The Hume Manual, Version 0.1

Greg Michaelson[1] Kevin Hammond[2]

[1]Department of Computing and Electrical Engineering
Heriot-Watt University
greg@cee.hw.ac.uk, +44 131 451 3422

[2]School of Computer Science, University of St Andrews
kh@dcs.st-and.ac.uk, +44 1334 463241

Contents

Changes as of 11/3/2003

Hume additions:

- output to std_out need not be forced by an explicit newline

Changes as of 5/11/2002

Hume additions:

- id@patt in patterns
- update ¡vector¿ ¡index¿ ¡value¿

Changes as of 22/8/2002

Hume additions:

- within for box
- within for expression
- within for output

Changes as of 13/3/2002

Hume additions:

- _* matching i.e. _ or
- Support for bitwise operations on integers with && and || (RP)
- Coercions from list/string ¡-¿ vector (RP)
- Coercions from sized int -¿ vector/list of sized int (RP)

Changes as of 7/3/2002

Hume additions:

- string as list of char
- list of char as string
- expression ¡expr¿ for top level ¡expr¿ evaluation

Changes as of 14/2/2002

Hume additions:

- box profiling;
- fair matching now least recently used instead of round robin;
- renumbered as Version 0.1.

Changes as of 7/2/2002

Hume additions:

- `as` with `int/float`, `int/string` and `float/string`;
- trigonometric functions: `sin`, `cos`, `tan`, `sqrt`, `exp` and `log`;

Chapter 1

Introduction

1.1 Overview

Hume is a programming language based on concurrent finite state machines with transitions defined through pattern matching and recursive functions. A formal definition of Hume may be found in [HM01].

This document describes how to write and run programs in Hume 0.1, a prototype implementation of a substantial Hume subset. Hume 0.1 includes:

- imprecise integer and float types
- bool, char and string types
- tuple, vector and list types
- conditional and case expressions
- local definitions
- recursive function definitions
- constant definitions
- type synonyms
- unions
- exceptions
- pattern matching boxes with unfair and fair matching
- wiring with input initialisation and output tracing
- streams, connected to files and standard input and output
- I/O and comparison overloaded for arbitrary sized structures

Hume 0.1 lacks:

- precise integer and float types
- nat, unicode, word, fixed and exact types

- ports
- time
- modules
- output initialisation and input tracing

The Hume 0.1 implementation:

- offers stepped or continuous mode execution
- offers output tracing in both modes
- displays box and wire states in stepped mode
- runs each box, to completion, with round robin scheduling
- lacks polymorphic type checking
- lacks exception consistency checking

1.1.1 Acquiring Hume

Hume can be acquired via: `www-fp.dcs.st-and.ac.uk/hume`.

DREADFUL WARNING...

Hume is very much under development. It is likely that this first implementation is full of bugs and unexpected behaviours. Please report:

- parser problems to Kevin Hammond - `kh@dcs.st-and.ac.uk`;
- run-time problems to Greg Michaelson - `greg@cee.hw.ac.uk`.

Chapter 2

Running Hume programs

2.1 Execution Model

A Hume program consists of one or more *boxes* with *inputs* and *outputs*. Boxes are generalised finite state machines. A box's transitions are defined by *patterns* on the inputs. Boxes are connected to each other, and to *streams* and *ports* by links established by *wiring*.

When a program is running each box repeatedly tries to match one of its pattern against its inputs. When a match succeeds, the expression associated with the pattern generates the values for the box's outputs. At the end of each execution cycle a box may be in one of three states:

- *runnable*: the box completed successfully;
- *matchfail*: the box lacked appropriate inputs;
- *blockedout*: the box completed but blocked as some outputs from the previous cycle were not consumed. On the next cycle, the box will again attempt to establish its outputs.

Matching may be *unfair* or *fair*. For unfair matching, on each cycle the matching starts with the box's first pattern. For fair matching, on each cycle the matching starts with a least recently used pattern over all previous cycles.

2.2 Runing Hume

By convention, Hume programs end with `.hume`.

To run a Hume program in *continuous mode*;

```
% hume program.hume
Hume 0.1
RUNNING
...
```

To run a Hume program in *stepped mode*:

```
% hume -t program.hume
Hume 0.1
STEPPING
ugly print outputs
```

```
traces: ...
states
wires: ...
NEXT> return
...
```

All boxes run for one cycle.

ugly print displays the program in a strange mix of Hume syntax and internal representation.

outputs displays the outputs to `std_out`.

`traces:` displays all outputs followed by `trace` in the corresponding wiring.

states: displays the state of each box.

`wires:` displays all unconsumed outputs.

Pressing *return* after `NEXT>` initiates the next cycle.

2.3 Halting execution

`Control C` halts execution and returns to the host system.

2.4 Example 1: Counter

The following program generates a sequence of ascending integers on the standard output, starting with `0`.

Note that line numbers are not part of Hume but are used in subsequent discussion.

```
 1 -- counter - inc.hume

 2 program

 3 stream output to "std\_out";

 4 box inc
 5 in (n::int 64)
 6 out (n'::int 64,shown::(int 64,char))
 7 match
 8  x -> (x+1,(x,'\n'));

 9 wire inc
10  (inc.n' initially 0)
11  (inc.n,output);
```

1: Comments begin with `--`.

2: Programs start with `program` or `module`.

3: The stream `output` is associated with standard output, named through the string `"std_out"`.

4: The box is named `inc`.

