathit{Result}$$

Since the states will be passed around, we need to extend our *Result* domain with states:

$$\mathit{Result} = ((\mathit{Value} \times \mathit{State}) + \mathit{Error})_\perp$$

The new CPS denotation will be of the form: $\mathcal{C}[\![e]\!]\rho\sigma\kappa$.

Now let's see how we would define the semantics of references using this denotation. Before we do that, we would like to modify the *check_X* functions to be able to check pairs in the form *(X, state)*. We need this modification because the *Result* domain consists now of pairs of values and states. We will call these new check functions, *check_X'*.

- *Reference expressions:*

  $\mathcal{C}[\![\mathsf{ref}\ e]\!]\rho\sigma\kappa = \quad \mathcal{C}[\![e]\!]\rho\sigma\ (\mathit{check\_val}'(\lambda v, \sigma'.\ \text{let}\ l = \mathit{alloc}(\sigma)\ \text{in}\ \kappa(l, \sigma'[l \to v])))$

  Evaluate e, then allocate a location not already allocated in σ,
  then call the continuation on this location and the new state where l is assigned v.

- *Dereferencing:*

  $\mathcal{C}[\![!\,e]\!]\rho\sigma\kappa = \quad \mathcal{C}[\![e]\!]\rho\sigma\ (\mathit{check\_loc}'(\lambda l, \sigma'.\kappa(\sigma'(l), \sigma')))$

  Evaluate e, check if it is a location and state pair,
  then call the continuation on the value of the location and the new state.

- *Assignment:*

$$\mathcal{C}[\![e_0 := e_1]\!]\rho\sigma\kappa = \mathcal{C}[\![e_0]\!]\rho\sigma\ (check\_loc'(\lambda l, \sigma'.\ \mathcal{C}[\![e_1]\!]\rho\sigma'\ (check\_val'(\lambda v, \sigma''.\kappa(unit, \sigma''[l \to v])))))$$

Evaluate e_0, check if it is a location and state pair,
then evaluate e_1 check if that returns a value and state pair,
then call the continuation on unit and the new state with l is assigned v.

- *Sequencing:*

$$\mathcal{C}[\![e_0; e_1]\!]\rho\sigma\kappa = \mathcal{C}[\![e_0]\!]\rho\sigma\ (check\_val'(\lambda v, \sigma'.\ \mathcal{C}[\![e_1]\!]\rho\sigma'\ (check\_val'(\lambda v', \sigma''.\kappa(unit, \sigma'')))))$$

Evaluate e_0, check if it is a value and state pair,
then evaluate e_1 check if it evaluates another value and state pair,
then call the continuation on the unit value and the new state.

Note: we could alternatively defined the semantics of assignment (or sequencing) to be the value of the assigned expression (or the value of the last command executed in the sequence) instead of unit.

2.1 CPS as a Programming Style

One can actually write programs with explicit continuations passed as arguments to procedures. Here's an example ML program which computes the sum of the elements in a list using continuations.

```
fun sum l =
  let
    fun  sum' [] k = k 0
       | sum' (x::t) k = sum' t (fn y => k(x+y))
  in
    sum' l (fn x => x)
  end
```

The helper function `sum'` takes its continuation `k` as an explicit argument. If the list is empty it just applies the continuation to 0; if not, it calls `sum'` recursively on the tail of the list while passing the addition of the head element as a continuation. Finally, for the evaluation of the whole program, we use the halting continuation `fn x => x`.

3 Constructs with Unstructured Control Flow

We next discuss various language constructs with unstructured control flow. Continuation passing style makes it easier to express the semantics of such constructs.

3.1 Label and Goto

First we extend our language with C-like label and goto statements:

$$e ::= \ldots \mid \textsf{label}\ x\ e_v \mid \textsf{goto}\ x\ e_v$$

Informally:

label x e_v : evaluates e_v to a value v
stores the current continuation in x (i.e. we capture the current control flow point in x)
evaluation of the whole expression yields value v

goto x e_v : evaluates e_v to a value v
passes the control and the value to the continuation stored in x.

Since label expressions assign continuations to locations, we need to extend the *Value* domain to include continuations as well:

$$Value = \ldots + Cont$$

The CPS semantics (denotations) for the label and goto expressions are as follows:

$\mathcal{C}[\![\mathsf{label}\ x\ e_v]\!]\rho\sigma\kappa = \quad \mathcal{C}[\![e_v]\!]\rho[x \mapsto \kappa]\sigma\kappa$

Map x to the current continuation κ and return the denotation of e_v

$\mathcal{C}[\![\mathsf{goto}\ x\ e_v]\!]\rho\sigma\kappa = \quad \mathcal{C}[\![e_v]\!]\rho\sigma\rho(x)$

This is actually not completely correct since we have to check if the value of x is actually a continuation. Either of the rules below performs this checks. They differ in the order they evaluate x and e_v.

