The Loko Scheme Developer's Manual

Gwen Weinholt

Table of Contents

Preface

I've been writing Scheme code for some time now. While doing so I was also working on Scheme implementations. This is the first one that seems to have come out okay.

Gwen Weinholt, 2019

Purpose, audience and scope

This manual has two major parts: usage and internals. The first part is intended to let the developer start using Loko to write useful software. The second part goes into details on how Loko works and why things are the way they are.

Some knowledge of Scheme is assumed for the usage part. The reader who has no prior knowledge of any Lisp dialect will initially find it difficult to parse the language.

This is not a complete description of the Scheme language. The reader who wants a more detailed description of Scheme may want to read The Scheme Programming Language (`https://www.scheme.com/tspl4/`) (TSPL) by R. Kent Dybvig. The language described in that book is the same language that you can use in Loko Scheme.

This manual is also not a replacement for comments and descriptions in the code. Loko Scheme is not meant to be a closed box; you are supposed to open it up and look at the parts. Maybe even fix some to suit your situation. Loko Scheme needs its own source code to compile your programs, so every installation should come with source code.

Credits

Many have brought ideas, techniques and instructions to fruition that later went into making Loko Scheme. It would not be what it is without their contributions to science.

The syntax-case implementation is from *r6rs-libraries* by Abdulaziz Ghuloum and R. Kent Dybvig, with bug fixes and improvements from Llewellyn Pritchard.

The high-level optimizer *cp0* is based on the chapter *Fast and Effective Procedure Integration* from *Extending the Scope of Syntactic Abstraction* by Oscar Waddell (Ph.D. thesis).

The low-level optimizer is based on concepts taught in a course given by David Whalley in 2011.

The register allocator, except for the bugs, is from *Register Allocation via Graph Coloring* by Preston Briggs (Ph.D. thesis).

The letrec handling is from *Fixing Letrec (reloaded)* by Abdulaziz Ghuloum and R. Kent Dybvig.

The Unicode algorithms are also by Abdulaziz Ghuloum and R. Kent Dybvig.

The bignum algorithms are based on algorithms from *BigNum Math* by Tom St Denis.

The `list?` procedure uses Olin Shiver's version of Robert W. Floyd's cycle-finding algorithm.

The `equal?` procedure is from the paper *Efficient Nondestructive Equality Checking for Trees and Graphs* by Michael D. Adams and R. Kent Dybvig.

Some intricate parts of the records implementation are from the reference implementation of SRFI-76 by Michael Sperber.

The optimization of procedural records is based on the paper *A Sufficiently Smart Compiler for Procedural Records* by Andy Keep and R. Kent Dybvig. It's not as good, but it's something.

The list sorting code is from SLIB, was written Richard A. O'Keefe and is based on Prolog code by David H. D. Warren.

The dynamic-wind code is from SLIB and was written by Aubrey Jaffer.

The division magic, and many other wonderful hacks, is from the excellent book *Hacker's Delight* by Henry S. Warren, Jr., with foreword by one Guy L. Steele, Jr.!

The fibers library is loosely based on *Parallel Concurrent ML* by John Reppy, Claudio V. Russo and Yingqi Xiao. The API is based on Guile fibers by Andy Wingo and the implementation is closely related to his blog post *a new concurrent ml*.

The implementation of multiple values is based on *An Efficient Implementation of Multiple Return Values in Scheme* by J. Michael Ashley and R. Kent Dybvig. Advice contained wherein not heeded.

The R7RS-small standard library is based on code originally written by OKUMURA Yuki for the Yuni project.

The floating point to string conversion is based on Bob Burger's code described in the paper *Printing Floating-Point Numbers Quickly and Accurately*. Any bugs are our own.

The pretty printer comes from Marc Feeley's implementation written way back in 1991.

Thanks also to Abdulaziz Ghuloum for *An Incremental Approach to Compiler Construction*, which helped me consolidate the Scheme compiler experience I had already accumulated through experimentation.

How to License Loko Scheme

Loko Scheme is copyrighted software. The default legal state of software is that no rights are granted. However, Loko Scheme is licensed under a free software licence. This licence grants many permissions, but they are conditional on following the terms of the licence.

- Loko Scheme is licensed under EUPL-1.2-or-later (`https://joinup.ec.europa.eu/collection/eupl/eupl-text-eupl-12`). See the file `LICENSE` in the source code tree. If you haven't already, please go and read it. It's fairly short, easy to read, and available in many languages. The following points provide some guidance and explanations for your consideration, but they don't override the licence or any case law around it.
- The EUPL covers "software as a service" (SaaS) and other types of communicating Loko Scheme to the public. So if you make Loko Scheme available online as a compiler (e.g. like the Compiler Explorer at godbolt.org does with many other compilers) then that's the same as if you were giving your users a copy of the binary. Since the EUPL is a copyleft licence you then also need to provide the source. But this is limited to the "essential functionalities". Some possible scenarios where you have to give out the Loko Scheme source over the network is when you're using Loko Scheme as a compiler provided over the network, or if you're using the kernel functionality to provide a hosted execution environment.
- Programs compiled by Loko Scheme contain parts of Loko Scheme's runtime. The EUPL lets you combine Loko Scheme's runtime with your own program. See the

article Why viral licensing is a ghost (`https://joinup.ec.europa.eu/collection/eupl/news/why-viral-licensing-ghost`) for more details on this. This means that your program as a whole does not become licensed under the EUPL just because it was compiled with Loko Scheme; only those parts which were already under the EUPL. If it is your own program that you're compiling (even a commercial and proprietary program) then that is no problem. If the program is using another copyleft licence then there is potentially a problem if that other copyleft licence does not allow the combination with the EUPL. But the EUPL is designed to be compatible with several copyleft licenses, so it might still be okay. You can use the Joinup Licensing Assistant (`https://joinup.ec.europa.eu/collection/eupl/solution/joinup-licensing-assistant/jla-compatibility-checker`) to check if you can combine Loko Scheme's runtime with that other work.

- If you're going to distribute Loko Scheme (or a program compiled with Loko Scheme) then note in particular the obligations of the licensee in § 5. If you distribute binaries built with Loko Scheme then you can't remove the notices from the binary. When you distribute a binary of a work compiled with Loko Scheme then you are also distributing parts of Loko Scheme. So when someone receives a copy of that binary then they have the right to get your modified copy of Loko Scheme, according to the terms of the EUPL. But as per the previous point above, this does not affect your own code that is in that binary.
- Previous releases of Loko Scheme were under the GNU Affero General Public License (AGPL). This meant that the AGPL covered the Scheme runtime in the binaries that were generated by those versions, which then covered the whole binary. This severely limited the usefulness of Loko Scheme. To get around this problem it would have been necessary to formulate an exception to the license, but this proved to be very difficult in practice. The EUPL gets around this problem.
- The files under the srfi directory are published under the terms of the MIT license. This license is conventional for SRFI implementations and makes it easier to adapt them for use with other Scheme implementations, which you are encouraged to do. Just don't remove the copyright and licence notices.

The source code is automatically checked against the REUSE specification (`https://reuse.software/spec/`).

1 Introduction

1.1 Scheme

Scheme is a dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman in the mid 1970s. The first Lisp language was LISP, which was created by John McCarthy in the second half of the 1950s.

Lisp's syntax uses S-expressions, which were created by McCarthy to represent LISP functions as data in his *eval* function (published in 1960). *Eval* is a one-page universal LISP function capable of running any other LISP function. The first LISP interpreter was created when Steve Russell compiled the *eval* function by hand.

Today many languages offer a large subset of the features of Lisp, and some attempt to offer those of Scheme as well. S-expressions are not added because it would turn those languages into Lisps.

Distinctive features of Lisp languages are dynamic typing, garbage collection, and S-expressions ("symbolic expressions"). Scheme adds some additional distinctive features on top of those: static scoping, proper tail-recursion, hygienic macros, full continuations and *dynamic-wind*.

