A high level simulator for Gunk*

Stephen Adams
adams@ai.mit.edu

September 30, 1997

Warning: this document is in revision to incorporate the new thread features. Some of the old material requires minor changes.

Contents

1 Introduction

This note describes HLSIM, how to use it, and some details about its implementation. HLSIM was produced as part of the Amorphous computing project [1].

HLSIM is a simulator for an ensemble of 'gunk' processors which allows programs to be written in (nearly) R^4RS Scheme [3]. The gunk processor is distinguished by having no input or output other than a short-range radio-like communications device, and possibly some local physical sensors and actuators.

The need for HLSIM arises out of the desire to run several thousand of copies of the top level program within a single MIT Scheme development environment, and to run the programs at a fine grained level of interleaving.

2 Using HLSIM: a tutorial

Our example is a version of the *clubs* algorithm. The idea is to organise gunk into small cooperative groups, or 'clubs' that can communicate efficiently with one another. We will describe the algorithm and then how to run the tutorial program which implements the algorithm.

2.1 The Clubs algorithm

The Clubs algorithm organises gunk into clubs each with a single leader. This can then be used as a basis for higher level partitioning of the gunk processors. The size of the club is the communication radius of the leader, so the leader can talk to all members of its club. A club should contain only one leader, but otherwise clubs may overlap. This gives us an abstraction of a *graph* with *nodes* and *edges*, where edges are processors in two clubs and nodes are processors in three or more clubs. Another nice feature is that based on the geometry of the surface and the communication geometry we can give upper bounds on the number of clubs as well as the maximum degree of the graph.

The algorithm (figure 1) works as follows

```
1.  (define (club-event-loop)
2.    (select
3.      (global-timeout  'done)
4.      ((and still-competing? leader-timeout)
5.       (event.clear! leader-timeout)
6.       (color-me "red")
7.       (set! still-competing? #F)
8.       (broadcast id)                     ; a LEADER message
9.       (club-event-loop))
10.     (primitive-message-event
11.      => (lambda (message)
12.           (event.clear! primitive-message-event)
13.           (if (eq? message 'collision)
14.               (set! collisions (+ collisions 1))
15.               (begin
16.                 (set! still-competing? #F)
17.                 (set! clubs-i-belong-to
18.                       (cons message clubs-i-belong-to))
19.                 (color-me (length clubs-i-belong-to))))
20.           (club-event-loop)))))
```

Figure 1: Club algorithm main loop

1. The algorithm runs for a fixed amount of time, *global-timeout*.
2. Each processor chooses a random duration, *leader-timeout*, which is less than the global timeout. Each processor waits until either the leader timeout time has passed or a message has arrived.
3. If the leader timeout duration passes without a message arriving (line 4), the processor declares itself as a club leader to all its neighbors.
4. If a processor hears a message (line 10) before becoming a leader it becomes a club member and spends the rest of its time listening for other leaders.
5. When all the global timeouts have occured, every processor is either member of a club or a leader.

Caveats — by choosing a large *range* the probability of conflicts can be reduced to minimum. Otherwise, there needs to be conflict detection and resolution, which is not difficult but then the end time is also probabilistic.

2.2 Running the Clubs algorithm

Hlsim runs under MIT-Scheme. Hlsim requires the compiler to be loaded, so make sure that you run Scheme with a band that contains the compiler. If you are running Scheme under emacs, use the `-compiler` command line option when you run scheme (*C-u M-x run-scheme*). If you are using Edwin, use

Normal compilation (`cf` = `sf` followed by `cbf`)

$$\textit{file.scm} \xrightarrow{\texttt{sf}} \textit{file.bin} \xrightarrow{\texttt{cbf}} \textit{file.com}$$

$$\textit{file.scm} \xrightarrow{\texttt{cf}} \textit{file.com}$$

Gunk compilation

$$\textit{file.scm} \xrightarrow{\texttt{cps}} \textit{file.bin} \xrightarrow{\texttt{cbf}} \textit{file.com}$$

Figure 2: Compilation in MIT Scheme

the command line option `-band all.com` (on Windows NT) or `-band c+e.com` (unix).

Code for this tutorial example can be found in the *examples.zip* file on the 6.966 web page and in the file */home/gunk/hlsim/examples/Tutorial*. One way of running the tutorial example is to insert *Tutorial* into your Scheme interaction buffer and use *M-z* to send the expressions to the Scheme REPL.

HLSIM is available as a load option on the Switzerland HP Snakes.

```
(load-option 'hlsim)
```

If you are running HLSIM on another machine where `load-option` is not set up, you can always load HLSIM something like one of these:

```
;; AI Lab linux machine:
(load "/home/gunk/hlsim/stable-linux/loader")
;; Switzerland NT machine:
(load "//martigny/gunk/hlsim/stable-linux/loader")
;; From a local copy on an NT machine:
(load "d:/gunk/hlsim/stable/loader")
```

Make your own copy the example files, which contain a Gunk program (*club5.scm*) and some utilities for keeping a graphical display of the simulation's prograss (*club5d.scm*).

```
(copy-file "/gunk/hlsim/examples/club5.scm" "club5.scm")
(copy-file "/gunk/hlsim/examples/club5d.scm" "club5d.scm")
```

The Scheme code for the gunk processor cannot be executed directly. The resaons for this are explained later. The program first must be compiled into a special *continuation passing style* (CPS). This is achieved as follows:

```
(cps "club5")
```

The `cps` procedure reads in the *.scm* file and outputs a *.bin* file. In this respect, `cps` is a replacement for `sf`. `sf` processes declarations and macros, so `cps` first uses `sf`, and then transforms the result into CPS.

