# The Hume Manual, Version 1.5

Greg Michaelson[1] Kevin Hammond[2] Robert Pointon
Norman Scaife

[1]School of Mathematical and Computer Sciences
Heriot-Watt University
greg@macs.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

May 19, 2009

# Contents

## Changes as of 6/11/8

Extended I/O section to further illustrate file I/O (thanks to Xiaotian Xu)

## Changes as of 10/10/8

Updated FFI section. Added text on sockets.

## Changes as of 29/8/6

Chapter for humec added.

## Changes as of 1/8/6

Interpreter additions

- EndOfFile exception now thrown

## Changes as of 5/7/6

- renumbered as version 1.3 (based on comment in last changes)
- vector now use size rather than bounds
- Corrected syntax - vectors
- Updated standard function list

## Changes as of 6/1/6

Interpreter additions

- Version 1.2 has superstep scheduling as in semantics

## Changes as of 1/12/2005

Interpreter additions

- exception checking added to name pass
- system exceptions now name consistent with report

## Changes as of 12/9/2005

Chapter for phamc/hami added.

## Changes as of 8/9/2005

Interpreter additions

- trace works on input wire
- -t no longer prints all wires
- -t shows successful pattern
- -e traces expression evaluation

## Changes as of 22/8/2004

Interpreter additions:

- import for nested source files

## Changes as of 11/8/2003

Interpreter additions:

- hume -c for continuous tracing

## Changes as of 11/3/2003

Interpreter additions:

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

## Changes as of 5/11/2002

Interpreter additions:

- id@patt in patterns
- update `<vector> <index> <value>`

## Changes as of 22/8/2002

Interpreter additions:

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

## Changes as of 13/3/2002

Interpreter additions:

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

## Changes as of 7/3/2002

Interpreter additions:

- string as list of char
- list of char as string
- `expression <expr>` for top level `<expr>` evaluation

## Changes as of 14/2/2002

Interpreter additions:

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

## Changes as of 7/2/2002

Interpreter 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 [**?**].

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
- 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`, or `www.macs.hw.ac.uk/~greg/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@macs.hw.ac.uk`.
- compiler problems to Robert Pointon - `rpointon@macs.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 new values for the box's outputs. If all relevant outputs from the previous cycle have been consumed then the new outputs are established. Otherwise the box blocks without establishing any outputs until the next cycle.

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 at least one output from the previous cycle was 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.

To run a Hume program in *continuous* stepped mode:

```
% hume -c program.hume
```

## 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 `--`.

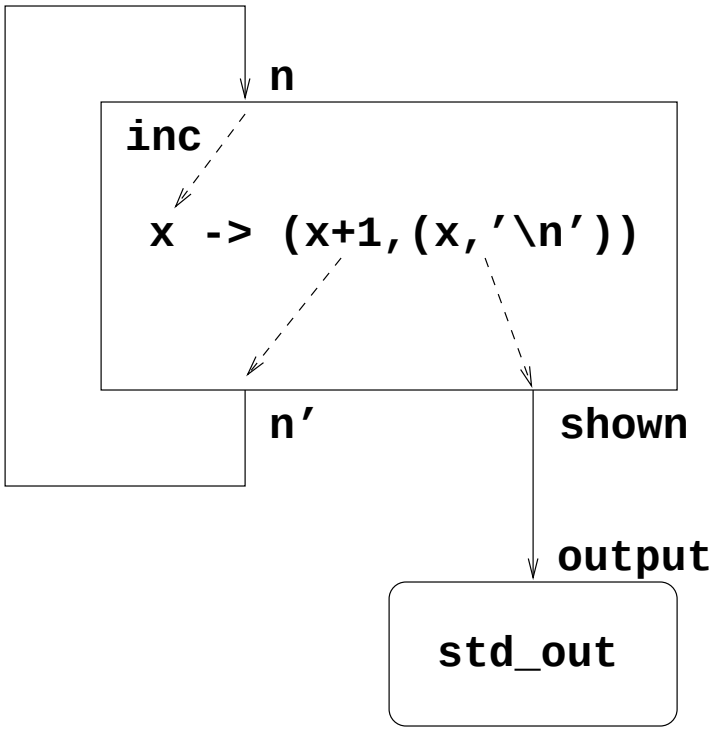


Figure 2.1: Counter - inc.hume

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.