- Dynamic typing means that the programmer does not need to prove to the compiler that a program is well-typed. This is beneficial because there are true statements about programs that are impossible to prove except by running the programs.
- Garbage collection is the automatic reclamation of unused memory. This is beneficial because there are programs that are impossible to write without it.
- S-expressions provide a convenient means to represent code as data, in a format that the Lisp programmer is used to working with, which enables automatic transformations via macros and dynamic evaluation through *eval*. This is beneficial because there are programs that are impossible to write without dynamic code evaluation. It also relieves the programmer of having to guess how the parser will pick apart the code.
- Static scoping means that variable usage is connected with the binding of the variable as it appears in the text of the program. This is standard in most languages today, but Scheme is one of the few languages that gets this consistently right. (Dynamic scoping is available if needed through *dynamic-wind*, because there are some programs that cannot be written without it).
- Proper tail-recursion provides a guarantee that when a procedure call appears in a *tail context* (such as in the final expression of a procedure) no extra stack frame is used. This is beneficial because there are programs that cannot be written without proper tail-recursion.
- Hygienic macros means that a library can provide extensions to the Scheme syntax that are indistinguishable from built-in Scheme syntax, and hygiene means that these macros do not accidentally break static scoping. Less powerful macro systems are prone to the insertion of variables that conflict with those already used in the programs.
- (Continuations are usually explained in hopelessly abstract terms, so I will be very concrete here). *Continuations are copies of the stack* and the state associated with *dynamic-wind*. Many languages provide a way to escape upwards in the stack (e.g.

longjmp or exceptions), but continuations also lets programs restore the stack to what it looked like when the continuation was captured. Imagine throwing an exception, thereby going up the stack, and then having the exception handler fix the problem and then go back down the stack to resume. Scheme is also unusual in that it lets continuations be reinstated multiple times. Continuations are useful because there are programs that cannot be written without them. Any control structure can be expressed with them, even those not built in to the language.

- *Dynamic-wind* provides a means to run code when a part of the program is entered and then when it is exited. Continuations means that it can happen multiple times. This feature is useful because there are programs that cannot be written without it.

The features described above can be simulated in languages that lack them. Tail-recursion can be implemented manually by simulating a stack with a list, dynamic typing can be implemented with an abstract data type and type dispatching, garbage collection can be implemented for a data structure, and continuations can be implemented manually with continuation-passing style.

Such simulations are always possible in the sense that any Turing-complete language can implement any other Turing-complete language. But these simulations will not exist on the same level as the host language and are therefore of an inferior nature.

1.2 Scheme standards

Scheme is standardized through two different types of documents. The first are called the *Revised^n Reports on the Algorithmic Language Scheme (R^{n}RS)*. R^{5}RS came out in 1998 and was followed by R^{6}RS in 2007. R^{5}RS was also followed by R^{7}RS, which came out in 2013. Both of them are successors to R^{5}RS.

The second type of documents are called *Scheme Requests For Implementation (SRFI)*. This is a community-driven process whereby new language features can be developed and suggested for implementations to use. Many of them are mostly portable code, while other ones require adaptions to each Scheme implementation that wants to support them.

1.3 Where Loko fits in

Scheme has many implementations. Every known way to implement a programming language has probably been tried with Scheme. There are Scheme implementations for basically all operating systems and all types of machines. There have even been Scheme CPUs. Some say there are more implementations than applications. And Loko Scheme is one of those implementations.

Every Scheme implementation has something that makes it unique. This is what is peculiar about Loko:

- Loko runs on bare metal (and on top regular Unix kernels).
- Loko builds statically linked binaries.
- Loko is written in only Scheme and a small amount of assembly.
- Loko's ABI is incompatible with C and does not use it on any level.
- Loko's runtime uses concurrency based on Concurrent ML.
- Loko supports both R^{6}RS and R^{7}RS libraries and programs.

- Loko provides the safety guarantees of R^{6}RS even for R^{7}RS code.
- Loko uses the hardware for free type checking (branchless `car`, etc).

Due to some of the above, Loko Scheme is not suitable for every use case. There are plenty of other Scheme implementations available if Loko Scheme cannot work for your application.

2 Using Loko

2.1 Building Loko

Download a release tarball from `https://scheme.fail` or clone the git repository: `git clone https://scheme.fail/git/loko.git/`. Release tarballs, git commits and tags are signed with the OpenPGP key 0xE33E61A2E9B8C3A2.

The release tarball has everything needed for building Loko Scheme from GNU/Linux. It comes with the required Akku packages and a pre-built binary.

- *(Optional if you have the release tarball).*

  Loko Scheme needs an R6RS Scheme implementation for bootstrapping. It can currently be bootstrapped with Chez Scheme.

  Install Chez Scheme (`https://cisco.github.io/ChezScheme/`), version 9.5 or later, as a bootstrap compiler.
- *(Optional if you have the release tarball, but highly recommended and required for the samples).*

  Install the package manager Akku.scm (`https://akkuscm.org`), version 1.0.0 or later.
- Run `make` to compile Loko. GNU make is required.
- Install with `make install`.
- *(Optional)* Build and install the manual with `make install-info`.

The binaries tend toward being reproducible. There are some minor differences between the results from Chez Scheme and Loko Scheme due to different gensym implementations. This is fixable.

If you're building from Git and pull in updates, then it will at certain points be necessary to rebootstrap. This can be done with `make rebootstrap`, but it's a bit impractical at the moment due to the number of commits that require a rebootstrap. A simpler way is to remove the `loko-prebuilt` binary and bootstrap again from Chez Scheme (but that's cheating).

2.1.1 Cross-compiling

Loko Scheme can run on NetBSD/amd64 and you can get such a binary by cross-compilation.

Ensure that you have a working `loko-prebuilt` binary, e.g. by using the bootstrap target, or copy a working `loko` binary. Remove config.sls and uncomment the lines in the makefile for your target environment. (You can also pass them as arguments to make). Build the `loko` target as usual. You should now have a working binary for the target environment.

2.1.2 Loko from a distribution

If you are using Arch Linux then Loko should be available in AUR.

(Please get in touch if you're packaging Loko for a distribution so your package can appear here).

2.2 Running

Loko Scheme runs either under an existing operating system kernel (currently Linux and NetBSD), under a hardware virtual machine or directly on bare hardware.

2.2.1 Running under Linux or NetBSD

The `loko` binary is a statically linked ELF binary that the kernel can load directly.

Loko doesn't come with a built-in line editor. It is convenient to use rlwrap when running Loko: `rlwrap loko`. The rlwrap program adds readline on top of any program, providing line editing and history.

Loko uses the environment variable `LOKO_LIBRARY_PATH` to find libraries. This is a colon-separated list of directories. If you're using the package manager Akku then this variable is set when you active your project environment. The default list of file extensions are `.loko.sls`, `.sls` and `.sld`. They can be changed by setting `LOKO_LIBRARY_FILE_EXTENSIONS`.

R6RS top-level programs can be run from the command line with `loko --program program.sps`.

The `loko` binary is also meant to be installed under the name `scheme-script`. If it is invoked with this name it will load a Scheme script, as described in the non-normative R6RS appendix. It is often used like this:

```
#!/usr/bin/env scheme-script
(import (rnrs))
(display "Hello, world!\n")
```

By marking such a file executable the system will hand it over to `scheme-script`, which will then run it as a Scheme top-level program. But the name `scheme-script` is usually handled by the alternatives system, so it could be another Scheme that runs the script.

Such scripts can also be compiled to static binaries that can be run directly. See Section 2.3 [Compilation], page 12.

2.2.2 Running under KVM (QEMU)

To get a repl on the serial port:

```
qemu-system-x86_64 -enable-kvm -kernel loko -m 1024 -serial stdio
```

There is no echo or line editing, but it works alright as an inferior Scheme for Emacs. You can also try `rlwrap -a`.

If you create a script with this command then you can easily run it as an "Inferior Scheme" in e.g. Emacs. There are some additional options you can try:

- Add files to `/boot` using `-initrd filename1,filename2,etc`.
- Set environment variables with e.g. `-append LOKO_LIBRARY_PATH=/boot`.
- Pass command line arguments in `-append` by adding them after `--`, e.g. `-append 'VAR=abc -- --program foo.sps'`.

See the `samples` directory in the source distribution for more examples.

2.2.3 Running on bare metal

Loko on bare metal does not yet come with an adequate user interface. There is rudimentary log output to the text console during boot. This can be redirected, see Section 4.1.3 [Debug logs], page 23.

