2013W1-lecture28

November 22, 2013

Contents

1 Question of the Day

Let's think about continuations.

- What is the continuation of `objectAt(size - 1)` in the following code?
- What is the continuation of `throw new EmptyStackException();` in the following code? Careful, this is a "double-trick" question:
  - The continuation divides the runtime call tree into three parts:
    - The past is gone, and we ignore it.

    - The "present" is the whole subtree that evaluates the current expression (or statement, in this case), and we **ignore it** as well. In other words, it could be replaced by any other expression/statement, and it would make no difference to the continuation! (For an expression, we give a name to the value that will come back so we can use that value. For a statement, we're going to ignore any value, anyway.)
    - The "future" is the continuation.

- What continuation **actually occurs** when we evaluate `throw new EmptyStackException();`?

```
// Code from the Oracle Java tutorial on throwing exceptions.
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

2 Logistics

2.1 Final Project: Proposal resubmit due today

If you choose to resubmit (in which case be sure to submit a file with the appropriate name!), you'll be remarked and receive the new mark.

2.2 Final Project: Background Research Report due today

Be sure to AT LEAST easily clear the resubmit bar!

Better: submit something you think is good and get feedback on it from your facilitator!

2.3 Steve owes lambda calculus and continuation practice problems

3 Converting to "Continuation-Passing Style" (CPS)

How in the world does `send/suspend` manage to "grab" the continuation? Let's figure it out.

In order to get there, we need to expose the continuation. Right now, we just rely on Racket (or Java or C++ or . . .) to keep track of the continuation for us. Let's see if we can instead make the continuation **explicit**. In other words, we're going to write programs that automatically generate the `lambda` forms representing the continuation

4 Converting an interpreter to CPS

Converting our programs to CPS is extremely messy, but we don't need to! As long as our runtime environment has a representation for the continuation, it can expose that to us! That's what "stack ripping" does. We'll accomplish the same thing by converting an interpreter to continuation-passing style.

5 What have we learned today?

- Continuations
  - Justify the utility of "grabbing" everything that has to happen **after the current evaluation finishes** in a program?
    - For a callback: as in web programming.
  - Express the continuation of a particular program point as a function (to the extent that that's possible), an English description, and a portion of the runtime call tree.
- Continuation-Passing Style
  - Define tail call/tail position
    - A tail call is a function call (e.g., a call to `interp`) that represents "all work remaining to be done" in evaluating the current expression.
    - Evaluation of an expression in tail position within a syntactic form is always a tail call. (Note: sometimes a syntactic form

    has multiple of these, as in `if` expressions! Sometimes an expression will have **no** tail position, as in `display`.)

  - Define continuation-passing style
    * A program with the invariant that every (non-trivial, for some appropriate definition of non-trivial) computation is in tail position and where each non-trivial computation consumes a continuation: a function which, given the value of the current computation, performs all computations after the current computation.
  - Convert a simple program into continuation-passing style.

- Converting an interpreter to CPS
  - Given an implementation of a case in an interpreter, convert it to continuation-passing style.
  - Explain the advantage of having an interpreter implemented in continuation-passing style.
  - (If we get to it:) Implement a simple control-flow construct using an interpreter in CPS (possibly by desugaring to `let/cc` and equivalent operations).