The next two steps are optional. The club files may be compiled for faster execution. *Compiled programs are harder to debug. It is strongly recommended that you don't compile the programs until you have got most of the bugs out.* Note that we use `cbf` to compile the *.bin* file, since `cf` would do the wrong thing by generating a new *.bin* file.

```
(cbf "club5")     ; compile gunk file
(cf "club5d")     ; compile regular scheme file
```

The relationships between the compilation commands are illustrated in figure 2.

We can now load some some display utilities and a procedure to help us make a simulation, since they are regular Scheme procedures:

```
(load "club5d")
```

The next stage is to construct a simulation object. A simulation object contains all the information required to run the simulation.

The procedure `make-sim/1` from *club5d* (see appendix for listing) is called to construct a simulation where 1000 processors are uniformly distributed in a unit square. The processors have an effective communications radius of 0.05 units:

```
(define sim (make-sim/1 1000 .05))
```

This example includes the use of a Scheme Graphics window to display the processors as color-coded dots. If you want to use this feature, then enter:

```
(simulation.display! sim #T)
```

An empty Scheme Graphics window will appear.

The penultimate step is to load the gunk program into the simulation. We load two files. *init*[1] contains a few useful procedures, such as the *color-me* procedure which calls the `%color-me` procedure loaded from *club5d*.

Load the gunk program:

```
(simulation.load sim 'init "club5")
```

Finally, we can run the simulation:

```
(simulation.run sim)
```

When the simulation has finished (or at any point if you stop it by breaking in `C-c C-b`) you can gather information from the simulation. For example, you might be interested to know how many collisions[2] each processor detected:

[1] Note that the *init* file is a standard file. Since its pathname depends on where HLSIM is stored, the *symbol* `'init` is used to name the file. This works only for a few known files. Other files are named by the usual MIT Scheme string or pathname names

[2] Sometimes we can tell a message arrived but has been garbled, probably because it 'collided' with another message [4].

```
(define cols (simulation.eval sim 'collisions))
```

The *simulation.eval* procedure evaluates its second argument in each processor, and returns a vector of results, so

```
(vector-ref cols 42)
```

will tell us how many collisions the 43rd processor counted (vector indexes are 0-based).

The expression to evaluate is an arbitrary Scheme expression, for example, we could use

```
(reduce max 0
  (vector->list
     (simulation.eval sim '(length clubs-i-belong-to))))
```

to calculate the maximum number of clubs that any processor belongs to, and check it against a theoretical result.

3 Procedures

`%make-simulation` *procedure*

Create a new simulation. The simulation requires its `x`, `y`, `z` and `neighbours` fields to be set and some processors to be created before it is useful. Returns the new simulation.

`simulation.x` *simulation* *procedure*
`simulation.y` *simulation* *procedure*
`simulation.z` *simulation* *procedure*

Returns the vector of `x` (`y` or `z`) coordinates of the *simulation*'s processors.

`set-simulation.x!` *simulation x* *procedure*
`set-simulation.y!` *simulation y* *procedure*
`set-simulation.z!` *simulation z* *procedure*

Set the coordinates of the *X*, *y* and *z* must be vectors of floating point numbers and all of the same length. Typically used at the time of the simulation's construction

`simulation.load` *simulation file file ...* *procedure*

Load the *file*s into the simulation. The files must have been processed by `cps` (or specially written in CPS style.) The file arguments must be either strings or pathnames, or one of a small set of special symbols.

Currently, there is only one special symbol, `'init`, which means that the standard library of low-level procedures should be loaded. Always put `'init` first.

`simulation.run` *simulation* *procedure*

Start or resume the simulation.

3.1 Customizing simulations

A simulation object may have additional slots and operations added to it at any time.

`simulation.set!` *simulation name value* *procedure*

Assign *value* to the slot with name *name*. If the slot does not exist it is created.

`simulation.get` *simulation name* *procedure*

Returns the value of the slot with name *name*. If there is no slot with name *name* an eror is signalled.

`simulation.define` *simulation name value* *procedure*

A synonym for `simulation.set!`.

`simulation.operation` *simulation name argument argument ...* *procedure*

Gets a procedure from the slot with name *name*. The procedure is applied to *simulation* and *arguments*. Example:

```
(define sim ...)
(simulation.define sim 'print
  (lambda (simulation port)
     ...  stuff to do with SIMULATION))

(simulation.operation sim 'print (current-output-port))
```

An alternative style of programming is to have the operations *closed over* the simulation:

```
(define sim ...)
(simulation.define sim 'print
  (lambda (port)
     ...  stuff to do with SIMULATION))

((simulation.get sim 'print) (current-output-port))
```

`simulation.eval` *simulation expression* *procedure*

Evaluate *expression* on each Gunk processor in the *simulation* and return a vector of the results, where the index of the result corresponds to the index of the processor. The *expression* can be either an s-expression, a procedure of no arguments, or a pre-cps-converted procedure.