The first user process will be attached to the COM1 serial port (115200, 8n1). This is adequate for development until there is networking support. The `loko` program's first process is a repl, but if you compile a program then it will be your top-level program.

The `loko` binary should work with any Multiboot boot loader, such as GRUB 2. See the menu entry examples below.

2.2.3.1 Network booting

Network booting is possible if your hardware supports it. It has been tested with GRUB 2 and should also be possible with PXELINUX.

You will need to add an entry in the network's DHCP server and you need a computer where you can install a TFTP server such as tftpd-hpa.

- After installing the TFTP server, create a network directory with GRUB:

  ```
  grub-mknetdir --net-directory /tftpboot
  ```

  The TFTP server might be serving up another directory such as `/srv/tftp`.
- Create a configuration in `/tftpboot/boot/grub/grub.cfg` with an entry for Loko, such as this:

  ```
  menuentry "Loko Scheme" {
    multiboot  /loko loko
  }
  ```

  You can also include files that will be available in `/boot`:

  ```
  menuentry "Loko Scheme with foo library" {
    multiboot  /loko loko LOKO_LIBRARY_PATH=/boot
    module     /foo.sls foo.sls
  }
  ```

  Or start a program in the interpreter:

  ```
  menuentry "Hello world" {
    multiboot  /loko loko -- --program /boot/hello.sps
    module     /hello.sps hello.sps
  }
  ```
- Copy the Loko binary to the `/tftpboot` directory.
- Add an entry in the DHCP server. It can look like this if you're using ISC dhcpd:

  ```
  host darkstar {
    hardware ethernet 00:11:22:33:44:55;
    filename "boot/grub/i386-pc/core.0";
    next-server 192.168.0.2;
  }
  ```

  The hardware address must be changed to match the interface used for network booting (the machine that will run Loko). The server address is the address of the TFTP server.

2.2.4 Running in Docker

Loko can be run in a Docker container. There is sometimes no need to provide any other files in the container; Loko Scheme is self-sufficient.

These Docker images are provided:

- '`weinholt/loko:base`' – Loko Scheme only, a base image.
- '`weinholt/loko:latest`' – Loko Scheme with Debian GNU/Linux stable.
- '`akkuscm/akku:loko`' – Loko Scheme with Debian GNU/Linux and the package manager Akku.

2.3 Compiling a program

Loko can compile R6RS top-level programs:

```
loko --compile hello.sps
```

The above will compile the R6RS top-level program `hello.sps` and create the binary `hello`, which will run on Linux and bare metal.

To compile an R7RS program:

```
loko -std=r7rs --compile hello.sps
```

Libraries are looked up from the `LOKO_LIBRARY_PATH` environment variable (which is automatically set by the package manager Akku). The use of `eval` is disabled by default to speed up builds, but can be enabled with `-feval`:

```
loko -feval --compile hello.sps
```

The supported targets can be changed with `-ftarget=TARGET`. The default target is `pc+linux`. Other targets are `linux` for Linux only, `netbsd` for NetBSD only and `pc` for only bare metal.

An additional weird target is provided. It is called `polyglot` and creates binaries that run on Linux, NetBSD and bare metal. This is even less useful than it seems and may be removed in the future.

You can also omit the normal Scheme libraries. If you use `-ffreestanding` then only the assembler based runtime is added on top of the libraries that your program uses. This is useful mostly when you're working on the compiler. The source distribution has an example where this is used, `samples/hello/just-hello.sps`.

You can tune the parameters for the source-level optimizer (cp0). The argument `-fcp0-size-limit=N` sets the size limit and `-fcp0-effort-limit=N` sets the effort limit.

The command line is very inflexible, so try to stick to the fixed argument order for now.

Loko integrates its run-time into the resulting binary and Loko's source code needs to be available for compilation to succeed. The location is decided by `PREFIX` when compiling Loko, but can be overridden using the `LOKO_SOURCE` environment variable.

3 Library reference

3.1 Standard libraries

The standard R6RS Scheme (`https://r6rs.org`) libraries are provided. Please see the documents on the website for the official versions. Unofficial versions of R6RS updated with errata (`https://weinholt.se/scheme/r6rs/`) are also available online.

The standard R7RS-small (`http://r7rs.org`) libraries are provided as well. They are not available by default in the REPL and need to be imported with e.g. `(import (scheme base))`.

3.1.1 SRFI implementations

See the package chez-srfi (`https://akkuscm.org/packages/chez-srfi/`) for many SRFIs that work with Loko.

The following SRFIs are provided with Loko.

- `(srfi :19 time)` – time and date procedures. `https://srfi.schemers.org/srfi-19/srfi-19.html`. The copy that exists in chez-srfi is only used up to and including Loko Scheme 0.6.0.
- `(srfi :38 with-shared-structures)` – implemented using the usual write procedure. `https://srfi.schemers.org/srfi-38/srfi-38.html`.
- `(srfi :98 os-environment-variables)` – read-only access to environment variables. Documented at `https://srfi.schemers.org/srfi-98/srfi-98.html`.
- `(srfi :170 posix)` – access to common POSIX operations, currently only available on Linux. Documented at `https://srfi.schemers.org/srfi-170/srfi-170.html`.
- `(srfi :215 logging)` – central log exchange. This is used in Loko Scheme as a target for all kinds of logs, including crashes in fibers and log messages from drivers. Documented at `https://srfi.schemers.org/srfi-215/srfi-215.html`.

The package loko-srfi (`https://akkuscm.org/packages/loko-srfi/`) also provides SRFIs. This package is for SRFIs that create too much entanglement through their dependencies to be suitable for either chez-srfi or Loko Scheme proper. At the time of writing it provides SRFI 106 (basic sockets) for Linux.

3.2 Base library

The `(loko)` library is automatically loaded into the repl. It provides all exports from the R6RS libraries, SRFI 98, and the exports shown below. It is intended as a convenient starting point for the repl user and a place to put features that are expected from a Scheme implementation, but that do not belong to any other library.

`interaction-environment` [Procedure]

`load` *filename* [Procedure]

`include` *filename* [Procedure]

Include code into the program from *filename*, as if it had been written in the place of the `include` form.

`void` [Procedure]

Returns a void object.

The value that came from nowhere. Even though it has never been standardized, it is customary for Scheme implementations to use void for `(if #f #f)`, `(values)` and the value of `set!` and other mutating operations.

In Loko Scheme, void objects carry a copy of the program counter.

```
(list (void))
⇒ (#<void #x21731F>)
```

`library-directories` [Parameter]

This parameter is a list of strings that name directories to check when importing libraries.

Default: `(".")`

`library-extensions` [Parameter]

This parameter is a list of strings with file extensions to use when importing libraries.

Default: `(".loko.sls" ".sls" ".ss" ".scm")`

`installed-libraries` [Procedure]

For use in the repl. Returns a list of libraries.

`uninstall-library` *name* [Procedure]

For use in the repl. Uninstalls the *name* library.

`environment-symbols` *env* [Prrocedure]

The list of symbols defined in the environment *env*.

`expand` *expr* [Procedure]

Expands the expression, returning core forms. The format of the returned forms should not be relied on.

`expand/optimize` *expr* [Procedure]

Expands and optimizes the expression, returning core forms. The format of the returned forms should not be relied on.

`cp0-size-limit` [Parameter]

Limits how much the source-level optimizer cp0 will allow the code to grow.

Default: `16`

`cp0-effort-limit` [Parameter]

Limits the effort spent by the source-level optimizer cp0.

Default: `50`

`disassemble` *procedure* [Procedure]

Print the disassembly of *procedure*. It is annotated with labels for local jump destinations and some simple code equivalents.

```
> (disassemble car)
Disassembly for #<procedure car loko/libs/pairs.loko.sls:3224>
```

```
 entry:
  206E00 83F8F8       (cmp eax #xFFFFFFF8)
  206E03 0F8505000000 (jnz L0)
; (set! rax (car rdi))
  206E09 488B47FE     (mov rax (mem64+ rdi #x-2))
  206E0D C3           (ret)
 L0:
  206E0E E96DA2FFFF   (jmp (+ rip #x-5D93))
```

`machine-type` [Procedure]

The machine type that Loko is running on. This is a vector where the first element is the CPU type `amd64` and the second is the system environment (`linux`, `netbsd` or `pc`).