5: The box has one input called `n`, a 64 bit integer.

6: The box has two outputs. `n'` is a 64 bit integer. `shown` is a tuple of a 64 bit integer and a character.

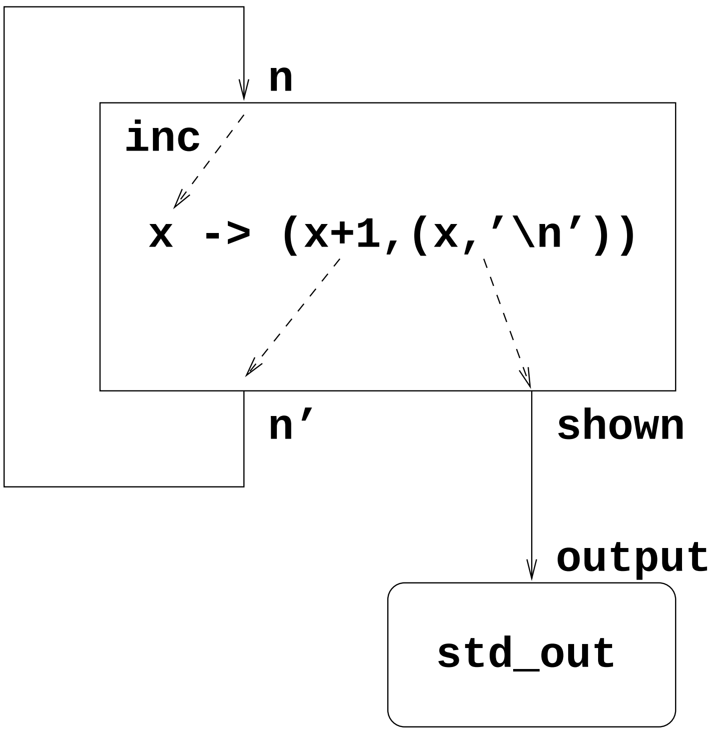

Figure 2.1: Counter - inc.hume

`7-8`: If there is a value on input `n`, then the pattern `x` will set a new local variable called `x` to that value and evaluate the right hand side. Output `n'` will be set to `x+1`. Output `shown` will be set to `(x+1,'\n')` i.e. `x+1` followed by a newline character.

`9`: Box `inc` is now wired by position.

`10`: `inc`'s input `n` is wired implicitly to `inc`'s input `n'`. `n` is initialised to `0`.

`11`: `inc`'s output `n'` is wired implicitly to `inc`'s input `n`. `inc`'s output `shown` is wired to `output` and hence to `"std_out"`.

The program is illustrated in Figure 2.1.

Thus, on each step, this program:

- sends `x+1` round the wire loop from `n'` back to `n`
- displays the old value of `n` i.e. `c`, followed by a newline

Running this example in continuous mode:

```
% hume inc.hume
Hume 0.1
RUNNING
0
1
2
3
...
```

Running this example in stepped mode:

```
% hume -t inc.hume
HUME 0.1
...
STEPPING
0
inc: RUNNABLE
wires: inc.n':1
NEXT>
1
inc: RUNNABLE
wires: inc.n':2
NEXT>
2
inc: RUNNABLE
wires: inc.n':3
NEXT>
...
```

At each stage:

- `n`'s value appears on the standard output.
- matching succeeds so `inc` is runnable;
- `n'` is set to one more than `n`'s value;

2.5 Example 2: Square and double

The following program extends the counter program to generate a sequence of squares and doubles of ascending integers. A new box `sqdouble` is added to:

- accept an integer from `inc` every two cycles;
- on the first cycle, print the integer's square;
- on the second cycle, print the integer's double;
- keep track as to whether it is currently squaring or doubling.

```
-- square and double

program

stream output to "std_out";

union STATE = SQUARING | DOUBLING;

box inc
in (n::int 64)
out (n'::int 64,shown::int 64)
match
n -> (n+1,n);

wire inc (inc.n' initially 0) (inc.n,sqdouble.n);
```

```
11 box sqdouble
12 in (n::int 64,oldn::int 64,s::STATE)
13 out (n'::(int 64,char),oldn'::int 64,s'::STATE)
14 match
15  (x,*,SQUARING) -> ((x*x,'\n'),x,DOUBLING) |
16  (*,x,DOUBLING) -> ((2*x,'\n'),*,SQUARING);

17 wire sqdouble
18  (inc.shown,sqdouble.oldn',sqdouble.s' initially SQUARING)
19 (output,sqdouble.oldn,sqdouble.s);
```

4 introduces a concrete data type `STATE` with values `SQUARING` and `DOUBLING`.

10 wires `inc.shown` to the new box `sqdouble`'s input `n`.

`11-16` introduce the new box `sqdouble`. It has 3 inputs and 3 outputs. Looking at the wiring in `18-19`: the link from`oldn` to oldn' circulates the last value from `inc`; the link from `s` to `s'` circulates local state information. Initially, `s` is set to `SQUARING`.

`15-16` define `sqdouble`'s transitions. For an input `n` from `inc.shown` and with `s` indicating the `SQUARING` internal state, `oldn` is ignored by the `*`, `n` is matched to `x` which is squared and output, `n` is circulated as `oldn'` back to `oldn`, and `s'` is circulated as `DOUBLING` back to `s`.

If `s` is `DOUBLING`, the `n` is ignored, `oldn` is matched to `x` which is doubled and output, no output is generated for `oldn'` by the `*` and `s'` is circulated as `SQUARING` back to `s`.