4 Multithreading

Gunk Scheme programs may be multithreaded. A programs starts off with one thread and may create more threads. Threads synchronise by *events*. Events have two states: unsignalled and signalled. Signalled events are associated with a value. A thread may wait for an event to be signalled by calling the `wait` procedure. The call to *wait* returns the value associated with the event when it becomes signalled. The thread is suspended until the event becomes signalled. If the event has already been signalled at the time of the call to *wait* then the call returns the associated value immediately.

Each thread is associated with an event object that becomes signalled when the thread terminates. In normal execution the thread termination event is associated with the value of the last expression.

Mutual exclusion is provided by *serialization*.

`parallel` *form form ...* *syntax*

Creates and starts a thread for each *form*, and waits for all those threads to terminate.

`select (`*event form ...*`) ...` *syntax*

The `select` form allows a nondeterministic choice between several events. The syntax is modelled after `cond`.

select-form	::=	(`select` clause clause ...)
clause	::=	(guard form ...)
clause	::=	(guard `=>` procedure)
clause	::=	(`else` form ...)

A *guard* is an expression that evaluates to an event object, a thread, or `#F`. All of the guard expressions are evaluated. Clauses with guard expressions evaluating to `#F` are eliminated from the choice. One of the remaining clauses is chosen when its event (or thread termination event) is signalled. If some clause has an event that is already in the signalled state, that clause may be chosen immediately. In the `=>` form, *procedure* is called with the value associated with the event.

If all guard expressions evaluate to `#F` then there is no event to wait for, so execution must proceed. If there is an else-clause then the else-clause forms are executed, otherwise the whole select-from finishes and has an unspecified result[3]. There may be at most one else-clause.

The following example will print out a message if one has arrived or arrives within 1000 time units:

```
(define primitive-message-event (make-event))
```

[3]Perhaps the default should be to signal an error

```
(select
  (primitive-message-event
   => (lambda (message)
        (event.clear! primitive-message-event)
        (write-line message)))
  ((make-timeout-event 1000)
   'timed-out!))
```

`make-thread` *thunk* — *procedure*

Create a new thread, but do not start the thread. Returns the thread object.

`start-thread` *thread* — *procedure*

Start a new thread. Returns *thread*.

`make&start-thread` *thunk* — *procedure*

Create a thread and start it. Returns the thread object. Could have been defined as:

```
(define (make&start-thread thunk)
  (start-thread (make-thread thunk)))
```

`make-event` — *procedure*

Create a new event. The event is in the unsignalled state.

`make-timeout-event` *duration* — *procedure*

Create an event. The event will become signalled in approximately *duration* clock ticks from the event creation time. *Duration* should be an exact integer.

Note: timeout events may be shared by several threads and remain in the system until the timeout occurs. If you want loop until a certain distant time waiting for messages, create the timeout event outside the loop rather than for each iteration, otherwise all the (essentially equivalent) timeout events might fill the Scheme heap and clog up the simulator.

`event.signal!` *event value* — *procedure*

Put *event* in the signalled state and make *value* its associated value. Any threads waiting on *event* become unblocked. The thread calling *event.signal!* does not block.

`event.clear!` *event* — *procedure*

Return *event* to the unsignalled state.

`primitive-message-event` *global variable*

If this global variable is bound to an event object, the event is signalled with every incoming message.

```
(define primitive-message-event (make-event))

(let loop ()
  (let ((message (wait primitive-message-event)))
    (event.clear! primitive-message-event)
    (write-line message)))
```

`wait` *event* *procedure*

Wait for an event to become signalled. Returns the value associated with the signalled event. The thread is suspended until the event becomes signalled. If the event has already been signalled at the time of the call to `wait` then the call returns the associated value immediately. As a convenience, *event* may be a thread, in which case the thread's termination event is used.

`wait-multiple` *event event ...* *procedure*

Wait for one of the *event*s to become signalled. Returns a pair of the form `(` *index* `.` *value* `)`, where *index*, which is zero-based, identifies the signalled event, and *value* is the value associated with that event. The thread is suspended until at least one event becomes signalled. If an event has already been signalled at the time of the call to `wait-multiple` then the call returns immediately. As a convenience, any *event* may be a thread, in which case the thread's termination event is used.

It is often easier to use the `select` special form than calling `wait-multiple` directly.

`make-serializer` *procedure*

Returns a new serializer. A serializer represents a set of procedures that make up a region of mutual exclusion, or a *critial section*. When a thread is execting in the critial section, no other thread may enter the critial section. Other threads attempting to call one of the procedures in the set are suspended until the first thread exits the critial section.

`make-serializer` returns a new serializer which is a higher order procedure. The serializer is applied to a procedure to incorporate it in critial section. The value returned is a procedure with the same behaviour, but the returned procedure also enforces the critical section.