$\mathcal{C}[\![\mathsf{goto}\ x\ e_v]\!]\rho\sigma\kappa = \quad \mathcal{C}[\![x]\!]\rho\sigma(check\_cont'(\lambda\kappa', \sigma'\ .\ \mathcal{C}[\![e_v]\!]\rho\sigma'\kappa'))$

$\mathcal{C}[\![\mathsf{goto}\ x\ e_v]\!]\rho\sigma\kappa = \quad \mathcal{C}[\![e_v]\!]\rho\sigma(check\_val'(\lambda v, \sigma'.$
$\mathcal{C}[\![x]\!]\rho\sigma'(check\_cont'(\lambda\kappa'.\kappa' v))))$

3.2 Setjmp and longjmp

Extend the syntax of F_{ref} with C-like setjmp/longjmp constructs:

$$e \quad ::= \quad \cdots \quad |\ \mathsf{setjmp}\ e_l\ \ |\ \mathsf{longjmp}\ e_l$$

For both constructs, e_l must evaluate to a location l. Then setjmp stores the current continuation in l; and longjmp jumps to the continuation stored in l. To distinguish between the two possible ways of "returning" from a setjmp, we will define the semantics such that setjmp returns false if the control reaches this statement through the evaluation of a longjmp and returns true otherwise (i.e. control normally reaches setjmp).

Here is an example C program that uses setjmp/longjmp to simulate exceptions. In contrast to the goto statement in C, setjmp/longjmp are able to transfer the control across procedures. This requires to save the program stack when setjmp is invoked, and restore the stack when longjmp is invoked. The structure `jmp_buf` is used to save the full program stack. In terms of CPS semantics, the `jmp_buf` holds a continuation.

```
jmp_buf buf;

void foo() {
  if (setjmp(buf) == 0)
    compute();
  else
    handle_error();
}
```

```
void compute() {
  ...
  if (n == 0)
    longjmp(buf);
  else
    x = 1/n;
  ...
}
```

In the event of an error (such as an attempted divide-by-zero), longjmp returns control to the location of the call to setjmp, similar to how an exception would return the control to a point higher up the call stack. However, unlike exceptions, setjmp may pass the control to a function which is not currently on the stack.

Here is another example of how setjmp/longjmp can be used to simulate a looping construct in F_{ref} :

```
let k = ref 0 in
  let i = ref 0 in
    let dummy = setjmp k in
      if (!i < 10)
      then (i := !i + 1; longjmp k)
      else !i
```

The formal CPS semantics of setjmp/longjmp can be defined as follows:

$$\begin{aligned}
\mathcal{C}[\![\textsf{setjmp}\ e_l]\!]\rho\sigma\kappa &= \mathcal{C}[\![e_l]\!]\rho\sigma(\mathit{check_loc}'(\lambda\, l, \sigma'.\ \kappa(\textsf{false}, \sigma'[l \rightarrow \kappa]))) \\
\mathcal{C}[\![\textsf{longjmp}\ e_l]\!]\rho\sigma\kappa &= \mathcal{C}[\![e_l]\!]\rho\sigma(\mathit{check_loc}'(\lambda\, l, \sigma'.\ \mathit{kcase}\ \sigma'(l)\ \text{of}\ \mathit{Cont}(\kappa') \rightarrow \kappa'(\textsf{true}, \sigma')))
\end{aligned}$$

3.3 Callcc

Extend the syntax of F with ML-like callcc ("call with current continuation"):

$$e \ ::= \ \cdots \ \mid \ \textsf{callcc}\ e_f \ \mid \ \textsf{throw}\ e_\kappa\ e_v$$

Informally:

- The expression $\textsf{callcc}\ e_f$ evaluates e_f to a function f which takes a continuation as an argument. The evaluation of callcc then invokes e_f with the current continuation as a parameter.
- The expression $\textsf{throw}\ e_\kappa\ e_v$ evaluates e_κ to a continuation κ, e_v to a value v, then calls κ with parameter v.

The formal semantics of these constructs are defined as follows (assuming for simplicity that we are working with the language F and not F_{ref}):

$$\begin{aligned}
\mathcal{C}[\![\textsf{callcc}\ e_f]\!]\rho\kappa &= \mathcal{C}[\![e_f]\!]\rho(\mathit{check_fun}\,(\lambda\, f.\ f\ \kappa\ \kappa)) \\[1em]
\mathcal{C}[\![\textsf{throw}\ e_\kappa\ e_v]\!]\rho\kappa &= \mathcal{C}[\![e_\kappa]\!]\rho(\mathit{check_cont}\,(\lambda\, \kappa'. \\
&\qquad \mathcal{C}[\![e_v]\!]\rho(\mathit{check_val}\,(\lambda\, v.\ \kappa'\ v))))
\end{aligned}$$