PROGRAMMING IN HUME
Collectors Edition...

Greg Michaelson and Kevin Hammond

Contents

Prologue

This work in progress has not been checked or proof read and is undoubtedly full of myriad mistakes, misconceptions and muddles.

Caveat emptor...

Greg Michaelson and Kevin Hammond, 21st August 2009.

Chapter 1

Introduction

1.1 Overview

Hume[7] is a modern programming language with strong foundations both in classical automata[14] and computability theory[4] and in contemporary functional languages like Standard ML[13] and Haskell[5]. Hume is explicitly oriented to developing systems requiring strong guarantees of well-bounded use of resources, such as execution time, memory or power, through static analyses of programs prior to execution rather than through run-time monitoring and instrumentation. To meet these needs, the Hume design makes an explicit distinction between *coordination*, concerned with managing how program components interact with each other and the environment, and *computation*, concerned with the activities carried out by individual components. This separation of concerns enables us to offer, in a principled manner, different *levels* of language providing different degrees of expressive power and precision of static resource analyses.

So, that's one story. Another story is that...

1.2 What's wrong with functional programming?

Once upon a time, well early in 2000, we had finished co-editing a substantial collection of invited contributions on parallel functional programming[6]. Why, we mused, had our favourite paradigm, functional programming, singularly failed to have any discernable impact outside academia, despite its undoubted elegance and formality? Perhaps it is precisely that elegance and formality that puts people off. Perhaps functional languages are all mouth and no trousers: full of promise of bridging theory and practicality, but just too difficult to deploy without thorough understanding of both. Certainly, many people used to a grounded von Neumann world, of assignment and iteration over concrete structures of concrete types, are nonplussed by deep hierarchies of abstractions more reminiscent of algebra than algorithms. And, while it may be that reasoning about and proving properties of functional programs are easier than for

imperative programs, they aren't actually that much easier beyond toy examples. In particular, full strength functional languages share all the undecidability properties of other Turing Complete languages[4], so fully automatic analysis of program properties is a non-starter and apparently tractable heuristics just don't scale to real world problems.

1.3 How about primitive recursion?

So, we thought, why not start with a language somewhat weaker than Turing Complete, for which interesting properties are decideable? For example, primitive recursion[11] has decideable termination and so might be a good starting point for determining time and space behaviour. Alas, it is undecidable whether or not an arbitrary program in a Turing Complete language is primitive recursive.

There have been attempts at crafting purely or partly primitive recursive languages, like Burstall's Inductively Defined Functions[3] or Turner's Elementary Strong Functional Programming[16]. Alas, these languages feel "unnaturally" restricted by syntax and types, and lack clear programming methodologies for coping with the absence of familiar properties like unbounded loops or arbitrary depth structures.

A contrasting technique is to constrain programs to always satisfying primitive recursive properties. Thus, Martin-Löf's type theory forms the basis of a constructive methodology for systematically refining a specification into a correct program[1]. Similarly, the recursion editor[2] supports the development of programs from templates that guarantee termination. However, both approaches also greatly constrain the familiar free form style of programming as programs must satisfy strong correctness and/or structural criteria at all stages of development.

1.4 How about finite state automata?

Our next candidate, finite state automata(FSA)[9] seemed like a more promising basis for developing a language with strongly analytic properties. FSA have decidable termination, which would provide firm foundations for time and space analyses. Furthermore, FSA have decidable equivalence and unique, well determined minimal forms, which would offer a good basis for program transformation and optimisation.

Because of their close relationshop with formal languages, FSA and their refinements have long been the basis for language transducers, for example in Unix pattern matching in tools like `grep` and `vi`, and in parser generators like `lex/yacc` and `JavaCC`. Furthermore, FSA formalisms and reasoning techniques underpin model checking which is increasingly used to establish properties of safety critical systems. For example, the Spin model checker complements the Promela language[8].

FSA have recently found new currency in software engineering. Thus: FSA-like state charts are key components of designs in the Unified Modelling Language (UML)[15]; state models may be used to design concurrent systems[12]; state charts may be used to derive graphical user intefaces[10].

