Pointers to General Resources on FP Language Compiler Construction

Ichinose Kaori

January 14, 2024

Step 0: Learn FP

- *Practical Common Lisp*
- *On Lisp*
- R^5RS
- *The Little Schemer*
- *Structure and Interpretation of Computer Programs*
- *ML for the Working Programmer*
- CS3110

TL; DW

- https://matt.might.net/articles/cps-conversion/
- https://matt.might.net/articles/compiling-scheme-to-c/
- https://github.com/akeep/scheme-to-c/
- http://churchturing.org/y/90-min-scc.pdf
- https://www.youtube.com/watch?v=Bp89aBm9tGU
- https://www.youtube.com/watch?v=M4dwcdK5bxE
- https://gist.github.com/nyuichi/1116686
- http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf
- ChezScheme/IMPLEMENTATION.md
- https://github.com/ichinosekaori/yass/ (possibly later)
- *Compiling with Continuations*

Goals

- To demonstrate compiling a functional programming language (Scheme) to a fairly low-level language (register VM bytecode)
- Do it using a simple functional language
- Do it using a fairly conventional VM (with the ISA mimicking commercial CPU designs, e.g. aarch64)

Overview of Scheme (1)

Primitive forms:

- Variable reference
- Quotation
- Procedure call
- Abstraction
- Assignment
- Conditional

Also derived forms programmed in the same language!
Some data usually not first-class are first-class: continuations, environments.

Overview of Scheme (2)

Primitive data structures:

- The Cons
- Vector
- (Bytevector)

Vectors are imperative arrays.
Data GCed.

Overview of the VM

- lea
- mov
- ld, st
- ldi
- $R_d \leftarrow R_a \,\mathrm{op}\, R_b$ (or unary; for arithmetic and logical operations)
- jmp
- je
- int (“hypercalls”)

where R can be X for 64-bit integers or D for double-precision floats.
VM for avoiding outputing PE/ELF/Mach-O or amd64/aarch64 machine code.

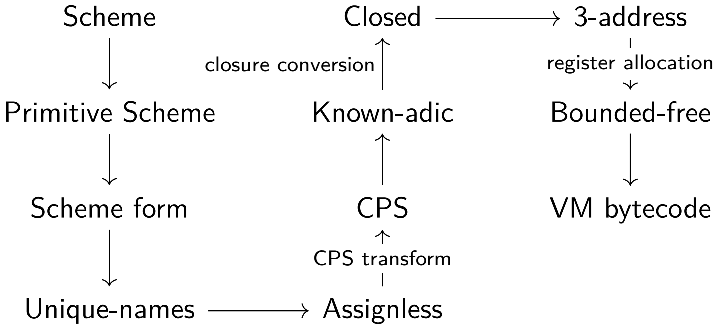
Merits of programming in Scheme

- small base
- extensibility
- `http://practical-scheme.net/docs/schemersway.html`

Gaps between source and target languages

- Nowhere to store computation state
- No notion of abstractions in the target
- Target works on numbers
- No memory management in the target
- Registers are limited in number

Compiler organization



Labeled passes are more traditional passes found in compilers for functional programming languages.

Passes specific to Scheme

- Macro expansion
- Unsplice
- Assignment conversion
- Variadic function elimination

Notes on continuations

- They represent "rest of the computation"
- Semantically is a function

Example

The continuation for the 2 in $2 + 3$ is $(\cdot + 3)$, and it for the 1×2 in $(2 \times 3) + (1 \times 2)$ is $(6 + \cdot)$. (assuming LtR evaluation order)

Rationale for CPS

- Explicit continuations for capture
- Continuations reified as functions for free
- Less code complexity
- All calls are in a tail context after CPS — space for control moved into the closure for the continuation
- More optimization opportunities

$$callcc(k, f) = f(\lambda k'.\lambda y.k(y), k)$$

CPS for the Lambda Calculus

Definition (Lambda terms)

$$t ::= x \mid \lambda x.t \mid t\ t$$

Theorem

$$\mathrm{CPS}(x, k) = k\ x$$

$$\mathrm{CPS}(\lambda x.t, c) = c(\lambda k.\lambda x.\,\mathrm{CPS}(t, k))$$

$$\mathrm{CPS}(t_1(t_2), k) = \mathrm{CPS}(t_1, \lambda r_1.\,\mathrm{CPS}(t_2, \lambda r_2.r_1(k)(r_2)))$$

Generalizing

Scheme has more primitives, including quotations, conditionals, and multiple arguments.

$$\mathrm{CPS}('a, k) = k('a)$$

$$\begin{aligned} &\mathrm{CPS}(\text{if } c \text{ then } a \text{ else } b, k) \\ = &\mathrm{CPS}(c, \lambda x.(\text{if } x \text{ then } \mathrm{CPS}(a, k) \text{ else } \mathrm{CPS}(b, k))) \end{aligned}$$

For generalizing to n-ary functions you need to bind all n operands to names, then apply.

Hidden code blowup!

$$\begin{aligned} &\mathrm{CPS}(\text{if } c \text{ then } a \text{ else } b, k) \\ =\ &\mathrm{CPS}(c, \lambda x.(\text{if } x \text{ then } \mathrm{CPS}(a, k) \text{ else } \mathrm{CPS}(b, k))) \end{aligned}$$

k appears twice — bind it before continuing!

CPS TL; DR

Copy from `https://matt.might.net/articles/cps-conversion/`.

The article has a fully-featured CPS transform implementation for Scheme.

Closure conversion rationale

Example

Consider the (different) return values of $\lambda x.\lambda y.x + y$.

$$\text{func} : \text{code} \times \text{any list}$$

Finding free variables of Lambda Calculus terms

Theorem

$$\mathrm{FV}(x) = \{x\}$$

$$\mathrm{FV}(t_1(t_2)) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$$

$$\mathrm{FV}(\lambda x.t) = \mathrm{FV}(t) \setminus \{x\}$$

No extensions to rules necessary for Scheme extensions to the lambda calculus.

Closure conversion

$$
\begin{aligned}
&\mathrm{ccvt}(T = \lambda x.t) \\
= &\,\mathrm{mkc}(\lambda c.\lambda x.\,\mathrm{ccvt}(t)[\forall s \in \mathrm{FV}(T).s \to \mathrm{cref}(c, s)], \mathrm{FV}(T))
\end{aligned}
$$

$$
\mathrm{ccvt}(t_1(t_2)) = (\lambda s.s(s)\,\mathrm{ccvt}(t_2))\,\mathrm{ccvt}(t_1)
$$

Embedding Scheme data into machine words

- tagged union (portable)
- tagged pointer (used by Chez)
- NaN boxing (used by V8)

Tagged pointers

- A word is 8-bytes long
- Pointers to 8-byte-aligned *things* will have 000 as their LSBs
- Use different values of the 3 LSBs to differentiate between types

See ChezScheme/IMPLEMENTATION.md.

Managing memory

- mark-sweep
- mark-compact
- mark-copy
- generational?
- concurrent?
- parallel?

Start simple: use Cheney's semispace algorithm

Register allocation

- Best: do whole-program RA and do coalescing (since control flow is broken up into slices after the CPS pass)
- Worse: *whatever correct*.

Ideas for more work

- More refined types
- Evaluation and environments
- Light processes
- Pattern matching
- Multi-dispatch methods
- Staged and safe code
- FBIP
- Zombie!
- Native backend
- More advanced RA/GC/optimizations

Slides at

`https://ichinosekaori.github.io/compiler-pointers.pdf`