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.

```
1 -- square and double

2 program

3 stream output to "std_out";
```

```
 4 union STATE = SQUARING | DOUBLING;

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

10 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:

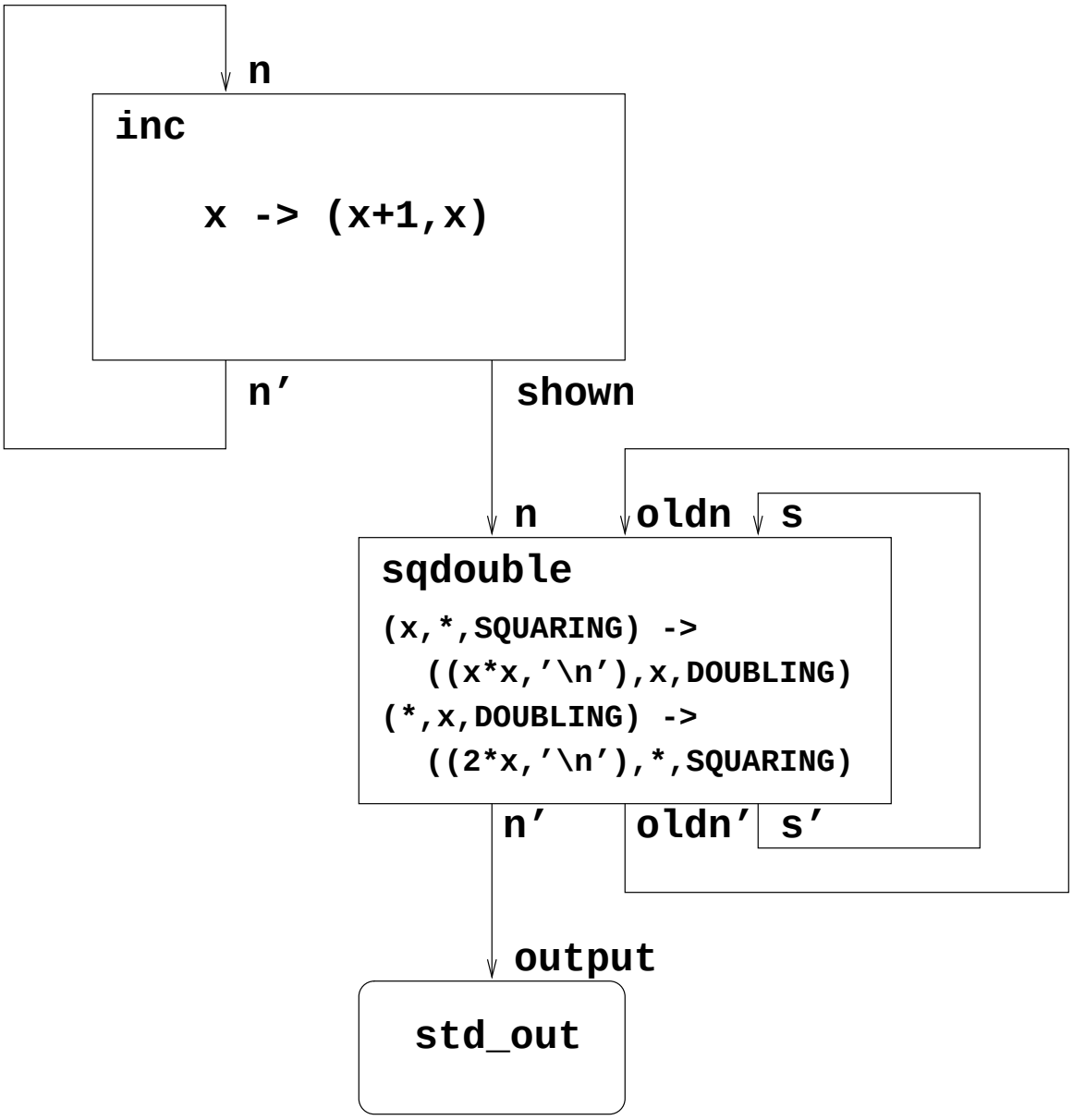


Figure 2.2: Square and double - sqdouble.hume

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

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

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

11 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
...
```

## 2.10 Importing files

A file may contain directives of the form:

```
import path
```

where *path*.`hume` is a valid file.

The file, and any files imported within it, will be appended together prior to parsing.

Imports may be nested to arbitrary depth. Recursive imports are ignored.

Alas, parser error message line numbers are relative to the fully expanded program.

# 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 a stream, the exception `EndOfFile` is raised. Note that this implies that it is not possible for a single box to acummulate input on a feedback loop for output on end of file. Instead, two boxes must be used, where the first repeatedly sends the next data values from the stream to the accumulating second. On handling `EndOfFile`, the first box must send a nominated end of file value, or an explicit handshake, to enable the second to output the accumulated input.