More generally, FSA are widely used by engineers to design both mechanical and electrical control systems. If you mention functional programming or primitive recursion to many engineers their eyes glaze over, but mention FSA and they immediately have a handle on what you're talking about.

However, while FSA simplicity is a major strength for their use in design, it greatly compromises their usefulness as a basis for programming languages. As FSA lack core abstractions for types and control structures, constructs that would be taken for granted in higher level languages must be crafted explicitly. Thus, models of realistic systems in FSA languages like Promela quickly become large and complicated. Furthermore, in the worst case, establishing formal properties of FSA requires explicit analysis of all possible transition paths, which grow exponentialy in number for linear growth of FSA states and arcs.

So we began to think about how we could base a programming language in FSA but still enable the use of advanced type and control abstraction of functional programming languages.

1.5 From FSA to Hume

Formally, a FSA is a recogniser for symbol sequences corresponding to a regular expression. At simplest, an FSA has a set of states linked by arcs labeled by symbols. In general, in some old state, a FSA inspects the next input symbol and changes to the state for which there is a connecting arc labelled with that symbol. This process is called a *transition*. One state is designated the start. There may be multiple final states corresponding to successful or failed recognition of symbol sequences. Note that there is no output during transition: a basic FSA can only indicate its final state. For example, Figure 1.1 shows a FSA to check whether a sequence of bits ending with a `!` has odd or even parity. A simple FSA can be expressed as a set of transition rules of the form:

$$(oldstate, symbol) \rightarrow newstate$$

so the example is:

```
(START,1) -> ODD
(START,0) -> EVEN
(EVEN,0) -> EVEN
(EVEN,1) -> ODD
(ODD,0) -> ODD
(ODD,1) -> EVEN
(ODD,!) -> STOP ODD
(EVEN,!) -> STOP EVEN
```

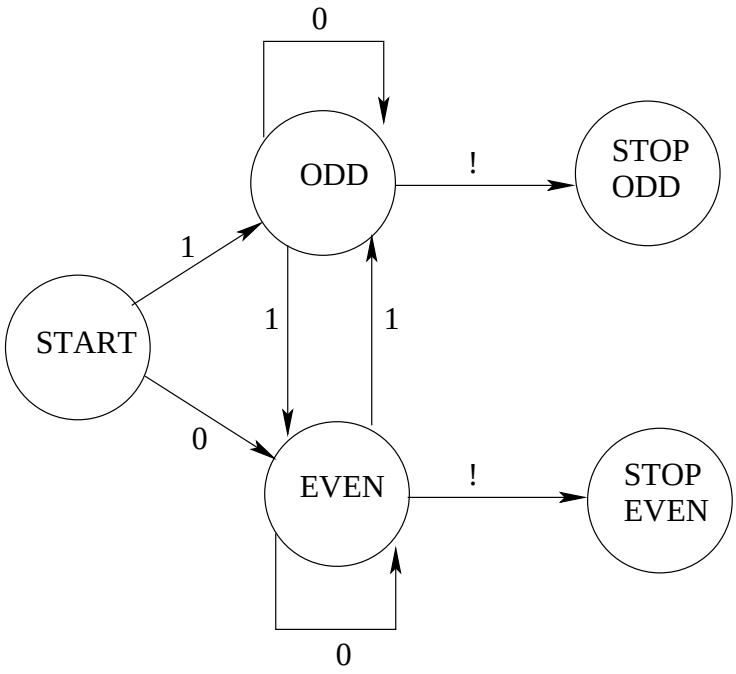

Figure 1.1: FSA for parity checking.

There are a number of points to notice about this state machine. First of all, it runs continuously, consuming inputs. But it is not apparent where the inputs come from. Secondly, every time it consumes an input it changes state. But the state change is implicit in the notation. We might redraw the figure to make these facits explicit - see Figure 1.2 Now we show links from the outside environment to supply the input source and to feed the new state from one transition to the old state of the next transition.

