CSE 341: Programming Languages

Dan Grossman

Spring 2004

Lecture 16— Continuations and Related Idioms

Today

- What are continuations?
- What does `let/cc` mean?
- How do continuations provide exception-like behavior?
- Related idiom not using `let/cc`: iterators

Time permitting: How "time travel" makes continuations so powerful (and easy to misuse).

Programs with holes

Consider:

`(+ 3 (* 2 □))`

What does this "program with a hole" mean?

"If you put a value v in the hole, the result is two times v plus 3."

That sounds like a function where the "function body" is the "rest of computation".

A continuation is "the rest of computation".

A language with "first-class continuations" lets you "get at continuations". Most let you treat them as functions (with weird semantics).

The `let/cc` primitive

`(let/cc k e)`

- Bind `k` to the current continuation (the rest of computation), a "function" that given a value (for a hole), completes computation.
- Evaluate `e` to `v` and the result is `v`
- *But:* Calling the continuation (e.g., `(k 7)`) means "forget everything else, the rest of computation is now the continuation with 7 in the hole".

Examples:

```
(+ 1 (let/cc k (+ 2 (if x 3 (k 4)))))
((lambda (pr) (if (= (car pr) 0) 7 ((cdr pr) (cons 0 #f))))
 (let/cc k (cons 3 k)))
```

Connection with exceptions

Instead of building exceptions into our language, we can:

- Pass in a continuation (or store it in a mutable global if you must)
- Call the continuation to "forget what you are doing" and transfer control to an outer "rest of computation"

A lower-level view

Continuations really are defined in terms of "holes" and "rest of computation".

But it's often easier to reason in terms of a "call-stack" implementation.

In this view:

- `let/cc` wraps the current call-stack in a special function
- Calling a continuation replaces the now-current call-stack with the one at the time of `let/cc`

And that's where "time travel" comes in: you can switch to a call-stack that without continuations would have not been needed any more!

Killer app: user-level non-preemptive threads!

Some perspective

Continuations are perhaps *too* powerful and difficult to use well.

Non-advanced programmers stay away from them.

But it's nice to think more generally about:

- Languages with more powerful "control operators" than if and function application (languages used to not have exceptions either)
- Programming styles (idioms) that exploit the idea of "rest of computations"

Example of the latter: iterator over trees