For example:

```
stream input from "data.txt";

data FSTATE = OK | EOF;

box getfile
in (n::integer)
out (s::FSTATE,n'::integer)
handles EndOfFile
match
 n -> (OK,n)
handle
 EndOfFile _ -> (EOF,*);

wire getfile
(input)
(getinput.s,getinput.i);

box getinput
  in (s::FSTATE,i::integer,li::[integer])
  out (lo::[integer],showRe::[integer] )
match
  (OK,n, l) -> (list l n,*) |
  (EOF,*, l) -> (*,l);

wire getinput
  (getfile.s,getfile.n',getinput.lo initially [] )
  (getinput.li,...);
```

Here, `getfile` repeatedly sends integers from `data.txt` to `getinput` for accumulation into a list, along with the flag `OK`. When `getfile` handles the `EndOfFile` exception, it sends `getinput` the flag `EOF` without an integer, and `getinput` sends the list to the output wire `showRe`.

Previously, for input, at the end of stream, the character `'\0'` was returned repeatedly. This is now mostly illegal.

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 size of char
```

### 4.3.2 Socket Input/Output

Since the stream file path is actually a URI we can also define other input/output types. As well as the plain file format:

```
stream file from "/home/user/filename.txt";
```

or:

```
stream file to "file:///home/user/filename.txt";
```

Hume also implements `data://` and `bind://`. There is also a very primitive implementation of `http://` but is not much use without the machinery to interpret the HTML read in. These allow access to socket interfaces (`AF_INET` sockets only). Servers are implemented using `bind://`:

```
stream input from "bind://localhost:5555";
```

This opens a socket to the `<host>:<port>` combination. A listener is spawned and when the first client connects the resulting path is connected to the Hume stream and can be read as normal. Note that only a single connection is possible and that currently, the server exits once the client disconnects. Clients are implemented using `data://`:

```
stream output to "data://localhost:5555";
```

This simply opens the socket and connects to it, linking the input stream to the associated Hume stream. The server socket has to exist, otherwise the program is terminated with an error. Note, however, that there is special provision in Hume's socket drivers to handle simultaneous read/write to the same socket:

```
stream output to "data://localhost:5555";
stream input from "data://localhost:5555";
```

### 4.3.3 Low-level Input/Output

For the Renesas M32C target, Hume provides access to low-level input/output ports and can generate values for some of the interrupts. For example, we can write to the `p2` output port with the `memory` declaration:

```
memory p2 to "P2:p8";
```

We can also read the analogue to digital port 0 with:

```
memory ad00 from "AD00L:p8";
```

The `p8` type definition is required to force 8-bit access to these memory locations. The `P2` and `AD00L` definitions are macros provided in the `arch_io_m32c.txt` file provided with the compiler. A full set of these are provided but initialization of registers upon bootup has to be handled in the startup code provided with whichever compiler is being used to target the board.

An additional facility is to generate values upon interrupts. For example, we can create an input stream from the Timer 0 and `INT0` interrupts with the following:

```
interrupt p  from "TIMERB0";
interrupt b1 from "INT0";
```

These statements create Hume streams `p` and `b1` which can be wired to box inputs. Interrupt service routines are generated by the compiler to service the relevant interrupts and place a unit "`()`" value on the associated Hume stream. Note that the code generated is most likely to be specific to one target platform and one compiler.

# 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 `timeout`s 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

# Foreign Function Interface

The Hume Foreign Function Interface (FFI) currently allows C functions to be called from within Hume programs. Note that the resource analysis tools are currently unable to analyse code with such calls although, in future, it may be possible to annotate these calls with resource bound information which must have been computed and validated externally to the Hume language.

The full Hume standard syntax for defining the call in Hume is as follows:

```
<foreigndecl> ::=
          "foreign" "import" <callconv> [ <safety> ] [ <string> ]
                             <id> "::" <type>

<safety> ::=
           "safe" | "unsafe"

<callconv> ::=
          "ccall" | "stdcall" | "cplusplus" | "jvm" | "dotnet"
```

although currently, only the `ccall` and `stdcall` conventions are implemented and the safety clause is not used at all. The string defines the name of the C function to be called and can optionally include the names of header files to be included in the wrapper code. The `<id>` is the name of the function in Hume's namespace. For example, to call the `sinh` function in the C standard library the following definition is required:

```
foreign import ccall "math.h sinh" hsin :: float 32 -> float 32;
```

Note that the contents of the string are actually operating system dependent. The function thus defined can be used as a standard Hume function:

```
expression(hsin 0.5);
```

Functions of more than one argument are handled correctly:

```
foreign import ccall "math.h powf"
  powf :: float 32 -> float 32 -> float 32;
```

Note, however, that only the types `int`, `float`, `char`, `bool` and fixed-length strings can be passed in this way and that the header file is *essential* in order to construct the call to the library function correctly.

To pass and return more complex types, there has to be wrapper code supplied by the user to unpack and repack the heap data which is supplied to the external function. For example, suppose we wish to pass a vector to a function:

```
foreign import ccall "test.h myvecfn"
  vecfn :: vector 2 of float 32 -> vector 2 of float 32;
expression(vecfn <<1.0,2.0>>);
```

In this instance the external function is passed the address of the Hume heap where the data resides:

```
float myvecfn(HEAP *x)
{
```

We then call the `heapToBytes` function to unpack the data into temporary storage:

```
  float tmpx[2];
  heapToBytes((int8 *)tmpx, x);
```

The `heapToBytes` function is able to decode the unstructured data (actually of type `uint8_t`) on the heap because the type of the data was pushed onto the stack before the function was called. The temporary data can then be overwritten, if required:

```
  tmpx[0] += tmpx[1];
```

Note that the entire data structure is copied in this way which has implications for the efficiency of external function calls.

Conversely, if we wish to return such data, we need to repack the data into the Hume heap:

```
  return bytesToHeap((int8 *)tmpx);
}
```

For the purposes of this interface, tuples appear identical to vectors:

```
foreign import ccall "test.h myinv"
  myinv :: (float 32,float 32) -> float 32;
expression(myinv (0.5,0.75));
```

requiring the following decode:

```
float myinv(HEAP *tup2)
{
  float tmp_t2[2];

  heapToBytes((int8 *)tmp_t2, tup2);
```

Vectors and tuples can be arbitrarily nested but it is the responsibility of the external function call to ensure that the boundaries are respected.

There is also a maximum number of arguments which can be passed, the limit can be deduced from the `ccall.h` header file supplied with the compiler, the maximum number being the highest-indexed `CCALL` macro (currently 7). However, new macros of this type can be added manually, if required.

The usual procedure for costructing more complex calls is to provide temporary storage for all structured data, unpack them from the heap, perform the required processing including calls to external libraries and then repack the result data, tupling if multiple return values are required.

As a complete example, suppose we wish to compute the inverse of a $4 \times 4$ matrix using LU factorization which is computed by the `getrf` and `getri` functions provided by the LAPACK library. The Hume code would consist of the following:

```
import inttypes;

exception matinverseLapackErrorCode :: (int32_t);

type t_4x4 = vector (4,4) of float64;
type t_1x4 = vector (1,4) of int32_t;

foreign import ccall "humelapack.h hume_getrf"
  hume_getrf_4x4 :: int32_t -> int32_t -> t_4x4
                 -> (int32_t, t_4x4, t_1x4);
foreign import ccall "humelapack.h hume_getri"
  hume_getri_4x4 :: int32_t -> t_4x4 -> t_1x4 -> (int32_t, t_4x4);
matinverse_4x4 a =
  case hume_getrf_4x4 4 4 a of
    (0,lu,ipiv) ->
      (case hume_getri_4x4 4 lu ipiv of
         (0,am1) -> am1
       | (info,_) -> raise matinverseLapackErrorCode info)
  | (info,_,_) -> raise matinverseLapackErrorCode info;
```

The types used here are Hume equivalents of the `inttypes.h` header file as part of the C standard library (it makes it easier to compare with the LAPACK C routines which also use them). We define two Hume types, `t_4x4` which is a $4 \times 4$ matrix and `t_1x4` for a $1 \times 4$ vector. Note that we wish to return the error code provided by the C library function so we have to arrange to tuple the return value. The C wrapper file might look like:

```
typedef struct info_res {
    int32 info;
} INFO_RES;

HEAP *hume_getrf(int m,int n,HEAP *a)
{
  static float64 *A=NULL;
  static integer *IPIV=NULL;
  static INFO_RES *res=NULL;
  static size_t A_size=0,IPIV_size=0,res_size=0;
  float64 *rA;
  int32 *rIPIV;
  integer info;
  int i,lda,asize;

  lda = max(1,m);
  asize = lda*n;
  ensure_size((void **)&A,&A_size,asize,sizeof(float64));
  ensure_size((void **)&IPIV,&IPIV_size,min(m,n),sizeof(integer));
  ensure_size((void **)&res,&res_size,
    sizeof(INFO_RES)+(asize*sizeof(float64))+(min(m,n)*sizeof(int32)),
    sizeof(int8));
  rA = (float64 *)(res+1);
  rIPIV = (int32 *)(rA+asize);
```

```
  heapToBytes((int8 *)A, a);
  dgetrf(m,n,A,lda,IPIV,&info);
  res->info = info;
  for (i = 0; i < asize; i++) rA[i] = (float64)A[i];
  for (i = 0; i < min(m,n); i++) rIPIV[i] = (int32)IPIV[i];
  return bytesToHeap((int8 *)res);
}
```

This function follows the advice concerning the sizes of intermediate data recommended by the documentation for the central `dgetrf` function. We also use a system of self-expanding buffers here (`ensure_size`) which allows the routine to handle any size of input matrix but for fixed matrix sizes we could just use local storage. The input data is decoded (`heapToBytes`), the target function called and then we build up the return tuple. In fact, we create a structure to hold the `info` value and then attach the other data (returned matrix `rA` and the pivot information `rIPIV`). Finally, the constructed data is packed back onto the heap using the `bytesToHeap` function. Don't forget the header definition:

```
extern HEAP *hume_getrf(int,int,HEAP *);
```

The Hume FFI call here is of a fixed type, here $4 \times 4$. If we are required to invert other sizes of matrices we have to create additional `foreign` definitions. However, the way we have setup the wrapper code, we can just call the same wrapper for any size of matrix. This is a slight problem for Hume since potentially we might require numerous instances of the same call.

Note that there currently some limitations of the interface. The C type `double` is not handled properly on return from the external C call. However, we can get round this by packing it into a single element tuple:

```
foreign import ccall "wvtod"
  wvtod_ :: int 32 -> vector 8 of word 8 -> vector 1 of float 64;
wvtod eid v = case wvtod_ eid v of << d >> -> d;
```

Another problem is that strings are not necessarily null-terminated when they are passed from Hume to C. A good policy is to pass the size of the string along with the string itself:

```
foreign import ccall "stdlib.h mysystem"
  system :: string -> int 32 -> int 32;
expression(system "ls" 2);
```

Finally, to compile external code, you can either simply add the wrapper code to the header file (a bit unethical for correct C programming) or you can compile the external code separately and include it in Hume as a library:

```
% gcc -g -march=i386 -m32 -I ~/humec/include/humec/ -c test.c
% humec -lotsaspace -libs test.o ffi4.hume
```

This would be for a 64-bit architecture where we currently have to use a 32-bit compile.

# Chapter 7

# Errors and debugging

## 7.1 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.

## 7.2 Error messages

### 7.2.1 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`.

### 7.2.2 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.

### 7.2.3 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.

## 7.3 Debugging

### 7.3.1 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
...
```

### 7.3.2 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 8

# Exceptions

## 8.1 Exception handling

Checks are made for 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.

## 8.2 System exceptions

The current interpreter implementation is compliant with the system exceptions specified in [**?**]. Note that in the Report the type of the return values of system exceptions is not specified. At present, the interpreter returns strings representing the offending code where possible.

# Chapter 9

# phamc/hami

## 9.1 Overview

The phamc (prototype Hume abstract machine compiler) and hami (Hume abstract machine interpreter) tools provide an alternative way of executing Hume programs.

## 9.2 Usage

phamc is a command line tool that takes a *.hume* file and produces a *.ham* file for use in hami, in addition it produces a *humestubs.c* file for foreign language interfaces. The tool has the following options:

```
phamc {options}* <file.hume>
-a show abstract syntax tree
-b compile byte code
-c show compiled
-d dissemble
-t type check
-nc use name pass
```

hami is a command line tool for executing a *.ham* file and has the following options:

`hami {options}* <file.ham>`
*Main options:*
`-c` *use cost analysis*
`-h` *print this help message*
`-l` *log execution of abstract machine instructions*
`-p n` *run for n scheduler cycles only*
`-q` *quiet mode: no banner*
`-t` *trace box inputs and outputs*
*Other Options:*
`-nr` *read instructions and initialise only, do not run*
`-ns` *do not use the builtin show primitive (as)*
`-o` *discard output*
`-s` *show execution statistics*
`-sio` *show wire statistics*
`-x box n` *run box n times only*
`-R` *allow incomplete rule matches*

In normal use phamc is given no options, and hami is encouraged to use the costing information:

```
% phamc program.hume
...
% ls
program.hume program.ham humestubs.c

% hami -c program.ham
...
```

## 9.3 Extensions

ehami (extended/embeded hami?) adds several extensions to the hami engine.

**Note: ehami is not under current development, as it was a more a vehicle for testing ideas for humec.**

### 9.3.1 Input/Output

A variety of I/O types are supported:

| Variety: | Keywords: | Parameter: | Type: | |
|---|---|---|---|---|
| console in | stream from | "std_in" | char | [1,2] |
| console out | stream to | "std_out" | <any> | [1] |
| console error | stream to | "std_err" | <any> | [1] |
| file in | stream from | <filename> | char | [1,2] |
| file out | stream to | <filename> | <any> | [1] |
| clock | stream from | "clock" | int | [3] |
| interrupt | interrupt from | <signalname> | () | |
| memory in | memory from | <address> | int | [4] |
| memory out | memory to | <address> | int | [4] |
| args | stream from | "main_args" | string | |
| environment | stream from | "main_envs" | (string,string) | |
| exit code | stream to | "main_exit" | int | [5] |

1. Console and file I/O is unbuffered, non-blocking, and signal driven.
2. hami has no type information so cannot do automagic typed reading like the interpreter.
3. Clock is a timebase at ms resolution.
4. Adddress has syntax ['0x' hexnum | decnum] { ':' [p8|p16|p32|0..7] }, where the final part refers to the size of the integer, or 0..7 to refer to an indivdual bit in a byte.
5. Write once to terminate the program.