Refinements on this basic FSA include the Moore machine, which generates an output on entering each new state, and the most general Mealey machine which generates an output on each transition. For the Mealey machine, we can extend the transition form to:

$$(oldstate,symbol) \rightarrow (newstate,output)$$

Our example, extended with explicit output, becomes:

```
(START,1) -> (ODD,Odd)
(START,0) -> (EVEN,Even)
(EVEN,0) -> (EVEN,Even)
(EVEN,1) -> (ODD,Odd)
(ODD,0) -> (ODD,Odd)
(ODD,1) -> (EVEN,Even)
```

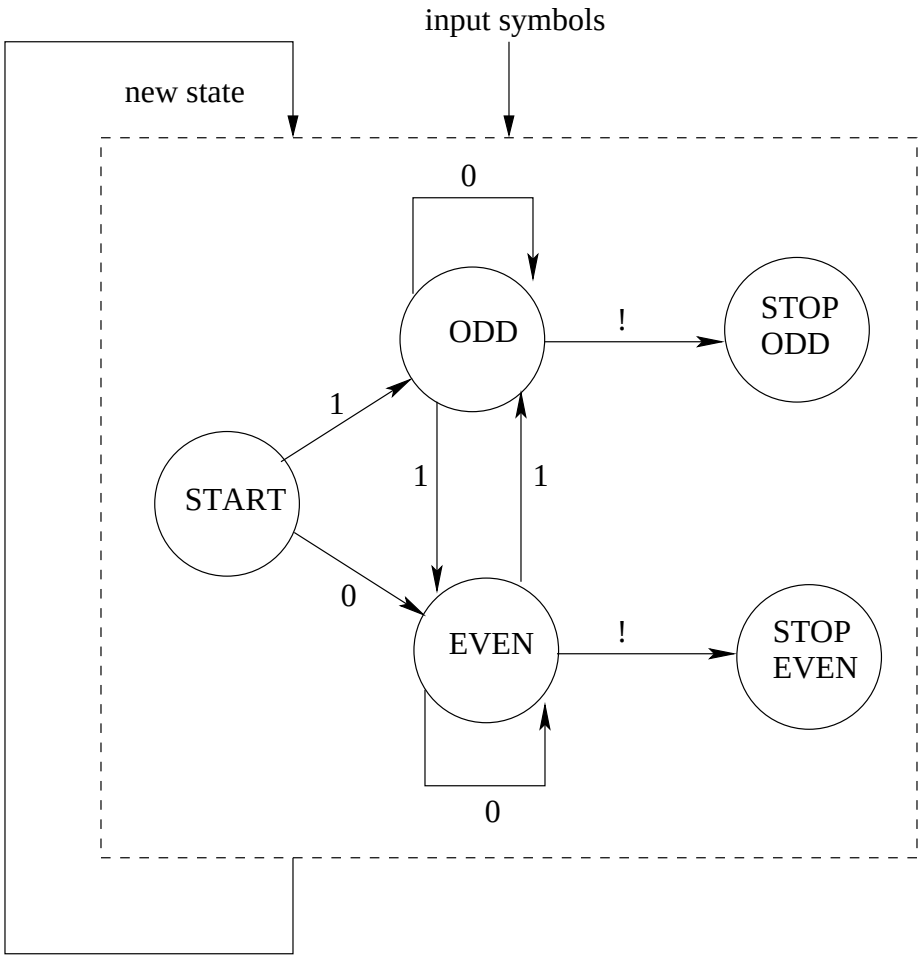

Figure 1.2: FSA for parity checking with explicit input and state.

```
(ODD,!) -> (STOP ODD,Stopd odd)
(EVEN,!) -> (STOP EVEN,Stop even)
```

with diagram shown in Figure 1.3.