The effect is that for each integer that `inc` generates, `doublesq` will generate its square on the first cycle and its double on the second cycle. After the first cycle, `inc` will generate a new integer but will block on output until `sqdouble` has completed the second cycle and consumes it.

The program is illustrated in Figure 2.2.

Running the program in continuous mode:

```
%hume sqdouble.hume
HUME 0.1
RUNNING
0
0
1
2
4
4
9
6
...
```

Running the program in stepped mode:

```
% hume -t sqdouble.hume
HUME 0.1
...
STEPPING

sqdouble: MATCHFAIL; inc: RUNNABLE
wires: inc.n':1, inc.shown:0
NEXT>
```

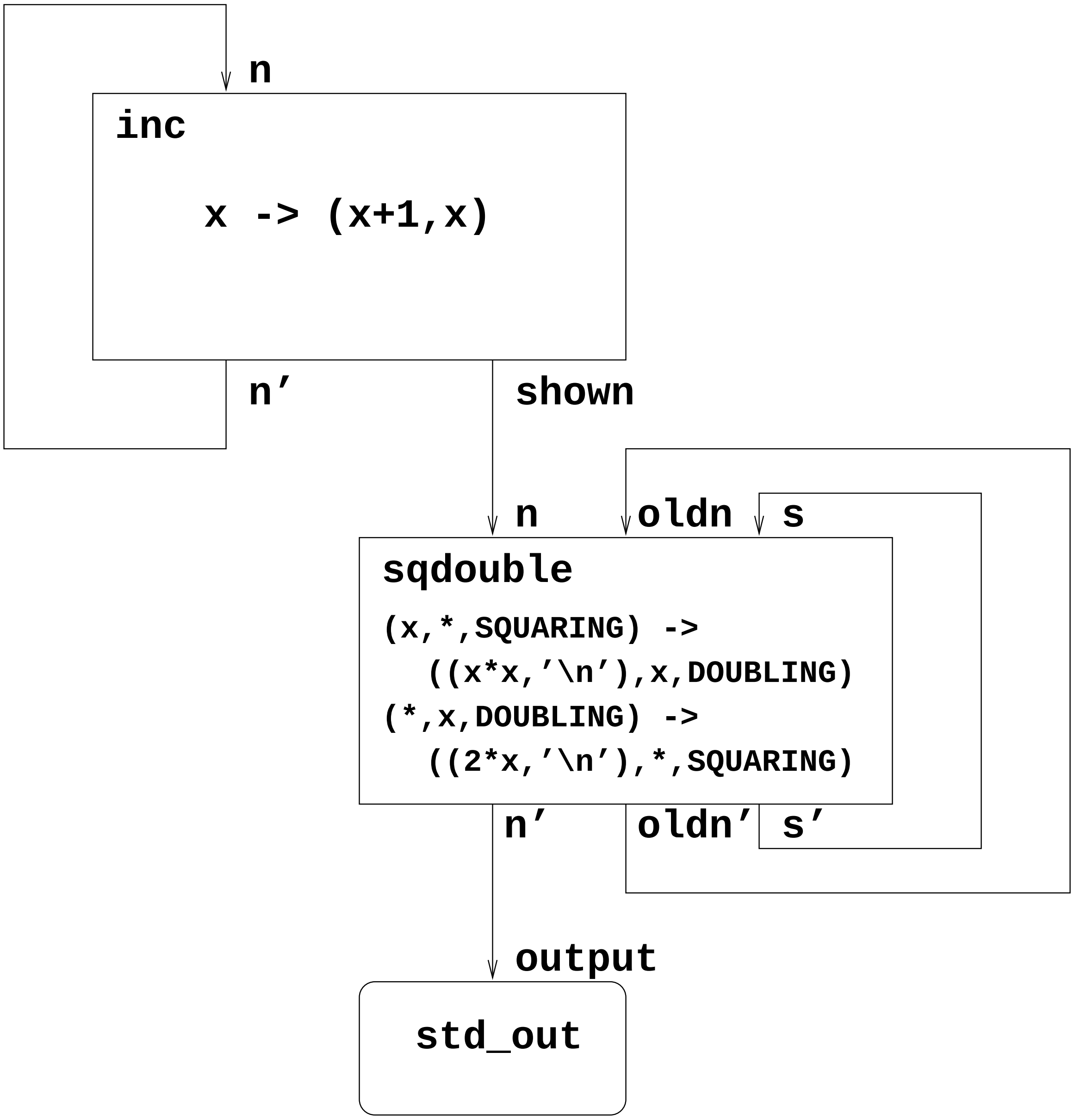

Figure 2.2: Square and double - sqdouble.hume

```
0
sqdouble: RUNNABLE; inc: RUNNABLE
wires: sqdouble.oldn':0, sqdouble.s':DOUBLING, inc.n':2, inc.shown:1
NEXT>
0
sqdouble: RUNNABLE; inc.shown: BLOCKEDOUT
wires: inc.shown:1, sqdouble.s':SQUARING
NEXT>
1
sqdouble: RUNNABLE; inc: RUNNABLE
wires: sqdouble.oldn':1, sqdouble.s':DOUBLING, inc.n':3, inc.shown:2
NEXT>
```

At the end of the first cycle:

- `sqdouble` could not match any of it's patterns.
- `inc`'s `n'` is 1 and it's `shown` is 0;

At the end of the second cycle:

- `sqdouble` has output the square of `0` from `inc`'s `shown` via it's own `n` i.e. `0`;
- `inc`'s `n'` is 2 and it's `shown` is 1;
- `sqdouble`'s `oldn'` is `0` and it's `s'` is `DOUBLING`.