`time` *expr* [Syntax]

Run the procedure *thunk* once with no arguments and print some numbers of memory allocation and elapsed time.

`time-it` *what thunk* [Procedure]

This is the procedural version of `time`.

`time-it*` *what iterations thunk* [Procedure]

Run *thunk* repeatedly *iterations* times and print some bogus statistics. The aim is that this procedure should be the best way to do micro benchmarks.

Please note that *iterations* is rounded upwards to some multiple close to the time stamp counter resolution. This procedure is not meant to be used for long-running procedures, the typical case is something that takes at most a few dozen cycles, at most a few thousand.

The code under test should also be compiled ahead of time for the results to reflect more than the interpreter's overhead. In the example below, code is not compiled.

```
> (time-it* "fx+" 10000000 (lambda () (fx+ x 1)))
Timing fx+ to find the minimum cycle time:
New minimum is 1819 cycles with 10000000 iterations to go.
...
New minimum is 234 cycles with 6257346 iterations to go.

  The cycle count varied between 234 and 83160784
  (Arithmetic mean)      μ  = 248.75
  (Standard deviation)   σ  = 24.33
  (Population variance)  σ² = 592.08
                    min x_i = μ-.61σ
  Used 9736890 samples (263110 outliers discarded).
234
> (time-it* "+" 10000000 (lambda () (+ x 1)))
Timing + to find the minimum cycle time:
New minimum is 1751 cycles with 10000000 iterations to go.
```

```
...
New minimum is 240 cycles with 9968540 iterations to go.

  The cycle count varied between 240 and 84141254
  (Arithmetic mean)      μ  = 252.96
  (Standard deviation)   σ  = 30.46
  (Population variance)  σ² = 927.82
                    min x_i = μ-.43σ
  Used 9979862 samples (20138 outliers discarded).
240
```

Note that cp0 will optimize the thunk before it runs, so you may end up benchmarking something other than what you thought. Check with `expand/optimize`. If the code is entered in the REPL then you also measure the overhead of `eval`.

Modern computers are notoriously difficult to get any consistent results from. An improvement in cycles could be because the code slightly moved in memory. See Producing Wrong Data Without Doing Anything Obviously Wrong (`https://john.cs.olemiss.edu/~hcc/researchMethods/notes/localcopy/mytkowicz-wrong-data.pdf`) (2009, Mytkowicz, et al). A more lively view of the problem is the presentation Performance Matters (`https://youtu.be/r-TLSBdHe1A`) (2019, Emery Berger at Strange Loop).

`open-output-string` [Procedure]

Make a new string output port that accumulates characters in memory. The accumulated string can be extracted with `get-output-string`.

`get-output-string` *string-output-port* [Procedure]

Extract the accumulated string in `string-output-port` and reset it. Returns the string.

`port-file-descriptor` *port* [Procedure]

Get the file descriptor associated with *port*. Returns `#f` if there is no associated file descriptor.

`port-file-descriptor-set!` *port fd* [Procedure]

Set the file descriptor associated with *port* to *fd*.

This procedure is primarily intended to allow custom ports to have file descriptors. It is unspecified whether changing a port's file descriptor affects the file descriptor used for subsequent operations on the port.

`gensym` [Procedure]

Generate an uninterned symbol. These are symbols which are not `eq?` to any other symbol.

`make-parameter` *default-value* [*fender*] [Procedure]

Create a new parameter object. Parameters are typically used to implement dynamically scoped variables together with `parameterize`. A parameter's current value can be queried by calling it with no arguments and its value can be modified by calling it with one argument, the new value.

The optional *fender* procedure is applied to the value whenever the parameter is modified. The return value of *fender* is used in place of the new value. A typical use of this procedure is to do some type checks on the new value.

`parameterize` ((*name value*) ...) *body...* [Syntax]

Parameterize rebinds the parameter *name* to *value* for the dynamic extent of *body*. This means that while *body* is running, *name* will be set to *value*. The value is possibly filtered by a fender procedure.

Whenever the program leaves the body, either by a normal return or a non-local exit (such as in a `guard` expression or by calling a continuation created by `call/cc`), the value is reset to the value it has outside of the body. If control reenters body, as in a call to a continuation created inside the body, the parameter will return to the value established by `parameterize`.

Although it has the same name, this syntax is a faster variant that is not fully compatible with SRFI-39. This variant is very common in Scheme implementations and matches the one used in e.g. Chez Scheme.

`loko-version` [Procedure]

The version number of the Loko Scheme runtime. This is a SemVer number and may include build information in the future.

`putenv` *name value* [Procedure]

Set the environment variable *name* to *value*. The *name* is always string.

If the *value* is a string then the variable is set to that value.

If the *value* is `#f` then the variable is removed.

This updates the environment used by SRFI 98 and can also be expected to be visible to any child processes started after the call.

The strings must not contain any `#\nul` characters. The *name* must not contain a `#\=` character. These limitations are not checked.

The names and values are always transcoded to/from UTF-8 in the POSIX interfaces.

Please beware that other Scheme implementations commonly leak memory through this procedure.

`collections` [Procedure]

The number of garbage collections.

`module` [*name*] (*exports* ...) *body* ... [Syntax]

Define a module. A module is like a library, but it uses an syntax which does not exist in any RnRS standards. It can also appear inside a library and is commonly used to hide internal definitions.

It is better to not use this syntax in your own code because it makes your code non-portable. It is provided for compatibility with other Scheme implementations.

`load-program` *filename* [Procedure]

Load and run the R6RS program *filename*.

`pretty-print` *obj* [*port*] [Procedure]

Writes *obj* to *port* using a `write`-compatible notation. Extra spaces and newlines are inserted to make the output more readable to a human.

3.3 Apropos

The `(loko apropos)` library exports procedures for looking up symbols in environments.

`apropos-list` *name* [*env*] [Procedure]

Search the names of the environment *env* and all loaded libraries for symbols containing the substring *name*, which can be a string or a symbol.

If *env* is omitted then the default is to use the interaction environment.

The returned list contains entries of these formats:

- (*library symbol*) – this means that *library* exports the *symbol*. The library is loaded, but it may need to be imported.
- *symbol* – this means that *symbol* is available in *env*.

This procedure should be compatible with the similarly named one in Chez Scheme.

`apropos` *name* [*env*] [Procedure]

This is an analogue of the `apropos-list` procedure that is meant for interactive use.

3.4 Fibers

The `(loko system fibers)` library exports procedures for fibers. Fibers are a form of lightweight concurrency based on Concurrent ML. For an overview, see Section 6.1 [Concurrency], page 27.

`spawn-fiber` *thunk* [Procedure]

Create a new fiber that will start running the procedure *thunk*, which takes no arguments.

`make-channel` [Procedure]

Create a new channel. Channels are places where two fibers can rendezvous to exchange a message. There is no buffering in a channel.

`channel?` *obj* [Procedure]

True if *obj* is a channel.

`put-message` *channel obj* [Procedure]

Put the message *obj* on the channel *channel*. Blocks until another fiber picks up the message. Returns unspecified values.

`get-message` *channel* [Procedure]

Get a message from the channel *channel*. Blocks until another fiber has arrived with a message. Returns the message.

`sleep` *time* [Procedure]

Block the fiber for *time* seconds.

`put-operation` *channel obj* [Procedure]

Returns an operation object that represents putting the message *obj* on the channel *channel*.

`get-operation` *channel* [Procedure]
: Returns an operation object that represents getting a message from the channel *channel*.

`wrap-operation` *op f* [Procedure]
: Returns an operation object that is the same as the operation *op*, except that the values a wrapped by the procedure *f*.

`sleep-operation` *seconds* [Procedure]
: Returns an operation object that represents waiting until *seconds* have passed from the time of the call to this procedure.

`timer-operation` *a* [Procedure]
: Return an operation object that represents waiting until absolute time *a* (in internal time units).

`choice-operation` *op . . .* [Procedure]
: Returns an operation object that represents a choice between the given operations *op* .... If multiple operations can be performed then one is selected non-deterministically.

It is not an error to call this procedure with no arguments. It is in fact a useful construction when gathering operations.