A particular implementation may not support all types, and will produce an appropriate runtime error when that I/O type initialises. The program will continue and the wire will then be treated as broken.

### 9.3.2 Termination

The whole program will terminate once all boxes are terminated, where I/O can terminate (e.g. end of file), and individual boxes can terminate if they deadlock.

The whole program will terminate if two seconds elapse without any events occuring, this typically occurs while waiting for console input.

# Chapter 10

# humec

## 10.1 Overview

The humec (humec compiler) provides an alternative way of executing Hume programs by first compiling them to Ham (via phamc), and then using ham2c to produce C code that is further compiled to give an executable.

## 10.2 Usage

humec is a command line tool that takes a file and produces an executable. The tool has the following options:

`humec {options}* <infile>`
`-lotsaspace` *Compile with lots of space (heap=10000, stack=10000, wire=10000)*
`-dham` *Dump final internal ham format (as used by ham2c), the stop*
`-g` *Generate code with interactive debug support*
`--help` *Show this information*
*<infile> Specify the Hume/Ham/C file to compile*

FSM-Hume program should compile with no additional arguments, all other levels of Hume will require the `-lotsaspace` option to help avoid heap/stack/wire overflow.

```
% humec program.hume
...
% ls
program.hume program.ham humestubs.c program.c program*

% program
...
```

Note: Due to the reliance of humec upon phamc it may sometimes be necessary to edit the global STACKSIZE/HEAPSIZE constants in the intermediate C file, alternatively (and always necessary for wires overflow issues) the intermediate ham file may need to be edited.

## 10.3 Debugging

By compiling with the `-g` option, an executable is produced that allows stepping through the Ham opcodes and inspection of heap, stack, wire and box state.

### 10.3.1 Command Line debugging

At the most primitive level this debugging can run within the command line simply by running the compiled executable with the `-g` option.

```
% humec -g program.hume
...

% program -g
...
<<system>> hp=00000000 sp=0 slp=0 mp=0 fp=0 rp=0 schedules=0
% help
usage := step|boxstep|superstep|quit|run|help ...
```

### 10.3.2 GUI debugger

For easy visual debugging, the stand-alone debugger is first started and then the compiled executable is feed the address of the debugger to connect with.

```
% java -jar hamdb.jar
Run programs with -g48049 ...

% program -g48049
...
```

Upon running the compiled executable, the stand-alone debugger will open a window through which you can interact with the program.

## 10.4 humec Status

This section tries to give specific details on how the Hume compiler differs/conforms to the rest of the Hume report/manual. It describes only the back-end of the compiler, and omits information about how the front-end (phamc, and Hume parser) may behave.

### 10.4.1 Recent Changes

1 August 2006

- Added the `EndOfFile` exception and removed the '\0' magic character.
- Exceptions can be passed along wires, this allows I/O devices to throw exceptions in the reading box.

18 July 2006

- An unhandled exception will now terminate the program rather than just a single box.
- The user no longer needs to append custom types onto a file name (i.e. a hack) to support typed reading.
- Typed stream I/O is symmetrical – it will read back the same thing you write.
- Typed stream I/O will now read fixed length strings.
- Added `lshl, lshr` as builtin operators.

### 10.4.2 Types

- `bool`, `char`, `string`, `tuple`, `union`, `vector`, `exception` – supported.
- `int` – supported, limited to precision 32.
- `float` – supported, limited to precision 32.
- `word`, `nat` – supported by treated internally as `int`.
- `list` – supported by treating internally as a `union`.
- `unicode`, `fixed`, `exact` – unsupported.

**Notes:** Contrary to what the report says, a `string` is not a synonym for `list(char)` and using the typical list syntax on a string will most probably lead to a type error.

### 10.4.3 Functions

- Closures – are supported.
- Over/Under-evaluation – may work but is mostly untested.

### 10.4.4 Builtin Operators

The builtin operators are all supported with the current exception of `rotl, rotr`.

A few extra functions are provided or the operators are able to support more types:

```
any:    <, <=, ==, >=, !=,
        as string[1],
        write[2],
        unsafeAssignment[3]

int/nat/word:  +, -, *, **, unary -, div, mod,
        &, |, ^, ~, lshl, lshr, as float, as char

float:  +, -, *, **, unary -, /, as int
        sqrt, exp, ln, sin, cos, tan, asin, acos, atan,
        log10, sinh, cosh, tanh, atan2

bool:   &&, ||, not

vector: @, length, ++, update, vecdef, vecmap,
        unsafeUpdate[4]
```

```
string: @, length, ++,

list:   @, length, ++,

tuple:  @, length
```

Notes:

1. `as string::a->string` coerce uses the automagic stream output mechanism, but this has the consequence that there may be an extra trailing space.
2. `write::a->b->b` dumps the first arg to stderr and returns the second.
3. `unsafeAssignment::a->a->a` copies the first arg destructively into the second (implies they are the same size), it then returns the second.
4. `unsafeUpdate::Vec a->Int->a->Vec a` is the same as update, but does destructive update rather than returning a new vector.