At the end of the third cycle:

- `sqdouble` has ouput the double of `0` from it's `oldn` via it's own `oldn'` i.e. `0`;
- `sqdouble`'s `s'` is back to `SQUARING`;
- `inc`'s `shown` is `1` but this has been ignored by `sqdouble`. Thus `inc` is blocked on output.

At the end of the fourth cycle:

- `sqdouble` has output the square of `1` from it's `n` via `inc`'s `shown` i.e. `1`;
- `inc`'s `n'` is `3` and its `shown` is `2`;
- `sqdouble`'s `s'` indicates the `DOUBLING`internal state for it's `oldn'` which is now `1`.

Thus `sqdouble` runs twice as often as `inc`.

2.6 Example 3: Square and double with fair matching

For this example, the same effect can be achieved through *fair matching* without the `SQUARING/DOUBLING` feedback loop:

```
-- square and double 2
...
wire inc (inc.n' initially 0) (inc.n,sqdouble2.n);

box sqdouble2
```

```
12 in (n::int 64,oldn::int 64)
13 out (n'::(int 64,char),oldn'::int 64)
14 fair
15  (x,*) -> ((x*x,'\n'),x) |
16  (*,x) -> ((2*x,'\n'),*);

17 wire sqdouble2
18  (inc.shown,sqdouble2.oldn')
19  (output,sqdouble2.oldn);
```

We have dropped:

- the `s` input and `s'` output at `12-13`;
- the corresponding matches for `DOUBLING` and `SQUARING` at `15-16`;
- the corresponding wiring for `s` and `s'` at `18-19`.

and replaced `match` with `fair` at `14`.

Now, if the match at `15` succeeds on one cycle then the alternate pattern at `16` will be tried first on the next cycle, and vice versa. Here, the behaviour is the same as for `sqdouble` above.

2.7 Profiling

To profile a Hume program:

```
% hume -pnumber program.hume
RUNNING
...
standard output
...
status counts
longest wire delays
...
total elapsed time
Fail:  END PROFILING
```

The program is run for *number* cycles displaying *standard output*. The system then halts displaying profiling information for each box.

status counts shows how often a box has been RUNNABLE(R), BLOCKED OUTPUT(BO) and MATCH FAIL(MF). It also shows the cummulative elapsed time for all invocations of that box.

longest wire delays shows for each input to the box, the longest delay between the corresponding input value being made available and being consumed.

total elapsed time shows the overall elapsed time for the *number* cycles of all boxes, including startup and inter-cycle administrative costs.

For example:

```
% hume -p1000 sqdouble.hume
...
sqdouble:  R: 999, BO: 0, MF: 1; Cumm.  time:  0.22241953 secs
Longest wire delays:  oldn':  3.0394e-2 secs, s':  4.1939e-2 secs,
shown:  4.2017e-2 secs
```

```
inc:  R: 501, BO: 499, MF: 0; Cumm.  time:  0.103128806 secs
Longest wire delays:  n':  3.0499e-2 secs i
Total elapsed time:  0.351578 secs
Fail:  END PROFILING
```

This shows that:

- `sqdouble` ran continuously (`R: 999`) apart from the initial startup delay (`BO: 1`);
- `inc` ran for 50% of the time (R: 501, BO: 499), as expected;

2.8 Consume matching with _*

The pattern _* in a box match will behave like wildcard _ if the corresponding input is present, and consume it, or like * if no input is present, and succeed.

2.9 Top level expression evaluation

The top level construct:

`expression` *exp*;

will display the result of evaluating *exp*. For example:

```
-- expression.hume
expression 6*7;
...
```

will display:

```
% hume expression.hume
Hume 0.1
RUNNING
42
...
```

Chapter 3

Wiring

3.1 Normal Wiring

Wiring in Hume is used to define the connections between boxes within a Hume program, and the connection of that program to external streams of data. The normal syntax for wiring is currently:

```
<wiredecl> ::=  "wire" <boxid> <ins> <outs>
```

where `<ins>` and `<outs>` are either tuples of links, or just a single link.

```
<ins>/<outs> ::=        "(" <link1> "," ... "," <linkn> ")"
        | <link>
```

A link attaches inputs to outputs, or vice-versa, or attaches an input or output to a globally named stream. The links to a given box are specified positionally for the inputs/outputs of that box.

```
<link> ::=      <boxid> "." <id>
        |       <streamid>
```

The first form defines a box input or output, the latter attaches a box input or output to a stream. The predefined streams are given below

Stream	Attached
StdIn	standard input
StdOut	standard output

For example

```
box parity in ( input, par :: short ) out ( output, par' :: short )
match
  ( i, v ) -> let res = i + v in ( res, if even res then 1 else 0 )
;

wire parity ( StdIn, b.par' ) ( StdOut, b.par );
```

Initially-clauses may be attached to inputs. This is useful to initialise streams or values that are internal to a box as `par/par'` above.

```
wire b ( StdIn, b.par' initially 0 ) ( StdOut, b.par );
```

