Many language features and tools need access to the control stack of a running program.

- steppers
- debuggers
- stack tracing
- exception handlers
- capturing continuations (call/cc) / delimited control
- dynamic binding / parameterize / anything that needs dynamic context
- determining call depth to properly indent pretty-printed output (small example from thesis)
- AOP (involves modifying code at run time)
- stack inspection

It's a bad idea to give full access to the control stack.

- breaks language abstractions - not safe
- also, revealing stack implementation prevents certain optimizations
- implementation that uses stack information is tied to one particular evaluator - not portable
- makes it difficult to formally specify language features or tools - need to include low level information

0.0.1 Continuation Marks

John Clements gives another solution in his dissertation, continuation marks. He shows that instead of giving full access to the control stack, allowing restricted access through a simple, easy to implement interface is sufficient to implement pretty much all the features and tools that we mentioned.

Benefits of Continuation Marks:

- can more easily give a formal specification to tools and language features implemented with continuation marks (ie - specified at a higher level)
- Continuation marks are simple enough that they can be added to existing virtual machines without affecting evaluation of existing programs (Clements, Sundaram, Herman - Implementing Continuation Marks in Javascript (Scheme Workshop 2008))
- can implement first class continuations (without implementing your VM in CPS) - PettyJohn, Clements, Marshall, Krishnamurthy, Felleisen - Continuations from Generalized Stack Inspection (ICFP 2005) – still requires whole-program instrumentation though

0.0.2 Continuation Marks Semantics

Adding continuation marks requires adding just two operations:

- `(with-continuation-mark key mark-expr body-expr)` or `w-c-m`
- `(current-continuation-mark key ...)` or `c-c-m`

Allows program to inspect its own call stack by labeling each frame in the control stack (key-value pairs). Labels can only be values available to the program. So continuation marks does not expose implementation details of the control stack (stack implementation, activation frame layout, stack depth)

$$\langle\ (\texttt{wcm}\ k\ C'\ C),\ E,\ K,\ M\ \rangle \rightarrow \langle\ C',\ E,\ (\textbf{wcm}:\ k\ C\ E\ K\ M),\ \emptyset\ \rangle$$

$$\langle\ V,\ E',\ (\textbf{wcm}:\ k\ C\ E\ K\ M),\ M'\ \rangle \rightarrow \langle\ C,\ E,\ K,\ M[k \rightarrow V]\ \rangle$$

$$\langle\ (\texttt{ccm}\ k\ldots),\ E,\ K,\ M\ \rangle \rightarrow \langle\ \pi(K,M),\ \emptyset,\ K,\ \emptyset\ \rangle$$

where $\pi(K,M) = [\phi(M_i) \mid M_i \in K \cup M]$ and $\phi(M) = [(\text{list}\ 'k_i\ V) \mid k_i \in k\ldots \text{ and } M(k_i) = V]$

0.0.3 Examples

2.1, 2.2, 2.4

0.0.4 Tail-calling

Another benefit of continuation marks that may be less obvious is that it preserves tail calls in many situations. One example is Racket's parameterize feature, which implements dynamic binding. Normally dynamic binding is implemented with something like dynamic-wind, which allows actions to be specified before and after a certain piece of code. However, the body of dynamic-wind is not in tail position because you need to reset the binding after executing the body. With continuation marks, you can add the "current binding" for a variable as a continuation mark.

Stack inspection is a security implementation strategy in various programming languages. The idea is to start with a small trusted computing base, which can then grant temporary permissions to perform certain actions to untrusted code. Proper authorization is verified by inspecting the runtime stack. Stack inspection is used by both the JVM and Microsoft's CLR VM. It was previously thought that stack inspection and tail calling were incompatible because removing stack frames may remove some permission information. Matthias says that Guy Steele told him that was the reason why Java does not implement proper tail calling. In Microsoft's CLR, tail calling must be disabled when using stack inspection. Clements and Felleisen (ESOP 2003) showed that stack inspection can be implemented in a tail-calling evaluator, using continuation marks.

0.0.5 Racket Stepper

When the stepper is invoked, each subterm is instrumented with a decompiled version of the source program (as tagged lists), as welll as a "breakpoint". At each breakpoint, the program is reconstructed using the stored continuation marks.