Precision (under/overflow), and size constrains are not exhaustively checked in the results of the operations.

### 10.4.5 Boxes

- `fair/match` – supported.
- `handlers` – supported.

**Notes:** The RTS uses strict superstep scheduling as implied by the report.

**Known bugs:** The number of output wires on a box is limited to 15 (future versions of the compiler may increase this to 31).

When using the "expression" statement phamc will wrap it in a special box, this currently only allows one "expression" per program.

Currently a closure can't be sent along a wire – should this be allowed?

### 10.4.6 System Exceptions

- `StackOverflow`, `IndexOutOfBounds` – supported.
- `EndOfFile` – supported.
- `Underflow`, `Overflow` – supported, but currently are never raised.
- `HeapOverflow` – supported, but also distinguishes a special case with `WireOverflow`.
- `Div0` – renamed as `DivisionByZero`.
- `Timeout` – renamed and split up into `BoxWithin`, `BoxTimeout`, `WireWithin`, `WireTimeout`, but currently are never raised.
- `ExpressionWithin` – added as not explicitly named in the report.
- `MatchFail` – added for function pattern match.
- `InternalError` – added to cover any other (RTS) error, e.g. type error – in general should not occur in a correct program, only other know place is from malformed input stream data.

**Notes:** All system exceptions pass an empty tuple as the exception argument. The RTS internally also raises so-called `SoftStackOverflow` and `SoftHeapOverflow`, but these are invisible to the program (i.e. cannot be caught by a handler) and instead serve to generate warnings on the console.

### 10.4.7 Timeout/Within

**Notes:** The smallest unit of time is 1ns. During debugging time is effectively frozen so that the program behavior appears unchanged.

#### Expressions

- Time – supported (including nesting of), an unnamed exception will raise a `ExpressionWithin`.
- Heap – supported (including nesting of).
- Stack – supported (including nesting of).

#### Boxes

Within begins upon start of the evaluation of the box RHS, and stops at the end of this evaluation. The box pattern matching, exception handling, and writing of expression results to wires is all considered to be part of the RTS rather than the box itself.

Timeout/Within maximums are available via debugging/instrumentation, but are currently not used to raise exceptions.

#### Wires

Timeout/Within maximums are available via debugging/instrumentation, but are currently not used to raise exceptions.

**Known bugs:** The wire information collected is wrong.

### 10.4.8 Foreign Function Interfacing

- `ccall`, `stdcall` – supported.
- `cplusplus`, `jvm`, `dotnet` – unsupported.

Structures that can be exchanged to/from C from Hume include:

- simple – int/float/char/bool, fixed length strings.
- structure – vectors/tuples, any nesting of, and the above simple types.
- limited – simple type and variable length strings.

All the limited type values are converted to C equivalents. A structure type is passed as an opaque heap pointer, which may then be unpacked/packed to/from byte arrays via `void heapToBytes(uint8*, HEAP *)` and `HEAP *bytesToHeap(uint8*)`. Note that a structure type is a fixed size type, and that in this case a fixed length string is an array of chars with no terminating NULL.

### 10.4.9 I/O

| Variety: | Keywords: | Parameter: | Type: |
|---|---|---|---|
| console in[1] | stream from | "std_in" | <any>[2] |
| console out | stream to | "std_out" | <any> |
| console error | stream to | "std_err" | <any> |
| file in[1] | stream from | <filename> | <any>[2] |
| file out | stream to | <filename> | <any> |
| uri in[1] | stream from | <uri>[3] | <any>[2] |
| uri out | stream to | <uri>[3] | <any> |
| clock[4] | stream from | "clock" | int |
| interrupt[5] | interrupt from | <signalname>[6] | () |
| memory in | memory from | <address>[7] | int |
| memory out | memory to | <address>[7] | int |
| args | stream from | "main_args" | string |
| environment | stream from | "main_envs" | (string,string) |
| exit code[8] | stream to | "main_exit" | int |
| -unsupported- | port to/from | | |
| -unsupported- | fifo to/from | | |

**Notes:** A particular implementation/platform may not support all types, and will produce an appropriate runtime error when that I/O type initialises. The program will continue and the wire will then be treated as broken.