3.2 Templates and Replication

Box templates are defined similarly to boxes:

```
template <templateid> <ins> <outs>
<body>
;
```

where `<ins>` and `<outs>` are tuples of inputs and outputs respectively. Unlike a box, a template never generates a runtime process, and may be polymorphic. A template may be instantiated one or more times to yield concrete boxes.

```
template t in ( input, par :: short ) out ( output, par' :: short )
match
  ( i, v ) -> let res = i + v in ( res, if even res then 1 else 0 )
;

instantiate t as b1;
instantiate t as b2;


wire b1 ( StdIn, b1.par' initially 0 )    ( b2.input, b1.par );
wire b2 ( b1.input, b2.par' initially 0 ) ( StdOut, b2.par );
```

Boxes may also be replicated to give other (identical) boxes.

```
replicate box1 as box2;
```

A common requirement is to replicate a box or instantiate a template several times. A shorthand form can be used for this:

```
instantiate <templateid> as <boxid>*<exp>;
```

where `exp` yields a compile-time constant integer, n, where $n \geq 1$. For some free variable, `i`, this is exactly equivalent to the following form:

```
for i = 1 to <exp>
        instantiate <templateid> as <boxid>{i}
;
```

Wiring declarations may be included within loops. The general form of a wiring loop is:

```
for <id> = <expr1> to <expr2>
  [ except ( <eexpr1>, ..., <eexprn> ) ]
  <wiredecl>;
```

Within the loop, annotations may be used to index variants of names. The annotations (which are enclosed in braces – "" ... "") are constant integer expressions formed from wiring loop variables, integer literal constants, integer addition, subtraction, multiplication, division, and remainder operations, named compile-time constants declared in constant declarations, and expansions of macros that have been declared in macro declarations.

```
constant bmax = 8;

macro next n = n + 1;
macro prev n = n - 1;

instantiate t as b*bmax;

for i = 2 to bmax-1
   wire b{i} ( b{prev i}.output, b{i}.par' ) ( b{next i}.input, b{i}.par );

wire b1 ( StdIn, b1.par' ) ( b2.input, b1.par );
wire b{bmax} ( b{prev bmax}.output, b{bmax}.par' ) ( StdOut, b{bmax}.par );
```

It is possible to nest for loops, and loop variables may be used within inner wiring declarations.

3.2.1 Except-Clauses

Except-clauses are used to exclude some values from the loop variable. For example

```
constant min = 1;
constant max = 8;
constant bypass = 3;

instantiate t as b*max;

for i = min+1 to max-1 except ( bypass )
   wire b{i} ( b{i-1}.output, b{i}.par' ) ( b{i+1}.input, b{i}.par );

wire b{min} ( StdIn, b{min}.par' ) ( b{min+1}.input, b{min}.par );
wire b{max} ( b{max-1}.output, b{max}.par' ) ( StdOut, b{max}.par );
```

3.3 Wiring Macros

Wiring macros allow generic wiring templates to be defined. Within the wiring macro, the parameters may be used to specify the names of boxes or input/output links. For example,

```
constant RingSize = 8;

macro predR i = (i-1) % RingSize;
macro succR i = (i+1) % RingSize;

wire Track ( this, prev, next ) =
  wire {this} ( {this}.value', {prev}.outval, {next}.inctl )
              ( {this}.value,  {next}.outctl, {prev}.inval )
;

for i = 0 to RingSize
        wire Track ( Ring{i}, Ring{predR(i)}, Ring{succR(i)})
;
```

The syntax of a wiring macro definition is:

```
<wiredecl> ::= "wire" <macroid> <ids> "=" <wiredecl>
```

Wiring macros are instantiated by passing in concrete parameters to the macro

```
<wiredecl> ::= "wire" <macroid> <args>
```

Depending on usage, these arguments may be either box names or names of inputs/outputs. At present, it is not possible to use unrestricted values such as integers as arguments to wiring macros.

3.4 Extensions and Changes

A more flexible wiring primitive might be:

```
<wiredecl> ::= "wire" <linko> "to" <linki>
```

meaning that the output of `<linko>` should be wired to the input of `<linki>`. This primitive would allow more flexible wiring. For example, a wiring loop could set up a number of wires.

The track layout example makes extensive use of named records to set initial values. For example,

```
initial Ring{Train1Pos} { value = Just "Train1" };
```

Such uses can considerably simplify initialisation. The full syntax is:

```
<wiredecl> ::=  "initial" <boxid> "{" <wireinits> "}"
<wireinits> ::= <id> "=" <expr> [ "," <wireinits> ]
```

Chapter 4

Input/Output

4.1 Overview

In Hume, boxes communicate with the outside world over links to *streams*, associated with files, and to *ports*, associated with devices.

As with all Hume wiring, an individual stream or port can only be linked to one box input or output.

4.2 Streams

Streams are uni-directional and may be used for either input or output. Streams are declared by:

> input: `stream` *streamid* `from` *string*
> output: `stream` *streamid* `to` *string*

where:

> *streamid* = Hume *identifier;*
> *string* = file path within "s.

Standard input is `from "std_in"`.

Standard output is `to "std_out"`.

Streams are linked to inputs and outputs by mentioning their names in the appropriate positions in wiring.

Streams accept/deliver character sequences from/to sources/destinations. Such sequences are terminated with the character `'\0'`.

For input, streams should only be linked to inputs of determinate size type i.e. character, integer, float and bool, and arbitrarily nested tuples and vectors of these base types. The source character sequences will be automatically coerced to values of such types.

For output, streams may be linked to outputs of string and list type, as well as determinate size types. Link values will be automatically coerced to character sequences.