To illustrate the problem that serializers solve, consider the following code. At the end, `count` could be -1, 0, 1 or 2, depending on the ordering of the three reads of `count` and the three writes.

```
(define count 0)
(define (add x y) (+ x y))
(define (incr)
  (set! count (add count 1)))
(define (decr)
  (set! count (add count -1)))
(parallel (incr) (decr) (incr))
```

At the end, `count` will be 1:

```
(define count 0)
(define (add x y) (+ x y))
(define (%incr)
  (set! count (add count 1)))
(define serialized (make-serializer))
(define incr (serialized %incr))
(define decr
  (serialized
    (lambda ()
      (set! count (add count -11)))
(parallel (incr) (decr) (incr))
```

Note that other threads are only prevented from calling `incr` and `decr`, and not prevented from calling `add` or `%incr` directly.

5 Sensors and actuators

Processors in a simulation may have 'hardware' attached to them to enable them to measure properties of their environment and to effect changes to their environment.

5.1 Defining sensors and actuators

`make-io-device` *initializer action-procedure* *procedure*

Creates a sensor or actuators I/O device. A device has two aspects: initialization and I/O.

The *initializer* procedure is called at the beginning of the simulation. It has the following signature:

```
(lambda (simulation device) ...)
```

The *action-procedure* is called each time a processor performs a `read-sensor` or `apply-actuator` operation. It has the signature

```
(lambda (simulation device index time) ...)
```

where *index* is the index of the processor performing the operation and *time* is the time of the operation.

`simulation.define-device` *name device* *procedure*

Associate *name* with a device.

`make-static-projected-PNM-colour-sensor` *image-file* *procedure*

Makes an input device that senses RGB colour. The PPM image in *image-file* is projected onto a unit square. The sensor has a constant stimulus which is taken from the image. Reading the sensor returns a RGB value.

`make-static-projected-PNM-intensity-sensor` *image-file* *procedure*

Makes an input device that senses light intensity. The PPM image in *image-file* is projected onto a unit square. The sensor has a constant stimulus which is taken from the image. Reading the sensor returns an intensity value between 0.0 (black) and 1.0 (white).

5.2 Using sensors and actuators

The following two procedures are availaible to Gunk programs for controlling the processor's sensors and actuators.

`read-sensor` *name argument argument ...* *procedure*

`apply-actuator` *name argument argument ...* *procedure*

5.3 Example

Gunk:

```
(define colour (read-sensor 'LIGHT-SENSOR))
```

Setup:

```
(simulation.define-device sim 'LIGHT-SENSOR
  (make-static-projected-pnm-colour-sensor "picture.ppm"))
```

6 Tips and useful procedures

6.1 Placing processors

`point-neighbours` *x y z d* *procedure*

Takes a set of points and returns an undirected graph of those points which are near each other. The points are represented by three vectors, x, y and z, which must all be the same length, n. The ith point is at position *(x(i),y(i),z(i))*.

The value returned is a vector v of vectors of exact nonnegative integers. The vectors of integers *v(i)* represent the set of points near the ith point. If the point *(x(i),y(i),z(i))* is within distance d of *(x(j),y(j),z(j))*, then i will appear in *v(j)* and j will appear in *v(i)*. A point is never in its own set of neighbours.

`cyclic-point-neighbours` *x y z d* *procedure*

This procedure is like `point-neighbours`, but with cyclic boundary conditions on a unit cube.

`asymmetric-point-neighbours` *x y z d1 d2* *procedure*

Takes a set of points and returns an directed graph of those points which are near each other. This procedure works like `point-neighbours` with the following difference. Points within distance *d1* are always symmetrically neighbours. Points outside distance *d2* are never neighbours. Points between distance *d1* and distance *d2* are chosen as neighbours randomly with probability of 0.5.

The *d1* parameter may be 0.0.

```
(asymmetric-point-neighbours x y z 0. 0.2)
```

`asymmetric-cyclic-point-neighbours` *x y z d1 d2* *procedure*

This procedure is like `asymmetric-point-neighbours`, but with cyclic boundary conditions on a unit cube.

`random-2d-points` *n* *procedure*

Create *N* points at random positions in the unit square. Returns two vectors of floating poit numbers, so use `random-2d-points` like this:

```
(call-with-values
    (lambda () (random-2d-points 1000))
  (lambda (xs ys)
     (let ((sim (%make-simulation xs ys (make-vector 1000 0.0))))
       ...)))
```

6.2 Repeatable experiments

It can be really difficult to debug programs that run concurrently on many processors.

Unlike a real amorphous computer, HLSIM is *deterministic*. Some decisions are *arbitrary*, for example, the placement of a processor might be decided by using pseudo-random numbers. This means that it is possible to run the 'same' simulation by using the same set of arbitrary decisions.

You can make a random state and save it like this:

```
(fasdump (make-random-state #T) "1.rnd")
```

Later you can initialize the default random number generator as follows. Note that you should always use `make-random-state` to copy random state before assigning it to `*random-state*`.

```
(begin
 (set! *random-state* (make-random-state (fasload "1.rnd")))
 unspecific)

(define sim (make-sim/1 1000 0.05))
```

6.3 PNM images

There are a handful of procedures for using *portable anymap* (PNM) images [6]. Images are useful for providing stimuli to processors with light-sensors.

