Continuations: implementation issues

Amer Diwan

Continuations are powerful, but so what?

• Many “exotic” features of functional languages have found their way into mainstream languages
  – “weak” first class functions
  – garbage collection
  – separation of mutable and immutable variables
  – ?? objects
• Weak notions of continuations already exist:
  – setjmp/longjmp in C
  – exceptions
Rules of the game

• Assume there are no assignments
  – if there are, then variables modified since callcc are not restored on a throw
  – in functional languages, modifiable objects are usually put in the heap
    • all variables in the stack are immutable

Challenges in implementing continuations

• Continuations may escape:
  – fun yield() =
    if random()
    then ()
    else callcc(fn k => (enqueue k; dispatch()))
  – k is put on a queue and may be thrown even after yield and its callers return

• A continuation may be thrown multiple times
  – As long as there is a possibility that a continuation may be thrown, we must preserve all associated data
Not too different from challenges in implementing closures

```c
void client() {
    int i;
    f = return_fcn(10);
    t = f(5);
}
```

```c
return_fcn(int i) {
    int n;
    int incr(int elem) {
        return elem+n;
    }
    n = i;
    return incr;
}
```

Need to keep an activation record around

---

Implementation strategies: allocating activation records on the stack

Allocate activation records on the heap

Stack

- \[\text{Stack}\]

- \[\text{Heap}\]

- \[\text{callcc here}\]
Example (cont.)

Let’s say the continuations is stored away and the top two functions return

![Stack Diagram]

![Heap Diagram]

Example (cont.)

Now let’s suppose the continuation is thrown

![Stack Diagram]

![Heap Diagram]
Example (cont.)

Now the code can return and then make another call...

Example (cont.)

Now the continuation may be thrown again...
Analysis of example

• Trying to put activation record on the stack can involve a lot of copying whenever a continuation is captured/thrown
  – Could put all activation records on the heap (SML/NJ model)
  – Or can do the copying smartly (this paper)
• If each activation records is in its own heap object, then copying can be avoided at the cost of more expensive common case (calls and returns)

Solution presented in the paper

• Allocate stack on the heap, but
  – Allocate large chunks of data to be used as the stack
  – Within a chunk, things are almost as efficient as on a hardware supported stack
  – Copy chunks (or part of chunks) lazily when continuations are captured/thrown
Normal execution (i.e., continuations are not captured or thrown)

Discussion

- Stack segment
  - Within a stack segment, you pretend you have a normal stack: activation records are pushed and popped just like in a hardware stack
  - If you reach the end of a segment, need to create a new segment
  - What happens when you return from the first AR in a segment?
Returning from a segment

- In a “normal” return set the PC to return address
- In a segment return, the return address is a fake one (underflow): it is of a code that will use the “stack records” to adjust the pointers

Issues

- How big should a stack segment be?
  - Stack segment must be at least as large as the largest possible AR
  - If stack segments are small, then this strategy degenerates into putting each AR in its separate heap object
What happens when a continuation is captured?

What does underflow’ do?

• When the activation record containing it returns, it jumps to underflow’
  – underflow’ knows that the code under it is a captured continuation so it makes a copy of the continuation records in the current segment
  – activation records in a continuation are copied a few at a time in this way
What happens when a continuation is thrown?

Copy the top segment of the continuation, and set the PC to RA2

Some discussion

- The “underflow” trick allows them to do the copying lazily without adding additional checks
- If one knows that a continuation will only be thrown once, then don’t need to copy continuation segment on a throw
More discussion

• The underflow trick works well when one returns from a segment. But how about segment overflow?
• What did you like/dislike about this paper?

Discussion

• Continuations are powerful but are they needed in their full generality?
  – How about if they were limited never to escape?
  – How about if they could be thrown at most once?
  – ...?
Next topic: Garbage collection

• We will look at different garbage collection algorithms
• How they language features interact with garbage collection
• Some implementation strategies