1.6 The essence of Hume

Now, thinking like good functional programmers, the left hand side of a Mealey machine transition looks like a tuple pattern and the right hand side looks like a tuple expression. So why not generalise the transition form to:

pattern → *expression*

Here we have the essence of Hume. We no longer distinguish the old state and input symbol, or the new state and output symbol, though these remain special cases. Instead, we have a much more powerful form as shown in Figure 1.4, which we call a *box*, with links which we call *wires*. Inside the box is a sequence of transitions from *pattern*s to *expression*s, where both are support a rich range of polymorphic types. Inputs may come from external sources, such as files or devices or other boxes, or from the box itself. Similarly, outputs may go to external sinks, such as files or devices or other boxes, or to the box itself. Thus, the state has withered away and is encompassed by a a box wired to itself. Now, on each execution cycle, inputs are *matched* against patterns, with a successful match triggering the corresponding expression to generate the outputs. Thus, a Hume *program* consists of a number of boxes, wired to each other and to the external environment, repeatedly executing concurrently.

1.7 Example: parity checking

Let's now write the parity checker in Hume, ignoring the terminating states for the time being:

```
type bit = word 1;

data STATE = ODD | EVEN;

stream input from "std_in";
stream output to "std_out";

box parity
in (s::STATE,symb::bit)
out (s'::STATE,message::string)
match
 (ODD,0) -> (ODD,"Odd\n") |
 (ODD,1) -> (EVEN,"Even\n") |
 (EVEN,0) -> (EVEN,"Even\n") |
```

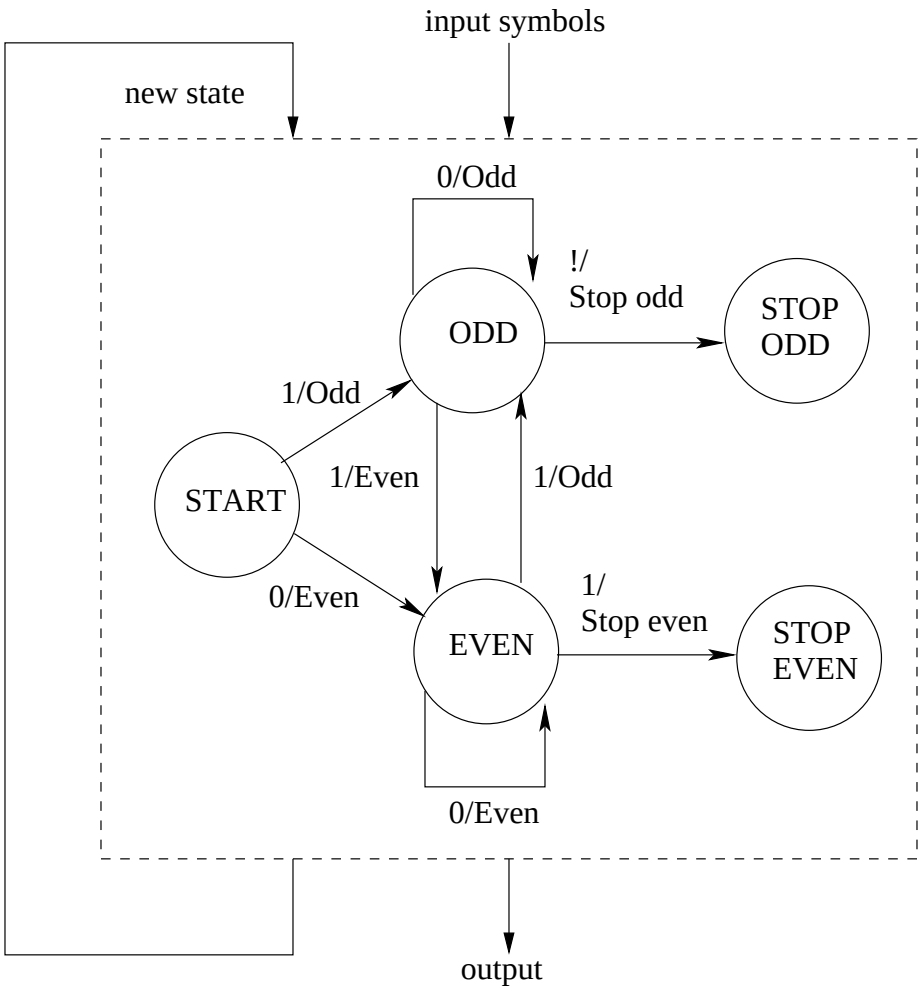

Figure 1.3: Mealey machine for parity checking with explicit input, output and state.