If `wrap-operation` is used on a choice operation then every operation will be wrapped.

`perform-operation` *op* [Procedure]
: Perform the operation *op*, possibly blocking the fiber until the operation is ready.

With `choice-operation` and `perform-operation` it's possible to write code that waits for one of several operations. This can be something simple like waiting for a message with a timeout:

```
(perform-operation (get-operation ch) (sleep-operation 1))
```

This example will wait for a message on the channel *ch* for up to one second. In order to distinguish between a message and a timeout, `wrap-operation` is used:

```
(perform-operation
 (choice-operation
  (wrap-operation (get-operation ch) (lambda (x) (cons 'msg x)))
  (wrap-operation (sleep-operation 1) (lambda _ 'timeout))))
```

This code will either return `(msg . x)` where `x` is the received message; but if more than one second passes without a message it returns `timeout`.

The object returned from `choice-operation` can be returned from a procedure, stored in a data structure, sent over a channel, etc.

`make-cvar` [Procedure]
: Make a new *condition variable* (in Concurrent ML's terminology). These allow a program to wait for a condition to be signalled. See the procedures below.

`cvar?` *obj* [Procedure]
: True if *obj* is a condition variable.

`signal-cvar!` *cvar* [Procedure]

Signal the condition variable *cvar*, unblocking any fibers that are waiting for it.

`wait` *cvar* [Procedure]

Wait for the condition variable *cvar* to be signalled, blocking until it is.

`wait-operation` *cvar* [Procedure]

Return an operation that represents waiting for the condition variable *cvar* to be signalled.

`yield-current-task` [Procedure]

Yield the current task and and let another fiber run. This is generally not needed in I/O-bound programs, but is provided to let CPU-bound programs cooperate and voluntarily let other fibers run.

`exit-current-task` [Procedure]

Stops the running fiber.

`run-fibers` *init-thunk* [Procedure]

Provided for compatibility with Guile. It runs the procedure *init-thunk* in the fibers scheduler. This procedure can return earlier in Loko than in does in Guile. Guile provides it because fibers are not an integrated feature in its runtime, so it needs an entry point for when to start and stop the fibers facility.

3.5 Unsafe procedures

The `(loko system unsafe)` library provides raw access to kernel services, linear memory and I/O bus registers.

`syscall` *n arg . . .* [Procedure]

Calls the kernel's system call number *n* with the arguments *arg . . .*. Returns a fixnum.

Example fork on Linux amd64:

```
(when (zero? (syscall 57))      ; __NR_fork
  (display "child process\n")
  (exit))                       ; child become a zombie
```

Scheme programs should generally not be written directly with syscalls any less than C programs would do the same. There are usually interactions with the standard library that should be considered, such as flushing of ports to prevent duplicated output.

`bytevector-address` *bytevector* [Procedure]

Get the linear address of the first byte of *bytevector*.

The first byte is guaranteed to have an alignment of eight bytes.

A moving garbage collector is used for bytevectors created with `make-bytevector`. There is no way to ensure that they do not move during GC, so their addresses should not be used to perform bus-mastering DMA.

Returns a fixnum.

`get-mem-u8` *addr* [Procedure]
`get-mem-u16` *addr* [Procedure]
`get-mem-u32` *addr* [Procedure]
`get-mem-s61` *addr* [Procedure]

Read a u8, u16, u32 or s61, respectively, from linear address *addr* and return it as a fixnum.

- If *addr* is unaligned then an exception is raised.
- On targets where fixnums are not wide enough to hold the result, a bignum will be used instead, but the bus access will be the correct width.
- Calls to these procedures will not be optimized away or reordered by the compiler.
- Each call corresponds to one memory read instruction of the matching size. For the signed 61-bit procedure this only applies to targets that can issue 64-bit memory bus operations.
- These calls may be issued out of order by the processor or go to the cache. This is usually only a problem when the memory area is backed by RAM. On AMD64 it is safe to read memory-mapped hardware registers with these procedures thanks to MTRR. Other situations may require locking instructions or memory barriers.

`put-mem-u8` *addr n* [Procedure]
`put-mem-u16` *addr n* [Procedure]
`put-mem-u32` *addr n* [Procedure]
`put-mem-s61` *addr n* [Procedure]

Write *n* as a u8, u16, u32 or s61, respectively, to linear address *addr*.

- If *addr* is unaligned then an exception is raised.
- Calls to these procedures will not be optimized away or reordered by the compiler.
- Each call corresponds to one memory write instruction of the matching size. For the signed 61-bit procedure this only applies to targets that can issue 64-bit memory bus operations.
- These calls may be issued out of order or be merged by the processor. This is usually only a problem when the memory area is backed by RAM. On AMD64 it is safe to read memory-mapped hardware registers with these procedures thanks to MTRR. Other situations may require locking instructions, manual flushes or memory barriers.

Returns unspecified values.

`get-i/o-u8` *busaddr* [Procedure]
`get-i/o-u16` *busaddr* [Procedure]
`get-i/o-u32` *busaddr* [Procedure]

Read a u8, u16 or u32, respectively, from I/O bus address *busaddr* and return it as a fixnum.

The `get-i/o-u32` procedure may return a bignum on targets where `(<= (fixnum-width) 32)`, but the bus access will be 32-bit.

`put-i/o-u8` *busaddr n* [Procedure]
`put-i/o-u16` *busaddr n* [Procedure]

`put-i/o-u32` *busaddr n* [Procedure]

Write *n* as a u8, u16 or u32, respectively, to I/O bus address *busaddr*.

Returns unspecified values.

`get-i/o-u8-n!` *busaddr addr n* [Procedure]
`get-i/o-u16-n!` *busaddr addr n* [Procedure]
`get-i/o-u32-n!` *busaddr addr n* [Procedure]

Read *n* units of u8, u16 or u32, respectively, from I/O bus address *busaddr* and write them to memory starting at linear address *addr*.

The address *addr* must be naturally aligned to the size of the writes. Otherwise an `&assertion` is raised.

Returns unspecified values.

`put-i/o-u8-n` *busaddr addr n* [Procedure]
`put-i/o-u16-n` *busaddr addr n* [Procedure]
`put-i/o-u32-n` *busaddr addr n* [Procedure]

Write *n* units of u8, u16 or u32, respectively, to I/O bus address *busaddr* while reading them from memory starting at linear address *addr*.

The address *addr* must be naturally aligned to the size of the reads. Otherwise an `&assertion` is raised.

Returns unspecified values.

3.6 Target libraries

The following Linux-specific libraries are provided. Their usage mirrors 1:1 the functionality in the Linux manpages, section 2.

- `(loko arch amd64 linux-numbers)` – constants used in the Linux syscall interface (UAPI).
- `(loko arch amd64 linux-syscalls)` – thin wrappers around Linux syscalls.
- `(loko arch amd64 netbsd-numbers)` – constants used in the NetBSD syscall interface.
- `(loko arch amd64 netbsd-syscalls)` – thin wrappers around NetBSD/amd64 syscalls.

4 Repair instructions

4.1 Tools support

This section describes some tools can be used together with Loko Scheme when things go wrong.

4.1.1 Disassembly

Loko can be disassembled using any regular disassembler that supports ELF and amd64, such as the one in GNU binutils and GNU gdb. There is also a built-in disassembler for procedures, see Section 3.2 [Base library], page 13.

4.1.2 Debugging

Loko can be debugged with GNU gdb (`https://www.gnu.org/software/gdb/`) with the help of `loko-gdb.py`. Stack traces and pretty printing should work. Line information is currently missing.

Scheme objects can get very big. Limit the output with e.g. `set print elements 5`.

Traps from the processor are translated into conditions with a `&program-counter` condition, e.g. this error from trying to evaluate `(#f)`:

```
The condition has 5 components:
 1. &assertion &violation &serious
 2. &who: apply
 3. &message: "Tried to call a non-procedural object"
 4. &irritants: (#f)
 5. &program-counter: #x309795
End of condition components.
```

You can look up the trapping instruction with a disassembler.