These procedures are restricted to binary or *raw* anymaps, and the maxpixel values of a graymap (PGM) or pixmap (PPM, colour) must be 255. The value of a pixel depends on the type of image. Bitmaps (PBMs) have pixel value of 0 (black) and 1 (white). Graymaps have intensity levels between 0 (black) and 255 (white). Colour images have integer pixel values representing a colour as a RGB triple. The encoding expressed as a hexadecimal number is `#x`*rrggbb*, where *rr* represents the 8 bits for the red component (between 0 and 255) etc.

`read-pnm` *pathname* — *procedure*

Read the image from the file named by *pathname*. The type of anymap returned depends on the file contents.

`pnm.write` *pnm pathname* — *procedure*

Write the image *pnm* to the file named by *pathname*.

`make-initialized-pbm` *width height procedure* — *procedure*
`make-initialized-pgm` *width height procedure* — *procedure*
`make-initialized-ppm` *width height procedure* — *procedure*

Make a new image of *width* by *height* pixels. The pixel values are initialized from *procedure*.

`make-pbm` *width height* — *procedure*
`make-pgm` *width height* — *procedure*
`make-ppm` *width height* — *procedure*

Make a blank image of *width* by *height* pixels. All pixels are initially zero.

`pnm?` *object* — *procedure*
`pbm?` *object* — *procedure*
`pgm?` *object* — *procedure*
`ppm?` *object* — *procedure*

Type predicates. Note that `pbm?`, `pgm?` and `ppm?` each imply `pnm?`.

`pnm.get` *pnm x y* *procedure*

Retrieve the value of the pixel at (x,y). The range of the value depends on the kind of image as described above.

`pnm.set!` *pnm x y value* *procedure*

Set the pixel at (x,y). *Value* must be in the right range for the kind of PNM image.

`pnm.get-brightness` *pnm x y* *procedure*

Returns a pixel value as floating point values between 0.0 and 1.0 (inclusive). Works for any kind of PNM image.

`pnm.get-rgb` *pnm x y* *procedure*

Returns a pixel value as a RGB integer as described above. Works for any kind of PNM image.

7 Gunk Scheme vs. ordinary Scheme

Global environment. Each gunk processor runs the program in its own top level environment. Thus each processor potentially has different values for its global variables.

As a convenience feature, variables that are not defined by the gunk program are 'defaulted' from the user's environment. This allows gunk programs to use standard procedures like `+`, `length` and `equal?`.

However, it should be noted that defaulted procedures are assumed to behave like primitives, that is, they take very little time to execute. For some simulations this might be an important consideration, in which case the procedure can be added to the gunk code for more rigourous simulation. A warning is printed the first time a global value is used.

The defaulting process does not work properly with higher order procedure like `map` and `for-each` because the procedure that you might pass out has additional arguments used by the simulator. You have to write your own version of these procedures[4].

Unbound and unassigned variables. *I am thinking of changing this.* Currently there is no such thing as an unbound or unassigned gunk variable. A new variable receives a default value which is a symbol of the form `#[unassigned` *name*`]`. Adding unassigned reference traps will slow things down slightly but it is probably worth it for debugging.

[4] *Sometime I'll write a little library of the common ones*

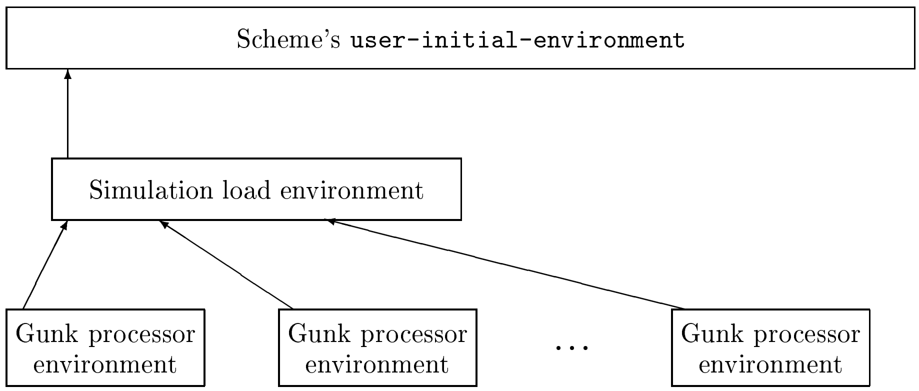

Figure 3: Environments

Irritating warnings The transformation (by `cps`) introduces a lot of small constant vectors for accessing variables in the gunk processor global environment. The compiler shares (coalesces) these constants and by default issues warnings like this:

```
;Warning: Coalescing two copies of constant object #(collisions #f #f)
```

To turn these warnings off:

```
(set! compiler:coalescing-constant-warnings? #F)
```

Another kind of compilation warning happens when the compiler discovers that the hairy CPS-transformed code is a little simpler than it first thought.

```
;Warning: non-trivial procedure becoming trivial #[liar:procedure 25 let-38]
```

There is no way to turn these warnings off.

8 Extending HLSIM

8.1 Parts of HLSIM

- cps converter
- init
- scheduler
- communications
- gunk program

Figure 3.