In both cases a canonical, flat, textual representation is used:

- integer/float/bool - text representation preceded by white space

- tuple/vector - unbracketed, white space separated element representation
- character - single un-quoted character
- string - output only - un-quoted text representation
- list - output only - unbracketed, white space separated element representation

For input, at the end of stream, the character `'\0'` is returned repeatedly.

For output, a stream is closed by the character `'\0'`.

In both cases, the system ought to shut any associated files...

4.3 How to...

4.3.1 Input a string

A string is of indeterminate size and delimitatation, and so cannot be input automatically. Instead, a vector of appropriate size should be used to input a fixed width string:

```
vector 1 .. size of char
```

Chapter 5

Timeouts

5.1 `timeout` and `within`

The Hume Report envisaged the placing of `timeout`s on boxes, expressions, inputs and outputs. However, Hume now distinguishes between:

- `within`: activity must complete within specified time from start;
- `timeout`: activity must start within specified time from last completion.

For `within`

- expression: expression must be evaluated within specified time;
- box: box must generate outputs within specified time of matching inputs;
- input: value on associated output must be consumed within specified time of being placed there;
- output: value on output must be consumed within specified time of being placed there.

For `timeout`

- expression: N/A;
- box: box must match input within specified time of last commiting outputs;
- input: value must be present on associated output within specified time of last value there being consumed;
- output: value must be present on output within specified time of last value there being consumed.

Note that input `within`s and timeouts are defined in terms of the associated output.

If a `within` or `timeout` does not satisfy its requirements then an exception is generated. The exception will be handled by the box that specifies the `within` or `timeout`. For an input,the exception is handled by its defining box, not by the box which defines the associated output.

Note that:

- input `within` and all `timeout`s are not yet implemented;
- exceptions and exception handling need to be tidied up and made uniform.

Chapter 6

Errors and debugging

6.1 Error detection

In Hume 0.1, errors are detected by the:

- *syntax analyser*: parses a program and builds an abstract syntax tree (AST);
- *name pass*: climbs the AST, checking static semantic consistency;
- *interpreter*: executes the AST.

Errors are also reported by the Haskell run-time system, within which Hume 0.1 is implemented.

It is important to note that Hume 0.1 lacks a type checker. The name pass will check many structural aspects of static semantics that type checking can encompass, for example consistent pattern size in matches and appropriate numbers of arguments in function calls. Furthermore, the interpreter carries out run-time type checking. Nonetheless, many statically detectable errors will not be picked up, resulting in deviant program behaviour or weird messages refering to failures in specific interpreter functions. For the latter, please send the Hume source and error message to: `greg@cee.hw.ac.uk` so that the error can be checked and documented.

6.2 Error messages

6.2.1 Syntax analyser

For a syntax error, the syntax analyser identifies the line number, character position and most recently consumed token. For example, for:

```
fac n = case n
         0 -> 1 |
         x -> x*fac (x-1);
```

the message is:

```
Fail: Parse error at line 2, column 15 (token CInt 1)
```

i.e. missing `of`.

6.2.2 Name pass

The name pass checks for:

- declaration before use
- repeated declaration of same identifier
- pattern size consistency in matches
- argument number consistency in function calls (incomplete)
- wiring consistency

Error messages identify the context of errors and their causes. The error messages are:

name1: *name2* `not declared`
- function *name*1 has not declared variable *name*2

name1: *name2* `declared already`
- function *name*1 has declared variable *name*2 more than once

name: `too many/few args:` *expression*
- function *name* is called with too many or too few arguments in the call *expression*

name: `too many/few patterns:` *pattern*
- function *name* contains a match option *pattern* with more/less elements than the first match option

`box` *name*: `missing wire`
- there is no `wire` definition for box *name*

`wire` *name*: `missing box`
- there is no `box` definition for *name* nominated by some `wire`

name1: *name2.name3* `not declared`
- the `wire` for *name*1 refers to undefined link *name*2.*name*3

name1: `stream.`*name2* `not declared`
- the `wire` for *name*1 refers to undefined stream *name*2

name: `too many/few wires`
- the `wire` for *name* has more/less `in/out` links than specified by its `box`

name1: `can't wire` *name2.name3* `to itself`
- the `wire` for *name*1 links *name*2.*name*3 to itself

name1: *name2.name3* `already` wired to *name4.name5*
- *one of* *name*2.*name*3 `and` *name*4.*name*5 `refered to in the wire for` *name*1 `has already been wired to something else`

name1: *name2.name3* `not wired reflexively to` *name4.name5*
`- a relexive link from` *name*2.*name*3 `to` *name*4.*name*5`, refered to in the wire for` *name*1`, is missing`

If a program contains name pass errors then it is not executed.

6.2.3 Interpreter

The interpreter checks for:

- exception catching and handling
- type consistency in operations

Error messages are only generated for

`uncaught exception:` *exception*
- the box which raised *exception* doesn't handle it

`exception mismatch:` *exception*
- the handler for *exception* has an inappropriate pattern

Other errors generate exceptions.

6.3 Debugging

6.3.1 Tracing

Outputs may be traced by writing `trace` after an input in a wiring specification. For example, given the counter program `inc.hume`:

```
stream output to "std_out";

box inc
in (n::int 64)
out (n'::int 64,shown::(int 64,char))
match
 n -> (n+1,(n,'\n'));

wire inc (inc.n' initially 0) (inc.n trace,output);
```

then the `trace` after `inc.n` traces `inc.n'` to which it is wired:

```
$ hume inc.hume | more
HUME 0.1
RUNNING
traces: inc.n':1;
0
traces: inc.n':2;
1
...
```

6.3.2 Common problems

Lots of wiring errors: check consistency of spelling of inputs and outputs.