4.1.3 Debug logs

The PC port of Loko can have debug logging redirected from the VGA console with the `CONSOLE` environment variable:

- `CONSOLE=vga` prints to the VGA text mode console.
- `CONSOLE=com1` prints to COM1 (BIOS's default baud rate).
- `CONSOLE=debug` prints to QEMU's debug console. This is enabled with e.g. `-debugcon vc`.

4.1.4 Profiling

Loko on Linux can be profiled with perf (`https://perf.wiki.kernel.org/index.php/Main_Page`).

Some micro benchmarks can be done from inside Loko, see Section 3.2 [Base library], page 13.

4.1.5 Memory checking

It's possible to run Loko in Valgrind (`http://valgrind.org/`). Valgrind does not support alignment checking, so Loko will print a warning about that. Loko also does not use the "red zone", so Valgrind will think that a lot of what Loko is doing uses uninitialized memory. Don't believe it.

4.1.6 Fuzzing

Loko can be used with AFL++ (`https://aflplus.plus/`). This tool can explore all possible paths through a program in order to find crashes. You can use it to automatically check that, e.g., a parser does not crash on any inputs.

The way to use it is to prepare a small program that reads from the standard input and passes it to the code under test. Build your program with `loko -fcoverage=afl++ program.sps`. This flag tells Loko to add branch instrumentation for AFL++. There is some overhead associated with this code, so only use it during fuzzing.

The instrumented binary will mutate a memory area that is shared with `afl-fuzz`. Every `if` expression (including syntax which expands to an `if` expression) will mutate the area differently depending on whether the true or the false branch was taken. This should mean that the area gives a unique fingerprint for each path taken through the program.

When fuzzing a program built with Loko you should normally run the fuzzer with `AFL_CRASH_EXITCODE=70 afl-fuzz -i inputs/ -o outputs/ -- ./program`, where inputs is a directory with files that contain sample inputs.

It makes sense to verify that the interaction between AFL++ and the program is working as intended and that AFL++ detects crashes. This can be done by introducing an explicit crash in the program for certain inputs.

A full discussion about AFL++ is out of scope for this manual. Please see the AFL++ website for more reading material.

The instrumentation support has been tested with AFL++ 4.04c.

5 Other resources

5.1 Loko Scheme website

The Loko Scheme (`https://scheme.fail`) website has official release tarballs and an online copy of the manual.

5.2 Online communities

There are two chat channels available on IRC. They are on the IRC network Libera.Chat (`https://libera.chat`) and they are `#scheme` and `#loko`. The `#scheme` channel is for the whole Scheme community. You will need an IRC client to join these channels.

If you're more of a forum or mailing list user, then there aren't any direct options like this yet. But we have Usenet, which has filled this need even since before there was an Internet. The community-wide Scheme group is called '`comp.lang.scheme`'. It is accessible via any Usenet provider, even free ones like Eternal September (`http://www.eternal-september.org/`). It is also accessible via one of the big search engine companies.

There is also a community-wide Scheme Wiki at `http://community.schemewiki.org`.

Another resource that has popped up recently is Discord. There is a Scheme Discord (`https://discord.gg/ZcTYrdx`).

5.3 Issue tracker

There may be an issue tracker at GitLab (`https://gitlab.com/weinholt/loko`), but please instead use channels where issues nobody cares about can be lost in the mists of time. You can send bug reports to `bugs@scheme.fail`.

5.4 Package repositories

There are repositories of packages online where you can find many useful libraries that work with Loko:

- Akku.scm (`https://akkuscm.org/`) – an R6RS/R7RS package manager. Akku also translates R7RS libraries to R6RS to try and make them work with R6RS implementations.
- Snow (`http://snow-fort.org/`) – an R7RS package manager. These packages (or snowballs) are also included in Akku's repository.

6 Loko internals

6.1 Concurrency

The concurrency in Loko exists at two different levels:

- Loko processes, which are preemptible processes running at the same privilege level and in the same page table as the Loko runtime.
- Loko fibers, which are lightweight processes that run inside a Loko process. They are an implementation of Concurrent ML (`https://people.cs.uchicago.edu/~jhr/papers/cml.html`).

When Loko starts it immediately sets up a Loko process for the scheduler (also called pid 0). The boot loader has already created a heap and stack for it. The scheduler is responsible for starting pid 1, handling interrupts, managing preemption, message passing between processes and other maintenance tasks. The first scheduler that starts is also responsible for booting all other processors in the system.

Currently all other processors boot up but stop after initialization. Some work needs to be done to allocate new scheduler processes for them.

Apart from the scheduler and Loko processes with fibers there are also normal usermode processes with their own page tables. This is where you can put all your FORTRAN and C programs.

Fibers are a lightweight concurrency system based on Concurrent ML. The implementation is heavily inspired by a Andy Wingo's articles Growing Fibers (`https://wingolog.org/archives/2017/06/27/growing-fibers`). and A New Concurrent ML (`https://wingolog.org/archives/2017/06/29/a-new-concurrent-ml`). Concurrent ML forms a fundamental principle for concurrency that can be used to implement higher-level concurrency like Go channels or Erlang processes.

For details on how to use fibers in your program, see Section 3.4 [Fibers], page 18. You can also consult the Guile fibers manual (`https://github.com/wingo/fibers/wiki/Manual`) to some extent; it has a lot of background information.

The implementation in Loko is different from Guile fibers in these ways:

- Loko fibers need the import `(loko system fibers)` rather than `(fibers)`.
- Loko fibers are based on pure `call/cc` to switch between fibers and the fiber scheduler. This is mostly because the Loko runtime doesn't have delimited continuations, but it made it easier to quickly get the correct semantics for parameters.
- However, basing the implementation on `call/cc` brings a space leak. It means that there is some extra overhead from lugging around the unused parts of the continuation and the dynamic winders. In some programs, this state can potentially grow without bound. This could be solved without implementing delimited continuations, but extra support from the runtime is needed either way.
- Loko fibers are not preemptively scheduled. This is because of several reasons, explained in the next section. This means that you shouldn't run long-running computations in a fiber that shares a Loko process with other fibers that need to be responsive (unless you explicitly yield the fiber every now and then).

- Loko fibers in a process do not run in parallel; they are not shared between processors. The way memory allocation is done in Loko means that two processors can't manage the same heap.

XXX: The implementation described above has a broken `dynamic-wind`*. See issue #26 in the bug tracker.*

The API is compatible with Guile fibers, so the concurrency parts of a program written for Guile fibers should work with Loko fibers. The largest exception is Guile's `(fibers internals)` library, which manages fiber schedulers. Loko only has one of those per Loko process. Another difference is that I/O is non-blocking by default on Loko.

A large part of what makes fibers attractive is that code can be written as if it were non-concurrent. You're free to read and write to pipes and network streams without explicitly dealing with polling for when data is available or when file descriptors are ready for writing. One of the sample programs is a tiny web-server that spawns a fiber per connected client.

Loko on Linux takes care to set `O_NONBLOCK` on file descriptors and suspends the current fiber when syscalls return `EAGAIN` (also called `EWOULDBLOCK`). Loko uses epoll to find out when the file descriptor will be ready.

But Linux does not implement `EAGAIN` for regular files, so reading from a file can block. When this happens it prevents other fibers from running. Standard I/O is non-blocking, but other programs that use the same terminal can either get confused by that and/or turn it off. Even memory accesses can be blocking on Linux because pages can be swapped out to disk. (Turning off swap is not a good idea). The binary for your program was mmap'd by Linux when it got started and unmodified pages can be read back from disk, so Linux can freely evict the pages from memory. So having anything like a guaranteed responsive program on Linux is challenging. If anything goes wrong with the disk, all your processes can end up in uninterruptible sleep.

Loko on bare hardware has no operations that block other fibers from running.

6.1.1 Why fibers are not preemptible

Here's the excuse. Loko processes can temporarily use registers in such a way that they contain arbitrary bit patterns. If such a register were to be saved to a continuation object, the garbage collector would choke on it. Other fibers in the same Loko process use the same heap, so a fiber can't simply be suspended and left alone as Loko processes are when they're preempted.

One common solution to this problem is that the compiler inserts counters at various points in the code. These counters are incremented at safe points in the code and are used to a implement software-based timer interrupts. This solution brings with it some overhead and needs special care to not ruin the performance of tight loops. It may be done later unless another solution is found.