8.2 Standard communications

Properties:

- Messages take time to transmit.
- Messages may be lost completely, known to be garbled, or detected completely.
- Garbled messages are detected and represented as a message, the symbol `collision`.

`broadcast` *message* — *procedure*

The gunk processor initiates the broadcast of a *message*. The program continues while the message is transmitted. The message may be any scheme object, but it is best to stick to realistic data structures like lists, vectors, strings, numbers and symbols.

At the completion of the transmission, if the message was not corrupted, the receiving processor's `primitive-message-event` is signalled with the message. If the message was corrupted then the receiving processor might detect the collision (signalling `'collision`) or fail to detect the message completely (not signalling anything).

`message-delay` — *scheme variable*

The time units taken to transmit a message, an exact nonnegative integer. Initial value is 100, which is probably too small.

`carrier-sense?` — *procedure*

This predicate queries the receiver apparatus to determine if there is an incomming message in progress. It can be used to reduce collisions by checking just before a broadcast. There is no guarantee that if `(carrier-sense?)` returns `#T` that at some later point `primitive-message-event` will be signalled.

8.3 Efficiency trade-offs

A Gunk simulation is only a model of a clump of amorphous computers. As with any other model, trade-offs have been made between the faithfulness of the model and complexity and efficiency of the model. It is important to be aware of this fact.

HLSIM is capable of a reasonable degree of accuracy in its model of Scheme program execution. For 'ordinary' programs, a Gunk program will execute in a number of clock ticks that is roughly proportional to the time the original Scheme program would take. The degree of accuracy is probably similar in predictive power to using an interpreter to predict compiled code performance. You might predict "40 times faster than the interpreter" but some of the compiled code would be only slightly faster (e.g. input/output) and other parts might benefit a lot from compiler optimizations and be a several hundred times faster. However, the thinking of so-many-times faster is adequate for most purposes. The program, whether intepreted or compiled (or, run on different machines), is executing the same algorithm.

Problems can arise from two sources. These sources can be categorized into *timing dependent* and *non-linear timing*.

If the correctness of the program depends on the relative speed of different program segments then is it *timing dependent*. A program is also timing dependent if its correctness depends on the relative speed of the program to communications. The communications parameters can be adjusted to compensate for problems with program/communications timing problems.

Non-linear timing problems occur when the simulation accounts a program fragment as requiring an amount of time that is not plausably linearly related to the time it would take a 'real' program to perform the same operations. The main reason this might occur is that the simulator will use standard Scheme procedures if no Gunk version is defined. It assumes that the ordinary Scheme versions take a fixed, small amount of time.

Consider the following three procedures: `car`, `*` and `assq`. One might assume that `car` always completes within a fixed amount of time. Calling `*` is usually is pretty fast too, but there are exceptions, for example, when operating on very large exact integers. `Assq` is usually expected to take time proportional to the length of the list.

It is often reasonable to use the standard Scheme version of procedures like `assq` because the timing behaviour should not affect correctness. One might argue this is reasonable usage because the alist is always quite small, or that a hash table would turn the operation into a constant time operation.

For the purposes of speeding up the simulation, it is almost always faster to call the standard Scheme procedure.

8.4 Cheating

The standard initialization file (*init.scm*) contains some Gunk procedures that allow the processor's program to cheat. By cheating, I mean stepping outside the abstraction of each Gunk processor running isolated and clueless about its surroundings except by external stimulus.

`the-simulation` *procedure*

Return the current simulation object. Since Gunk programs can call ordinary Scheme procedures, it is possible (though not recommended) for the processor to rummage through the entire simulator's state and get information that is against the spirit of Gunk.

```
(declare (usual-integrations))

(define (color-me cont fuel store color)
  (cont (%color-me (store.simulation store) ; get simulation
                   (store.index store)      ; get processor index
                   color)                   ; pass on argument
        (fix:+ fuel 4)                  ; `cancel' the cost of calling COLOR-ME
        store))

;; This must me tha last line in the file, since the file must
;; contain a CPS converted expression:
(lambda (cont fuel store)
  (cont unspecific fuel store))
```

Figure 4: Writing a cheat procedure

`processor-number` *procedure*

Return the index of the current processor. While this can be used to wreak havoc with the Gunk abstraction, it is also very useful for tagging debugging output. For example, a Gunk program might print out a few of its variables like this:

```
(pp `(processor: ,(processor-number) foo: ,foo total: ,total))
```

8.5 Writing your own cheat procedures

Writing cheat procedures requires understanding of the `cps` transformation which is decribed in the appendix. For simple procedures is is possible to get by with extending the code in Figure 4.

As an example, we will show how the `color-me` procedure is defined. `color-me` needs to know which processor it should draw and where to draw it. Getting hold of the simulation and the index of the processor within the simulation is the tricky part. For that reason, we split `color-me` into two parts: one to get the simulatiuon object and processor index, and a helper procedure `%color-me` which does the real work and is written as an ordinary Scheme procedure (for an example see *club5d.scm* in the appendix.)