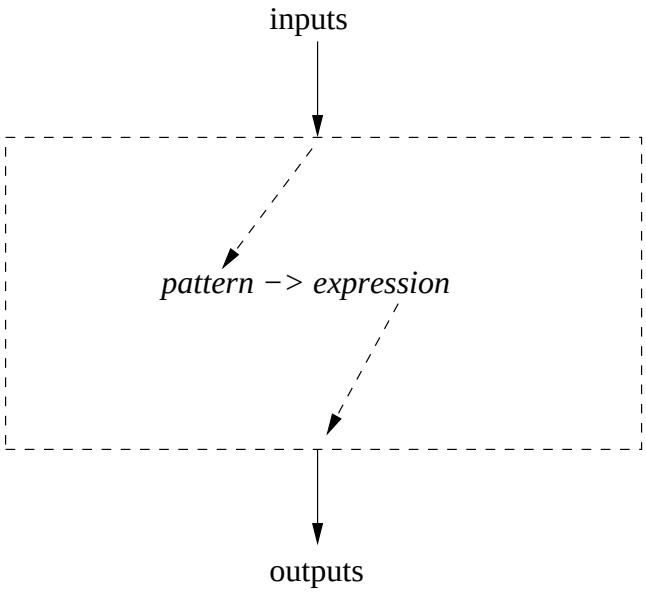

Figure 1.4: Generalised Mealey machine: the box.

```
 (EVEN,1) -> (ODD,"Odd\n");

wire parity
(parity.s' initially EVEN,input)
(parity.s,output);
```

First of all we have auxilliary declaration:

- `type bit = word 1;` - defines `bit` to be a synonym for a one bit word;
- `data STATE = ODD | EVEN` - defines a constructed type `data` which can have the values `ODD` or `EVEN`;
- `stream input from "std_in"` - defines an stream `input` associated with standard (i.e. keyboard) input;
- `stream output to "std_out"` - defines a stream `output` associated with standard (i.e. display) output.

Next we have the declaration of the box `parity` consitituting the program. We can discern:

- `box parity` - names the box;
- `in (s::STATE,symb::bit)` - identifies two inputs: `s` that will receive the current state and `symb` that will receive the next input symbol;

- `out (s'::STATE,output::string)` - identifies two output: `s'` that will produce the next state and `message` that will produce a string;

There follow the four transitions corresponding to the four states of the original FSA. For example, we can read the first transition:

```
(ODD,0) -> (ODD,"Odd\n")
```

as saying: if the input state is `ODD` and the input symbol is `0` then output the state `ODD` and the message `"Odd\n"`.

Finally, in the wiring:

- `wire parity` - identifies which box is being wired;
- `(parity.s' initially EVEN,input)` - says that:
  - `parity`'s input `s` is wired to `parity`'s output `s'` and has initial value `EVEN`;
  - `parity`'s input `symb` is wired to the stream `input`
- `(parity.s,output)` - says that:
  - `parity`'s output `s'` is wired to `parity`'s output `s`;
  - `parity`'s ouput `message` is wired to the stream `output`

This first example provides a simple flavour of Hume programming. In the next few chapters, we explore in considerably more detail what goes on both inside and outside boxes. We then turn to how the Hume design decisions enable us to meet our original objective of providing a language suitable for resource aware systems development.

1.8 Conventions

The Hume Report[7] contains the definitive, if flawed and evolving, account of Hume.

In this book, text is in Times Roman, *things we want to emphasise are in italic Times Roman*, and `code is in Courier`.

Syntactic classes are also in italic Times Roman. Syntax class names may not always be the same as those in the formal definition of Hume[7] but the correspondence should be straight forward.

Footnotes marked *TP* are *Trivial Pursuits*: engaging points of vacuous contention between the authors.

