Stack Mechanisms

Ross Tate

Running Example

Dynamic Scoping

Dynamically Scoped Variables

- declare dynamically scoped variable “indentation : int”
- fun print-indented(s : Statement) {
  match s with
  | while-loop eguard sbody => println-indented(“while (” + format(eguard) + “) {”);
  { indentation++; print-indented(sbody); }
  println-indented(“}”);
  …}
- println-indented(s : String) { println(“\t” * indentation + s); }

Implementation Strategy #1

- Use a mutable global variable
- Tradeoffs
  - Very fast updates
  - Very fast lookups
  - Does not work with multiple threads
  - Correctness relies on unwinding

Implementation Strategy #2

- Pass around a mutable dictionary of dynamically-scoped variables
- Tradeoffs
  - Fastish updates
  - Fast lookups
  - Correctness relies on unwinding
  - Works with multiple threads

Implementation Strategy #3

- Pass around an immutable dictionary of dynamically-scoped variables
- Tradeoffs
  - Slowish updates
  - Medium-speed lookups
  - Robust with respect to unwinding
  - Works with multiple threads

WebAssembly Challenges

- Interop with host (e.g. JS)
  - calls from JS will not provide dynamic-scoping dictionary
  - closures given to JS will contain dictionary at time of closure-creation, not closure-invocation
  - same problem with interoperating with other applications
- Extensions for algebraic effects or first-class stacks
  - stack is a linked-list of “stacklets” (where code clearly indicates stacklet boundaries)
  - stacklets can be removed, added, rearranged, and so on
  - common for lookups of dynamically-scoped variables to reflect such changes to the stack

Dynamic Scoping with Stack Marking

- Key idea:
  - The stack is your dynamic scope
  - For variable updates: mark the *stacklet* with bindings of dynamically scoped variables
  - For variable lookups: search the *stack* for closest mark
- Tradeoffs
  - Works with foreign interop (marks are on stack before foreign stack frames)
  - Works with rearrangeable stacklets (marks move with their stacklets)
  - Requires treating stack as an inspectable data structure

Implementation Strategy #4

- Use code ranges to indicate which return addresses are marked
  - Check the associated stack frame to get the value of the mark
- Tradeoffs
  - Nearly free updates (only cost is constraint on register allocation)
  - Very slow lookups
  - Stack-frame cleanup is free
  - Good when variable is not likely to be used, bad when variable is likely to be used
  - “Lazy” strategy

Implementation Strategy #5

- Maintain dictionary of most recent marks (within stacklet) at root of stacklet
- Tradeoffs
  - Fastish updates
  - Fastish lookups
  - Stack-frame cleanup requires restoring dictionary
  - Assumes cheap way to find root of current stacklet from current stack pointer
  - Good when variable is accessed frequently, but wasteful when variable is not accessed
  - “Eager” strategy

Dynamic Scoping

- 5 implementation strategies
  - each with different pros and cons, each best suited for different situations
  - the application, not the engine, knows which is best
- Dynamic scoping is not a low-level primitive
- Stack marking is a low-level primitive
  - two variants: “eager” and “lazy”
  - useful (necessary?) for dynamic scoping that interops well with foreign calls and stacklets

Goal

Requirement

Mechanism

- Dynamic scoping would be the goal that we want WebAssembly to support
- Interop with foreign systems and composition with future extensions would be requirements
- Stack marking would be the mechanism that we actually add to Webassembly

Red Flag: Policy

- Imagine adding dynamic scoping directly to WebAssembly
- Question you would encounter: how should it interact with stacklet rearrangement?
  - Should the dynamic scope update or not?
  - That is a policy decision!
  - This suggests you are adding a high-level feature rather than a low-level mechanism
  - Break the feature down into parts