Run-time error attributed to `updateinitially`: mismatch between number of expressions on right hand side of box match and number of `out` wires.

Box is never runnable: missing input initialisation.

Box never reaches a pattern: inappropriate use of `*` or fair matching required.

Box blocked on output: check wiring/behaviour of box that consumes output.

Nothing appears on standard output: missing `'\n'` at end of output.

Strange box input match behaviour: mis-spelled constructor in pattern treated as variable.

Chapter 7

Exceptions

7.1 Exception handling

Hume 0.1 does not yet check exception consistency, i.e. that:

- all exceptions that are handled by a box are specified as being handled by that box, and vice-versa;
- all exceptions that may be raised in a box are handled by that box;
- all exceptions that are handled by a box are defined.

7.2 System exceptions

The current implementation is not yet compliant with the system exceptions specified in [HM01]...

Chapter 8

Standard Prelude

The Hume *standard prelude* will probably look something like:

```
module Prelude where

data Maybe a = Just a | Nothing;

maybe :: a -> (b -> a) -> Maybe b -> a;
maybe x f Nothing = x;
maybe x f (Just a) = f a;

data Either a b = Left a | Right b;

map :: (a->b) -> [a] -> [b];
map f [] = [];
map f (a:as) = f a : map f as;

foldr :: (a->b->b) -> b -> [a] -> b;
foldr f x [] = x;
foldr f x (a:as) = f a (foldr f x as);

member :: a -> [a] -> bool;
member x [] = false;
member x (y:ys) = if x == y then true else member x ys;

any, all :: (a->bool) -> [a] -> bool;
any p [] = false;
any p (x:xs) = if p x then true else any p xs;

all p [] = true;
all p (x:xs) = if p x then all p xs else false;

not :: bool -> bool;
not true = false;
not false = true;

type Byte = int 8;
type Short = int 16;
type Int = int 32;
```

```
type Long = int 64;
```

The standard prelude will eventually be loaded with a Hume program at run time. At present, it should be inserted at the top of programs.

Appendix A

Syntax

This appendix gives a BNF definition of the Hume syntax. The meta-syntax is conventional. Terminals are enclosed in double quotes `" ... "`. Non-terminals are enclosed in angle brackets `< ... >`. Vertical bars `|` are used to indicate alternatives. Constructs enclosed in brackets `[ ... ]` are optional. Parentheses `( ... )` are used to indicate grouping. Ellipses (`...`) indicate obvious repetitions. An asterisk (`*`) indicates zero or more repetitions of the previous element, and a plus (`+`) indicates one or more repetitions.

Programs and modules

```
<program> ::=
        "program" <decls>

<module> ::=
        "module" <modid> "where" <decls>
```

Declaration Language

```
<decls> ::=
        <decl1> ";" ... ";" <decln>     n >= 1

<decl> :: =
          "import" <idlist>
        | "export" <idlist>
        | "exception" <exnid> <exprtype>
        | "union" <typeid> <varids> "=" <constrs>
        | "type" <typeid> <varids> "=" <type>
        | "constant" <varid> "=" <cexpr>
        | "stream" <iodes>
        | "port" <iodes>
        | <boxdecl>
        | <wiredecl>
        | <fundecl>

<constrs> ::=
            <conid1> <type11> ... <type1n>      m > 0, n >= 0
        "|" ...
        "|" <conidm> <typem1> ... <typemn>

<iodes> ::=
        ( <strid> | <portid> ) ( "from" | "to" ) <string>
        [ "timeout" <cexpr> ]

<fundecl> ::=
          <varid> "::" <type>
        | <varid> <args> "=" <expr>
        | <patt1> <op> <patt2> "=" <expr>

<args> ::=
        <patt1> ... <pattn>                   n >= 0

<fundecls> ::=
        <fundecl1> ";" ... ";" <fundecln>     n >= 1
```

Types

```
<type> ::=
          <exprtype>
        | "stream" <exprtype>           stream type
        | "port" <exprtype>             port type
        | "time"                        time type
        | "bandwidth"                   bandwidth type

<exprtype> ::=
          <basetype>                    base type
        | "vector" <range> "of" <type>  vector
        | "()"                          empty tuple
        | "(" <type> "," <types> ")"    tuple
        | "[" <types> "]"               list
        | <typeid> <type1> ... <typen>  discr. union, n >= 0
        | <type> "->" <type>            function type
        | "view" <type>                 view as type
        | "(" <exprtype> ")"            grouping

<types> ::=
        <type1> "," ... "," <typen>     n >= 0

<basetype> ::=
          "int" <precision>
        | "nat" <precision>
        | "bool"
        | "char"
        | "unicode"
        | "string" [ <intconst> ]
        | "word" <precision>
        | "float" <precision>
        | "fixed" <precision>
              [ @ ( "2" | "10" | "16" ) [ "**" <intconst> ] ]
        | "exact"

<precision> ::=
        "1" | ... | "64"

<range> ::=
        <intconst1> ".." <intconst2>
```