The file from Figure 4 is compiled as an ordinary Scheme file, but it is loaded into the simulation with `simulation.load`. Since it is loaded into the simulation load environment (Figure 3), the whole file must be written `cps` converted. This means that the very last expression in the file must be of the form:

```
(lambda (cont fuel store)
  ...
  (cont <file-value> fuel store))
```

The expression `(fix:+ fuel 4)` can be replaced by just `fuel`. Doing so will cause the Gunk call to `color-me` to take a normal amount of simulated processor time.

8.6 Adding an alternative communications model

Summary: you implement your own version of `%broadcast` (for an example, see *comms3.scm*). This is fairly difficult to do.

9 Bugs and problems

9.1 Known problems

9.1.1 Wrong number of arguments

Example error:

```
;The procedure car has been called with 4 arguments; it requires exactly 1 argument.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

The usual problem is passing a Gunk procedure out to an ordinary higher-order Scheme procedure, e.g. `map`. The simulator tells you when it decides that it must use an ordinary procedure. You can check the warnings to see if any of them are higher order.

There is one other case which is a bit unfortunate, but avoidable. Some primitive procedures in non-operator position are *integrated* because of the `usual-integrations` declaration. Currently HLSIM does not realize that the primitive procedure needs to be cps-converted too. It treats it like any other literal, e.g. `42`.

There are very few of these procedures which are useful like this, but `car` is one of them. Example Gunk program:

```
(define (make-point x y) (list x y))
(define point-x car)
```

When you call `point-x` you will get the error above. One fix is to η-convert:

```
(define (point-x pt) (car pt))
```

I could make `cps` do this transformation automatically in the case of `car` but it would cause different references to `car` to no longer be `eq?`.

Another fix is to define the data-structure (`point-x` etc.) as an ordinary Scheme program, i.e. in a separate file and load that file with `load` rather than `simulation.load`. I prefer the η-conversion fix.

9.2 Reporting bugs

There is an email address for reporting bugs with HLSIM.

Before reporting a problem, please try to narrow it down to a small, complete and repeatable example. Make your problem description as precise as you can. That way we have a chance or reproducing the problem, which we probably need to do in order to understand the problem, which we need to to in order to fix it.

The email address is: `bug-hlsim@martigny.ai.mit.edu`

A The transformer

HLSIM uses program transformation to convert the user's Scheme program into a form that can be executed efficiently in the simulation environment. The `cps` procedure (command) performs the transformation on a Scheme source file.

Questions:

"Why bother transforming the Scheme program?"

"Can't you just run Scheme directly?"

"Why don't you just write the communications primitives to context switch"

The transformation approach addresses all of the following requirements for the simulator.

- Programs should be expressed in a high level language, *e.g.* Scheme, and not be overly burdened with artifacts of the simulation mechanism (*e.g.* calls to `yield`, explicit construction of saved state).
- The ability to model time accurately.
- Modelling of asynchronous events (interrupts).
- Many (thousands) of programs running together. It is important that we can do simulations on ensembles large enough that bulk behaviour to appears.
- Execution efficiency. It is desirable for the simulation to run as fast as possible without sacrificing accuracy[5].

Conceptually, the gunk program is a program running on many machines. The simulator is a program running on one machine.

The transformer *compiler enforced cooperative multitasking*

Alternatives:

`call-with-current-continuation`

Interpreter — slower.

[5] How do we measure this?

A.1 Continuation passing style

One may partition the events in a computation into two sets: those that have already happened, and those that have yet to happen.

Continuation passing style is a style of writing programs where the events in the future, i.e. the rest of the computation, is passed as an explicit parameter. The value passed as this parameter is the *continuation*. A continuation is a procedure that takes the value of the current expression and computes the rest of the computation. Procedures do not return values; instead, they invoke the continuation with the result. If a program is fully CPS converted then there are no subproblem procedure calls. Every procedure call is tail call, and the program's 'control memory' is not stored in some invisible stack but explicity as the continuation.

A program can be converted to CPS relatively easily. This transformation is well understood as it is the basis for several optimizing compilers [2].

Consider the Fibonacci function, `fib`.

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

We CPS convert `fib`, adding k as the additional continuation parameter. We do not CPS convert the primitive procedure calls to `<`, `+` and `-`:

```
(define (fib k n)
  (if (< n 2)
      (k 1)
      (fib (lambda (fib-of-n-1)
             (fib (lambda (fib-of-n-2)
                    (k (+ fib-of-n-1 fib-of-n-2)))
                  (- n 2)))
           (- n 1))))
```

A.2 The engine transformation

We augment the CPS program to count operations. Later we will use the count to limit the number of operations that a program does. Instead of counting up, we count down, since by starting at some positive count, we can count down to zero, a fixed limit, and avoid having to pass a limit too.