1. Console/file/uri input use a thread to read and connects via a wire, whereas all other I/O types are passive and are read/writen immediately on consume/assert to the wire.
2. Stream input only allows reading from fixed size data structures.
3. A uri may refer to local file via file::\//localhost..., a tcp socket via data::\//..., or create a single bind tcp local socket via bind::\//localhost...

   When using a bind socket the program will block in initilisation until a connection is made, thus behavior will be unpredictable if multiple bind sockets are used.

   By addressing a socket with the same address it is possible (and desirable) to steam to and from the same socket.
4. Clock is a timebase at ms resolution.
5. Quick successive interrupts may be merged.
6. It depends on the target architecture the set of address and signal names that are supported.
7. Adddress has syntax:

   ```
   <addr> := [0x <hexnum> | <decnum> | <name>[6] ] :: <type>
   <type> := [p8 | p16 | p32 | 0..7]
   ```

   Where the type part refers to the size of the integer, or 0..7 to refer to an indivdual bit in a byte.

   This is a temporary solution – some of the info could come from the type, but something is necessary in order to address a specific bit.
8. Write once to exit the program

**Known bugs:** Console/file/uri output is blocking.

### 10.4.10 Automagic Typed Stream I/O

**Output**

- `int` – output as decimal with a trailing space.
- `float` – output as decimal with a trailing space, guaranteed to be 8 characters or less by using scientific notation if necessary.
- `bool` – output as "true" or "false" with a trailing space.
- `char`, `string` – output in it's native form.
- `tuple`, `vector` – traversed depth first.
- `union` – traversed depth first, the tag is ignored in terms of outputing.
- `closure` – shouldn't occur, but would result in an `InternalError`

**Known bugs:** The last digit of the float may not obey rounding rules. Print a floating larger than 1e38 may not terminate?!?

**Input**

Driven by the type signature.

- `int` – skips any leading white space, supports decimal only, skips one trailing white space.
- `float` – skips any leading white space, supports decimal including scientific, skips one trailing white space.
- `bool` – skips any leading white space, accepts only "true" or "false", skips one trailing white space.
- `char` – reads the current character.
- `string` (fixed length) – reads the exact number of characters.
- `tuple`, `vector` – reads the exact number of items.
- `union`, `closure` – not possible.

**Notes:** raises `EndOFile` at end of file, and `InternalError` on malformed input data.

### 10.4.11 Debugging

Debug support can be optionally compiled into a program (via the `-g` flag) along with additional instrumentation (as enabled via editing the RTS `"am.h"` file).

`profile` and `verify` are currently unsupported (but are on the todo list).

# Chapter 11

# 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" <intconst1> "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"
```

## 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
        | <varid "@" <patt>             alias variable

<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>
<spaceconst>  ::= <intconst><spacedes>
<timedes>     ::= "ps" | "ns" | "us" | "ms" | "s" | "min"
<spacedes>    ::= "B" | "KB" | "MB"
<char>        ::= "A" | ... | "Z" | " " | "\t" | "\n" | "\\" |
                  "0x" <hexdigit> +
```

## Hume Functions and Operators

Operations on *int p* types

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

Operations on *nat p* types

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

Operations on *word p* types

```
+, - :: Word -> Nat -> Word
==, !=, <=, <, >, >= :: Word -> Word -> Bool
rotl, rotr, lshl, lshr :: Word -> Nat -> Word
```

Operations on *float p* types

```
+, -, *, /, ** :: Float -> Float -> Float
sin, cos, tan, asin, acos, atan, log, exp, sqrt,
  ln, log10, sinh, cosh, tanh :: Float -> Float
atan2 :: Float -> Float -> Float
==, !=, <=, <, >, >= :: Float -> Float -> bool
```

Operations on vector types

```
length :: <<a>> -> Int
@ :: <<a> -> Int -> a
vecdef :: Int -> (Int->a) -> <<a>>
vecmap :: <<a>> -> (a->b) -> <<b>>
```

```
vecfoldr :: <<a>> -> b -> (a->b->b) -> b
update :: <<a>> -> Int -> a -> <<a>>
==, !=, <=, <, >, >= :: <<a>> -> <<a>> -> Bool
```

Operations on tuple types

```
length :: (a1, .., an) -> Int
@ :: (a1, .., an) -> Int -> ax
==, !=, <=, <, >, >= :: (a1, .., an) -> (a1, ..., an) -> Bool
```

Operations on list types

```
length :: [a] -> Int
@ :: [a] -> Int -> a
++ :: [a] -> [a] -> [a]
hd :: [a] -> a
tl :: [a] -> [a]
==, !=, <=, <, >, >= :: [a] -> [a] -> Bool
```

Operations on *string p* types

```
length :: String -> Int
@ :: String -> Int -> a
++ :: String -> String -> String
==, !=, <=, <, >, >= :: String -> String -> Bool
```