6.1.2 Loko processes

The use case for processes is pretty slim at this time, but they are the only way to get preemptive concurrency.

Loko on Linux currently has a very rudimentary scheduler that can't handle more than one process.

6.2 Drivers

A driver is a piece of code that allows for abstract access to a hardware device. In Loko Scheme they are collected in the `drivers` directory.

6.2.1 Driver abstractions

A large part of creating drivers is to adapt the hardware to the abstractions used in the rest of the system. Sometimes this means that the full capabilities of the hardware are hidden. A serial port supported by a UART may appear as a Scheme input port and an output port, which will allow you to hook it up to any Scheme code that works with ports. But a UART can do much more than a Scheme port can. An output port can't usually control baud rates or send breaks.

Scheme lacks abstractions for hardware. In the RnRS standards, files and ports are the only things that work as abstractions for hardware. This is not so bad, because Unix systems have shown that a lot of hardware can be represented as special files in `/dev` and the file descriptors you get when opening the special files.

However, these file descriptors are just handles that let user space communicate with the driver. Sometimes the communication is through an abstraction layer and sometimes it goes almost directly to the driver. User space needs to use special syscalls (e.g. `ioctl`) to actually do anything interesting that is not a read/write operation.

Drivers for Loko should not be locked in to any particular way of designing the user space interface. Decisions like that are taken on a different level. This will let developers experiment with different user space designs. Designs like Barrelfish (`http://www.barrelfish.org`), where hardware virtualized devices are handed out directly to user space, should be possible to express.

Drivers in Loko should be written as libraries that provide convenient APIs. These APIs should provide basic functionality in a way that is common between similar hardware of the same type. The interfaces will need to be developed over time.

When it is necessary to have concurrency (the most common case for modern devices), drivers should use channels to communicate with the rest of the system. Concurrent drivers can start as many fibers as they like. The messages sent on these channels should preferably be simple objects like vectors, symbols, pairs and fixnums. The messages become part of the driver API.

6.2.2 Hardware access

Drivers need access to their devices. The way this is done depends on what type of bus the device is attached to.

Modern PCs have busses that can be probed, like PCI and USB. This process provides enough information that you can easily detect the type of devices that are on the bus and how to access them. Hardware tends to appear like a tree-like structure, so it is natural that a bus driver will pass along a reference to the bus down in the call stack.

Older devices on the PC do not appear on the PCI bus and should be detected and started by just knowing that they ought to be there because it's a PC. Most ARM systems do not come with a PCI bus and have all their devices on addresses that need to be known ahead of time, but are different between platforms. A popular solution is to use DeviceTree to encode this information and that is certainly something that should be explored for Loko.

The major hardware interaction points are:

1. Scanning and configuring the bus; detecting new and removed devices.
2. Setting up access to the device.
3. Interfacing with the device through its registers, channels, etc.
4. Allowing the device to write to system memory.
5. Waiting on interrupts from the device.

The way these things are done depends on the bus. Further documentation is needed. For now, please consult the source code or ask.

6.2.3 Future directions for drivers

There is an interesting thing that can be done with drivers for PCI devices when `eval` uses online compilation. PCI devices can appear anywhere in memory and sometimes even anywhere in I/O space. Register access can look like this:

```
(define (driver·pci·uhci dev controller)
  ;; The UHCI registers are mapped to the location in BAR4
  (let ((bar (vector-ref (pcidev-BARs dev) 4)))
    ;; Disable keyboard and mouse legacy support
    (pci-put-u16 dev #xC0 #x0000)
    (driver·uhci (if (pcibar-i/o? bar) 'i/o 'mem)
                 (pcibar-base bar)
                 (pcibar-size bar)
                 (pcidev-irq dev)
                 controller)))

(define (driver·uhci reg-type reg-base reg-size irq controller)
  ;; Access to the device registers (independent of i/o vs mem)
  (define (reg-u8-ref offset)
    (assert (fx<? -1 offset reg-size))
    (case reg-type
      ((i/o) (get-i/o-u8 (fx+ reg-base offset)))
      ((mem) (get-mem-u8 (fx+ reg-base offset)))
      (else (assert #f))))
  ...)
```

It would be interesting if `driver·pci·uhci` used `eval` to compile a specialized version of the driver where `reg-u8-ref` (etc.) had been inlined by cp0. After compilation, each `reg-u8-ref` call would be a single instruction. Specializing, compiling and starting the driver can be as simple as this:

```
(let ((driver·uhci
       (eval `(lambda (controller)
                (driver-source·uhci ,reg-type ,reg-base
                                    ,reg-size ,irq
                                    controller))
             (apply environment driver-environment·uhci))))
  (driver·uhci controller))
```

In principle this kind of code would work even today, but the driver would be slowed down because `eval` is slow.

The same principle can be applied to embedded systems that use DeviceTree. If a static DeviceTree is used then this could even be done as part of the build process. If a dynamic DeviceTree is used (to allow the same kernel to run on different ARM platforms) then boot time may become an issue. But then drivers could be designed to initially use a non-specialized driver, call `eval` asynchronously, and tell the running driver to switch to the specialized driver when `eval` has returned.

6.3 Interrupt handling

Loko in essence treats device drivers as communicating sequential processes. IRQs from devices are treated as primitive messages from the device to the driver. Requests to configure the device or transfer data between the device and the rest of the system is also done with messages. Device drivers are not fundamentally different from other Scheme programs.

6.3.1 Historical background

This section uses the term *interrupt* to denote an interruption in the processor's normal execution sequence; *IRQ* to denote an interrupt from a hardware device and *trap* to denote an interrupt from the processor itself. There are more types of interrupts, but they are not discussed here.

IRQs are handled differently from how they are handled in normal kernels. An anecdote from The Rise of Worse is Better (`https://www.dreamsongs.com/RiseOfWorseIsBetter.html`) by Richard P. Gabriel is relevant here:

> Two famous people, one from MIT and another from Berkeley (but working on Unix) once met to discuss operating system issues. The person from MIT was knowledgeable about ITS (the MIT AI Lab operating system) and had been reading the Unix sources. He was interested in how Unix solved the PC loser-ing problem. The PC loser-ing problem occurs when a user program invokes a system routine to perform a lengthy operation that might have significant state, such as IO buffers. If an interrupt occurs during the operation, the state of the user program must be saved. Because the invocation of the system routine is usually a single instruction, the PC of the user program does not adequately capture the state of the process. The system routine must either back out or press forward. The right thing is to back out and restore the user program PC to the instruction that invoked the system routine so that resumption of the user program after the interrupt, for example, re-enters the system routine. It is called PC loser-ing because the PC is being coerced into loser mode, where loser is the affectionate name for user at MIT.
>
> The MIT guy did not see any code that handled this case and asked the New Jersey guy how the problem was handled. The New Jersey guy said that the Unix folks were aware of the problem, but the solution was for the system routine to always finish, but sometimes an error code would be returned that signaled that the system routine had failed to complete its action. A correct user program, then, had to check the error code to determine whether to simply

try the system routine again. The MIT guy did not like this solution because it was not the right thing.

From the New Jersey guy we get the `errno` value `-EINTR`, so he was clearly the more successful one. The MIT guy's approach would have been to carefully design syscalls so that they keep their full state in a structure or in the arguments to the syscall, which is kind of like doing a very manual and tedious `call/cc` when an interrupt arrives. And nobody has time for that.

But when the New Jersey guy feels like even `-EINTR` is too difficult to manage, we get *uninterruptible sleep*. This is a situation where programs aren't running, but they can't be killed either. This happens when a program is blocked in a syscall and is holding an unknown number of locks and other state in the kernel. Maybe it was reading from a file, but something got wedged and stopped responding. Instead of an error code, the process is in uninterruptible sleep. This slowly creeps from program to program as other parties try to communicate with the frozen process.

So Loko takes a different approach to all this.

6.3.2 An experimental approach to IRQs

IRQs are generated by hardware devices when they want the attention of the processor. An example is a UART (serial port controller) that has just received a byte. It will generate an IRQ to get the driver to read the byte from its buffer.

