Conscheme

Scheme on the Go

Per Odlund Göran Weinholt

Chalmers University

Frontiers of programming language technology, 2010

Motivation for the project

- If Go is a systems language it should be possible to make a fast Scheme with it (we need some pointer-level tricks).
- Trying to mimic Erlang (to some extent) in a Scheme environment.
- Trying to exploit the concurrency of Go.

Why another Scheme? Why Go?

- Easy to get started and to make something fairly complete and useful.
- Learning about Scheme. (Per didn't know Scheme).
- Learning about Go. (Göran didn't know Go).

Recap – Scheme

- A cleaned up dialect of Lisp. Simple syntax and semantics, small standard library.
- Guaranteed tail-call optimization.
- User-defined syntax fits in perfectly with existing syntax.
- Designed to be compiled – many language details were chosen to aid compilation.

Recap – Go

- A language with C-like syntax but safer semantics (like Java).
- Statically typed, but with support for dynamic typing.
- Classes and garbage collection.
- Minimalistic concurrency – goroutines. Function calls that are executed “in the background”.
- Systems language. Comes with a package for sidestepping the type safety.

Code organisation

- A compiler written in Scheme that produces bytecode for the virtual machine.
- A virtual machine written in Go.
- We started out with a very simple tree-interpreter written in Go. The interpreter was gradually replaced with a compiler and virtual machine.

Compiler design

- The front-end reads Scheme source code from a file or the terminal.
- Some of our compiler passes: macro expander, α-conversion, assignment analysis, free variable analysis, lambda lifting.
- Our backend produces a simple register-based bytecode.
- The compiler can compile itself. We bootstrapped with GNU Guile, but we no longer need any third-party Scheme implementation.

Virtual machine

- Runs our register-based bytecode.
- All registers contain Scheme objects, which are represented with a variant of low-tagging.
- Contains some low-level procedures: `car`, `null?`, `make-vector`, etc.

VM lowtagging

- Low-tagging uses the low bits of pointers to encode types. The traditional approach gave us problems with the GC.
- Low-tagging means that some object types fit completely in registers, e.g. 30-bit numbers. These are called immediates. Gives us faster arithmetic and fewer GC runs.
- All other objects use the Go type `*interface{}`. This basically wraps any Go value in a type descriptor.
- Before using Go's built-in type checking for interfaces we have to check that a value isn't an immediate.

More about the VM

- Provides some procedures to support threading: `make-thread`, `thread-start!`, `send`, `receive`, etc.
- Go has no thread-local variables, so a “current thread” object is passed around by the VM.
- The compiler runs on the VM and it can compile on-the-fly. We do not need an interpreter to run new code typed in by the user.

Echo server (Erlang code)

A simple echo server in Erlang.

```
echo() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            echo();
        stop ->
            true
    end.
```

Echo server (Conscheme syntax)

The echo server from the previous slide, but with Conscheme syntax.

```
(define (echo)
  (select
   (#(from msg)
    (send from (vector (current-thread) msg))
    (echo))
   (#('stop)
    #t)))
```

Discussion on Go

- Our message passing uses Go's channels. Not a perfect match with Erlang's concept, we can e.g. not guarantee relative message ordering.
- Threading was otherwise a one-to-one mapping onto Go.
- The types Foo and *Foo both seem to work the same until they don't.

Discussion on Conscheme

- Scheme has a standard threading library that we mostly ignored.
- Conscheme provides preemptive threading even though the current Go implementation doesn't.
- It has a lot of R^5RS, so it's usable for simple programming.
- Relatively fast. Our tree interpreter compiled Conscheme in around 2 minutes. The bytecode VM did the same thing in 20 seconds.

Summary

- Conscheme is a Scheme compiler written in Scheme.
- It runs on a virtual machine written in Go.
- Conscheme provides message passing inspired by Erlang.
- Go turned out to be suitable as a systems language, although the GC did get in the way just a little bit.

References

- Abdulaziz Ghuloum. An Incremental Approach to Compiler Construction. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*, pages 27–37.
- Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33, 9 (1998), 26–76*.
- The Go Programming Language. http://golang.org/
- The Conscheme website. https://github.com/weinholt/conscheme