Wound Tight $^\beta$

Lang Martin

2014-04-15

Late Binding

One way to think about progress in software is that a lot of it has been about finding ways to late-bind, then waging campaigns to convince manufacturers to build the ideas into hardware. Early hardware had wired programs and parameters; random access memory was a scheme to late-bind them. Looping and indexing used to be done by address modification in storiage; index registers were a way to late-bind.

—Alan Kay, "The Early History of Smalltalk"

The Call Stack (in JavaScript)

Representative of everything except Forth (and so forth) and Prolog and it's siblings (maybe?).

- Local variable bindings
- `return` address
- Exceptions

The mechanism that supports procedural programming.

Closures Capture Part of the Stack

Return a procedure from a local binding context, and it will keep (often by copying to some part of the heap) context from that inner environment.

Callbacks

JavaScript doesn't provide a mechanism for capturing the `return` address, so (in node, e.g.) we don't return from a procedure. We write our procedures in the form of callbacks.

Each line of code is encapsulated in it's own procedure which captures the local variable scope and expects an async parameter interface.

The Error Parameter

Since we've lost the built-in `try` and `catch`, our async libraries agree to pass the current exception as the first parameter and the return value as the second.

Let's Make One: The Goal

```
begin(
    errorHandler,
    [login,
     getCounters,
     first,
     inc,
     pushCounter],
    function(result) {
        console.log(result);
    });
```

The Evaluator

```
function begin (handler, expr, callback, res) {
    if (expr.length === 0) return callback(res);
    var fn = expr[0], rest = expr.slice(1),
        retry = function() {
            begin(handler, expr, callback, res);
        },
        cont = function(err, res) {
            if (err) handler(err, retry);
            else begin(handler, rest, callback, res);
        };
    try {
        res = fn(res, cont);
        if (res !== undefined) cont(null, res);
    } catch (err) { handler(err, retry) }
}
```

Return Address

- The return address is free, because we can expect procedures to adhere to the node interface and accept a callback
- Gambit Scheme and continuation passing style
- Chicken Scheme and copying the stack to the heap

Error Handling

The error handler is invoked for synchronous expressions by the try catch block, and for async procedures by the node protocol for handling the first callback argument.

The second argument to the error handler supports “retry”, the handler can prompt the user, say, and then invoke the retry thunk to resume the program in the same place.

Wiggles

- A stack trace contains outer context from the development environment that isn't part of your program
- Also, inner context that's from a different module of your program
- The solution: tagged, delimited continuations

New Things: Inversion of Control

- A common pattern, it's often the first step of optimization (that's why node does it!)
- The consumer decides when each line of library code runs
- Libraries present a lazy stream interface
- Enumeration equals iteration

Programming with the Debugger

- Batching requests
- Retry on error
- Progress reports
- Leaving control with the consumer keeps the application code legible.

Prior Art: REST

- POST to an endpoint generates an object with an id that represents the result of the computation
- State is externalized, and as a result a client can capture and replay state
- Version extensions help reason about the evolution of state

Monadic bind and return

- Bind inserts a new function to catch the return value
- Return passes its value to the next function in the current monad's stack

The "bigger" or "smaller" monad returned by these operations contains the queue of remaining operations. Some caller must `do` the monad in order to execute the planned call stack.

Functional Reactive Programming

- The stream of events externalizes the call stack
- Event listeners are functions
- Listeners are free to capture and manipulate the stack of procedures to execute