The normal idea of how to handle an IRQ is basically as follows: install the interrupt service routine (ISR) that comes as part of the driver, reset the device to its normal state, then enable the IRQ in the interrupt controller. An ISR is a special piece of code that must be prepared to run at potentially any time (except when interrupts are disabled). Usually there is a priority order on IRQs so that they can in turn be interrupted by higher-priority IRQs. Either way, when an IRQ arrives the driver quickly services the hardware. In the UART example it would read a byte from the UART and place it in a software controlled buffer.

In this way of doing things, there are unfortunately severe restrictions on what can be done in an ISR. An ISR runs in *interrupt context* where many usual kernel services are simply unavailable. Anything that would block the program is usually unavailable and access to memory is limited. Arranging things so that arbitrary Scheme code can run in interrupt context is difficult.

For this reason, Loko does not run driver code in ISRs. The ISRs are instead minimal pieces of code that cooperate with the process scheduler to make the driver's process runnable. An IRQ is sent as a message to the process and it handles it at its leisure.

This approach is somewhat experimental, and may have some problems with latency in legacy hardware such as UARTs, but modern devices do bus mastering DMA and are generally not sensitive to interrupt servicing latency. Bus mastering DMA means that the device has access to the system's memory. Generally such a device cooperates with the driver to maintain queues in system memory that describes data transfers.

There are pros and cons to Loko's approach. The cons are that IRQs are handled with some latency, but this is usually not a problem for modern devices. The pros are that it makes driver code *much* easier to write and maintain. Since it eliminates many of the usual difficulties with writing drivers, it may even mean that most competent Schemers with access to the hardware programming manual can write device drivers. (Some difficulties still remain with manual memory management, but they are not that hard to deal with).

Even if latency is a potential problem, Loko's approach should be good for throughput. A driver can easily decide that data is coming in at such a high rate that it doesn't need to use interrupts, and switch over to periodic polling instead. Linux uses this technique in its networking stack under the cryptic name "New API" (NAPI).

Another benefit of Loko's approach is that the dilemma in the "worse is better" story is resolved. User programs never need to be given an equivalent of `-EINTR` and the programmer does not need to manually keep track of where they are in the handling of a system call.

6.3.3 Loko's use of traps

Loko offloads as much error checking as possible on built-in mechanisms in the hardware. Instead of using explicit type checks, it lets the hardware do the type checks (where possible).

Programming errors like `(/ 1 0)` are often signalled by the hardware in most programming environments. Loko takes this further and extends it to errors like `(car #f)`, `(vector-ref "foo" 0)` and `(1 + 2)`. Loko uses the processor's alignment checking feature to trap wrong uses of pairs, procedures, strings, vectors and bytevectors. You can read more about this in Faster Dynamic Type Checks (`https://weinholt.se/articles/alignment-check/`).

6.3.4 Interrupts on bare hardware AMD64

The interrupt handlers are in `(loko arch amd64 pc-interrupts)` and are written in assembly.

There are two fundamentally different types of interrupts that both go under the name interrupt and that use similar mechanisms, but have very different sources.

6.3.4.1 Traps

Traps are interrupts that are triggered by some classes of errors in the running program. For these interrupts the interrupt handlers cause the program to invoke an error handler written in Scheme. No care is taken to preserve the program's current stack frame or current register values, which means that some useful debugging information is lost. While that is unfortunate, and should be fixed, it is still semantically correct since these errors are all categorized as `&serious`.

Traps are handled by entries in the interrupt descriptor table (IDT). The IDT controls whether the processor should change privilege levels, which address it should jump to, which stack it should use and whether interrupts should be automatically disabled or not.

The processor pushes some of the program state on the stack and gives control to the handler. For traps, the handlers identify the cause of the trap and decide which library function should take control over the program. It then executes a tail-call to that function. These functions live in `(loko arch amd64 lib-traps)` and are responsible for calling the error handler, which is written in Scheme.

6.3.4.2 IRQs

The IRQ handling in Loko relies on the processor's *interrupt stack table* (IST) and the interrupt controller's *specific end of interrupt* (SEOI) and *special mask mode*. On AMD64 systems the SEOI feature is available in the legacy PIC and the APIC (except in early versions). Interrupt priorities (nesting) are not meant to be used.

Interrupt masking is a very important part of Loko's IRQ handling. There are three places where interrupts can be masked: the device itself, the interrupt controller and the processor. The driver configures the device to generate interrupts in a way that suites the driver. The interrupt controller is responsible for delivering interrupts to the processor and can be asked by the processor to mask an interrupt. The interrupt controller also keeps track of which interrupts are being serviced and temporarily masks them until they are acknowledged. Finally, the processor can also mask interrupts with the '`RFLAGS.IF`' bit in the flags register. When '`RFLAGS.IF`' is clear, no interrupts are delivered.

Loko's IRQ handling system relies on masking interrupts with the '`RFLAGS.IF`' bit and delaying acknowledgement. '`RFLAGS.IF`' is used to mask interrupts while the scheduler is running. When inside the scheduler, interrupts can only happen in one very controlled circumstance: in the `sys_hlt` syscall. This is used when no processes are runnable and it lets the processor save energy. The `sys_hlt` syscall returns when an IRQ has been delivered to the processor. The scheduler then finds the driver process that has the IRQ registered, enqueues it as a message and makes the process runnable.

Normal processes always run with interrupts unmasked, except maybe for some brief and tricky moments during task switching. When an interrupt arrives, the processor uses the IST to switch to a different stack. This is necessary to avoid messing up the process's Scheme stack. All registers are saved in the process control block so that the process can be resumed later. The IRQ handler resumes the scheduler and lets it know what happened.

When the process resumes it will have an IRQ number in its message queue. It is up to the process when it wants to dequeue this message, but usually a driver process will be waiting for an IRQ to arrive. Whenever a process calls the scheduler it can also receive a pending message. The two cases are then that a process either becomes runnable due to an IRQ and that a process is already runnable and will see the IRQ later.

Processes that are notified about IRQs will handle them, doing whatever task is needed to service the IRQ in the hardware, and afterwards tell the scheduler to acknowledge the IRQ. The scheduler then sends an instruction to the interrupt controller to acknowledge that specific interrupt. At this point the device may again choose to generate an interrupt when the conditions are right.

There are a few classes of programming errors when it comes to IRQ code in drivers. The driver may decide that it has handled an IRQ and sends an acknowledgement. However, if it missed an interrupt reason, the device may generate the same IRQ again, immediately after acknowledgement. This slows down the system with unnecessary work in the driver.

The opposite can also happen if a driver does not properly handle all the interrupt reasons. The symptom can be that an IRQ arrives, is seemingly handled by the driver, but then no more IRQs ever arrive and the device seems to have frozen. Which class of error is more likely depends on if the device uses edge-triggered or level-triggered interrupts. A way to check if this has happened is to add a timeout when waiting for interrupts.

6.3.4.3 Observed bad IRQ behavior

Loko's handling of IRQs is experimental, as mentioned. These problems have been observed so far:

- There are sometimes bytes lost on the PC UARTs.
- Wrong hardware implementations of the i8042 controller (the PS/2 bus chip) seem to assume that IRQ 1 always interrupts the IRQ 12 handler. This results in keyboard data being read as mouse data.
- The PIT's IRQ 0 does not reach the CPU if the scheduler is running. The PIT is not used normally, so this is not a problem.

The UART and i8042 may need special interrupt handlers. But they are legacy devices and much amd64 hardware doesn't even have them.

6.3.5 Interrupts on Linux AMD64

The signal handlers are in `(loko arch amd64 linux-start)`.

Signals under Linux are similar to interrupts. Just like with interrupts, some signals are traps ('`BUS`', '`SEGV`', '`FPE`', '`ILL`', etc). Linux places the processor's trap number in the sigcontext of the signal handler, which makes it easy to handle them identically to the way traps are handled on bare hardware.

Other signals are from external sources and the program is innocent, so the current stack frame must be preserved. On bare hardware this is accomplished by the IST and the equivalent mechanism on Linux is called `sigaltstack`(2).

The normal userspace ABI, documented in System V Application Binary Interface AMD64 Architecture Processor Supplement (`https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI`), is not followed. Normally interrupts would be delivered on the same stack as the program is currently using, but that would mess up Scheme code due to the way Loko manages stack frames. The "red zone" concept does not exist in Loko.

Index