Expression Language

```
<expr> ::=
          <constant>
        | <varid>                      variable/constant
        | <expr1> <op> <expr2>         binary operator
        | <varid> <expr1> ... <exprn>  function appl., n >= 1
        | <conid> <expr1> ... <exprn>  constructor appl., n >= 0
        | "[" <exprs> "]"              list
        | "()"                         empty tuple
        | "(" <expr> "," <exprs> ")"   tuple
        | "<" <exprs> ">"              vector
        | "case" <expr> "of" <matches> case expression
        | "if" <expr1> "then" <expr2>  conditional
                       "else" <expr3>
        | "let" <fundecls> "in" <expr> local definition
        | <expr> "::" <exprtype>       type cast/view
        | <expr> "as" <exprtype>       type coercion
        | "raise" <exnid> <expr>       raise an exception
        | <expr> "within" <cexpr>      timeout
        | "(" <expr> ")"               grouping

<cexpr> ::= <expr>                     constant expression

<exprs> ::=
        <expr0> "," ... "," <exprn>    n >= 0

<matches> ::=
        <match1> "|" ... "|" <matchn>  n >= 1

<match> ::=
       <patt> "->" <expr>
```

Constants

```
<constant> ::=
          <intconst>
        | <floatconst>
        | <boolconst>
        | <charconst>
        | <stringconst>
        | <wordconst>
        | <timeconst>
```

Patterns

```
<patt> :: =
          <constant>
        | <varid>                      variable
        | <conid>                      nullary constructor
        | "_"                          wildcard
        | "[" <patts> "]"              list pattern
```

```
        | "<" <patts> ">"              vector pattern
        | "()"                         empty tuple pattern
        | "(" <patt> "," <patts> ")"   tuple pattern
        | <conid> <patt1> ... <pattn>  discr. pattern, n >= 1
        | "(" <patt> ")"               grouping

<patts> ::=
        <patt0> "," ... "," <pattn>    n >= 0
```

Coordination language

```
<boxdecl> ::= <prelude> <body>

<prelude> ::=
        "box" <boxid>
        "in"  <inoutlist>
        "out" <inoutlist>
        [ "handles" <exnidlist> ]

<inoutlist> ::=
        <inout1> "," ... "," <inoutn>          n >= 1

<inout> ::=
        <varid> "::" <exprtype>
```

Boxes

```
<body> ::=
        "match"
        <boxmatches>
        [ "timeout" <cexpr> ]
        [ "handle" <handlers> ]

<handlers> ::=
        <handler1> "|" ... "|" <handlern>      n >= 1

<boxmatches> ::=
        <matches>
```

```
<handler> ::=
        <hpatt> "->" <cexpr>

<hpatt> ::=
        <exnid> <patt1> ... <pattn>             n >= 1
```

Replication

```
<replication> ::=
"replicate" <boxid> "as" <boxid> [ "*" <intconst> ]
```

Wiring

```
<wiredecl> ::=
        "wire" <boxid> <sources> <dests>

<sources>/<dests> ::=
        "(" <link1> "," ... "," <linkn> ")"     n >= 0

<link> ::=
          <connection>
        | <strid>
        | <portid>

<connection> ::= <boxid> "." <varid>
```

Identifiers

```
<id>     ::= [ <modid> "." ] <localid>

<idlist> ::= <id1> "," ... "," <idn>            n >= 1

<varids> ::= <varid1> ... <varidn>              n >= 0

<exnidlist> ::= <exnid1> "," ... "," <exnidn>   n >= 1

<boxid>  ::= <id>
<modid>  ::= <id>
<exnid>  ::= <id>
<conid>  ::= <id>
<typeid> ::= <id>
<varid>  ::= <id>
<strid>  ::= <id>
<portid>  ::= <id>
```

Lexical Syntax

```
<localid> ::= <letter> ( <letter> | <digit> ) *

<op> ::= ( "+" | "-" | "*" | "/" ... ) *

<intconst>    ::= <digit> +
<floatconst>  ::= <intconst> "." <intconst> [ "e" <intconst> ]
<boolconst>   ::= "true" | "false"
<charconst>   ::= "’" <char> "’"
<stringconst> ::= """ <char> * """
<wordconst>   ::= "0x" <hexdigit> +
<timeconst>   ::= <intconst> <timedes>
<timedes>     ::= "ps" | "ns" | "us" | "ms" | "s" | "min"
<char>        ::= "A" | ... | "Z" | " " | "\t" | "\n" | "\\" |
                  "0x" <hexdigit> +
```

Hume Functions and Operators

Int

```
+, -, *, div :: Int -> Int -> Int
=, /=, <=, <, >, >= :: Int -> Int -> Bool
```

Nat

```
+, -, *, div :: Int -> Int -> Int
=, /=, <=, <, >, >= :: Int -> Int -> Bool
```

Word

```
+, - :: Word -> Nat -> Word
=, /=, <=, <, >, >= :: Word -> Word -> Bool
```

Vectors

```
range :: vector
```

Tuples

```
=, /=, <=, <, >, >= :: (a1, .., an) -> (a1, ..., an) -> Bool
```

Lists

```
++ :: [a] -> [a] -> [a]
member :: [a] -> a -> Bool
=, /=, <=, <, >, >= :: [a] -> [a] -> Bool
```

Bibliography

[HM01] Kevin Hammond and Greg Michaelson. The Hume Report, Version 0.0. Technical report, Deptarment of Computing and Electrical Engineering, Heriot-Watt Universty/School of Computer Science, University of St Andrews, 2001. `http://www-fp.dcs.st-and.ac.uk/hume/report/hume-report.ps`.