```
(define (fib k fuel n)
  (if (< n 2)
      (k (- fuel 1) 1)
      (fib (lambda (fib-of-n-1 fuel)
             (fib (lambda (fib-of-n-2 fuel)
                    (k (+ fib-of-n-1 fib-of-n-2)
```

```
7.                          (- fuel 1)))
8.                     (- fuel 1)
9.                     (- n 2)))
10.             (- fuel 2)
11.             (- n 1))))
```

The counting of the operations is as follows. Notice that although `<` appears twice in the accounting, it will only be counted once since each count appears on a different conditional path.

line	operations counted (line)
3	`<` (2)
6	`+` (6)
7	`-` (9)
9	`<` (2), `-` (11)

The variable `fuel` is *single threaded* (also known as *linear*), which means that once a new value of *fuel* is computed, the old value is never re-used.

[5]

A.3 Explicit store

With many thousands of top level environments it is necessary to carefully manage the size of the top level environment. MIT Scheme's first class environments are too heavy-weight. There are several reasons for this, but the most important is that compiled programs are linked against an environment, which requires separate copies of the compiled code for each processor.

The transformed programs use an explicit store, similar to this:

```
(define (fib k fuel st n)
  (if (< n 2)
      (k (- fuel 1) st 1)
      ((global-ref st '#(fib #F #F))
       (lambda (fib-of-n-1 fuel st)
         ((global-ref st '#(fib #F #F))
          (lambda (fib-of-n-2 fuel st)
            (k (+ fib-of-n-1 fib-of-n-2) (- fuel 1) st))
          (- fuel 1)
          st
          (- n 2)))
       (- fuel 2)
       st
       (- n 1))))
```

The variable `st` is also linear.

By using an explicit store it is possible to factor the environment into a shared part and a per-processor part. Each global gunk variable is allocated to a slot in the store. The allocation is dynamic, but each gunk processor uses the same store slot for the same variable.

References

[1] `http://www-swiss.ai.mit.edu/s̃witz/amorphous`.

[2] David Kranz, ORBIT: An Optimizing Compiler For Scheme, PhD Thesis, 1988 Yale University.

[3] W. Clinger and J. Rees (editors). Revised^4 Report on the Algorithmic Language Scheme. Nov. 1991.

[4] Daniel Coore, Radhika Nagpal, Ron Weiss, Paradigms for Structure in and Amorphous Computer.

[5] M. Feeley. Polling efficiently on stock hardware. In *Proceedings of the 1993 ACM Conference on Function Programming Languages and Computer Architecture*, 1993.

[6] J. Poskanzer. See `pnm(5)` unix man page.

Files

club5.scm

```
(declare (usual-integrations))

(define id (random 1000000))

(define global-timeout (make-timeout-event 10000))
(define leader-timeout (make-timeout-event (random 5000)))
(define collisions 0)

(color-me "blue")

(define clubs-I-belong-to '())

(define still-competing? #T)

(define primitive-message-event (make-event))

(define (club-event-loop)
  (select
    (global-timeout  'done)
    ((and still-competing? leader-timeout)
     (event.clear! leader-timeout)
     (color-me "red")
     (set! still-competing? #F)
     (broadcast id)                    ; a LEADER message
     (club-event-loop))
    (primitive-message-event
     => (lambda (message)
          (event.clear! primitive-message-event)
          (if (eq? message 'collision)
              (set! collisions (+ collisions 1))
              (begin
                (set! still-competing? #F)
                (set! clubs-i-belong-to
                      (cons message clubs-i-belong-to))
                (color-me (length clubs-i-belong-to))))
          (club-event-loop)))))

(club-event-loop)

(list id clubs-i-belong-to collisions)
```

club5d.scm

```
(declare (usual-integrations))

(define (make-sim/1 n r)
  (let ((x (make-initialized-vector n (lambda (i) i (random 1.0))))
        (y (make-initialized-vector n (lambda (i) i (random 1.0))))
        (z (make-vector n 0.0)))
    (let ((neighbours (point-neighbours x y z r)))
      (let ((sim (%make-simulation)))
        (do ((i 0 (+ i 1))) ((= i n))
          (simulation.add-processor sim))
        (set-simulation.x! sim x)
        (set-simulation.y! sim y)
        (set-simulation.z! sim z)
        (set-simulation.neighbours! sim neighbours)
        sim))))


(define (simulation.display! sim flag)
  (define (make-display)
    (let ((g  (make-graphics-device)))
      (graphics-operation g 'set-foreground-color "white")
      (graphics-operation g 'set-background-color "black")
      (graphics-set-coordinate-limits g -.01 -.01 1.01 1.01)
      (graphics-clear g)
      g))
  (set-simulation.display!
   sim
   (cond ((not flag)  #F)
         ((graphics-device? flag) flag)
         (else (make-display)))))

(define (%color-me sim i color)
  (define d 0.007)
  (define colors '#("blue" "orange" "yellow" "green" "cyan"
                    "magenta" "pink"))

  (let ((g (simulation.display sim)))

    (define (draw color)
      (let ((x (vector-ref (simulation.x sim) i))
            (y (vector-ref (simulation.y sim) i)))
        (graphics-operation g 'set-foreground-color color)
        (graphics-draw-text g x y ".")))

    (cond ((not g) unspecific)
          ((string? color)
           (draw color))
          ((exact-integer? color)
```

```
 (if (and (<= 0 color)
          (< color (vector-length colors)))
     (draw (vector-ref colors color))
     (draw "white")))
(else
 (error:wrong-type-argument color "color string"
                            'color-me)))))
```

Index