Acknowledgements

Chapter 2

Preliminaries

2.1 Introduction

In this chapter we are going to explore the underpining concepts and constructs for Hume programming, in particular program structure, box construction and wiring. We will also start to look at Hume types, expressions and patterns.

2.2 Program structure

A Hume program has three conceptual components:

- *box declarations*, specifying the I/O and transition properties of boxes;
- *wire declarations*, specifying how boxes are connected to each other and the wider environment;
- *auxilliary declarations* of types, constants, functions, streams amd so on used in box and wire declarations.

While elements of these may appear in any order, as with other languages, it makes sense to group related elements together. In particular, it is usual to start a program with all common auxilliary declarations, and for wire declarations to follow box declarations [1].

2.2.1 Identifiers

Hume identifiers are sequences of underbars (`_`), letters and digits, starting with an underbar or letter. For example:

```
hUmE _Hume_dispels_doubt H12345_is_not_very_informative_and_2_long
```

Note that a single underbar is used for the wildcard pattern.

[1] TP1: should each wire declaration immediately follow the associated box declaration or should all wire declarations be grouped together after all box declarations?

2.2.2 Box declaration

At simplest, a box declaration takes the form:

```
box identifier
in (identifier::type ...)
out (identifier::type ...)
match transitions;
```

The `in` names and gives types to a comma separated sequence of inputs, and the `out` names and gives types to the outputs.

All box identifiers must be unique.

All `in` and `out` identifiers must be unique to the box.

The *transitions* is a sequence of *pattern* `->` *expression* transitions separated by `|`. All *pattern*s must have the same type as a tuple of the input wire types. All *expressions* must have the same type as a tuple of the output wire types.

2.2.3 Box wiring

Boxes may be wired implicitly *by position* or explicitly *point to point*[2].

For wiring by position, lists are provided of links to other boxes and the enviroment, which are implicitly associated with the corresponding box inputs and outputs. Wiring by position takes the form:

```
wire identifier (inputlinks) (outputlinks);
```

The *identifier* names the box being wired. The *inputlinks* lists where the box's inputs are to be connected, and may be either box outputs or output streams, discussed below. Similarly, the *outputlinks* lists where the box's outputs are to be connected, either box inputs or input streams, also discussed below. The key notion is that the *inputlinks* should correspond in position and type with the associated box's inputs, and the *outputlinks* in position and type with the associated box's outputs.

A link to a box input or output has the form:

box − *identifier*.*in*/*out* − *identifier*

An input link may be optionally followed by:

```
initially value
```

to specify some initial *value* on the wire.

A link to a stream is the stream's identifier.

With point to point wiring, a link may be specified explicitly. This takes the form:

```
wire link to link;
```

[2] TP2: is by position or point to point wiring preferable?

Any wiring by position has an equivalent list of point to point wirings. However, point to point wires are initialised separately:

`initial` *box − identifier* {*in/out − identifier* `=` *value* `...`)};

where the *box−identifier* is followed by a comma separated list of initialisations.

We will discuss wiring macros in a later chapter once we have met multi-box programs.

2.2.4 Stream

A stream provides a connection to the operating environment enabling access to devices, files, sockets and so on.

Streams communicate sequences of characters to and from boxes. The system uses *automagic I/O* to ensure that values are converted appropriately.

An input stream is defined as:

`stream` *identifier* `from` *string*`;`

and an output stream by:

`stream` *identifier* `to` *string*`;`

where the *string* is a double quoted file or socket path of an appropriate format for the host system.

Standard input, typically from a keyboard, is denoted by `"std_in"` and standard output, typically to a display, by `"std_in"`.

2.3 Types

Hume has a rich polymorphic type system in the functional tradition. A key innovation is the explicit distinction between *sized* and *unsized* types, which greatly facilitates resource analysis. Sized types, including booleans, characters, words, integers, floats, tuples, vectors and enumerated discriminated union, are of stated precision or size. In contrast, unsized types, including strings, lists and recursive discriminated union, may be of arbitrary precision or size.

As we shall hint at in this chapter, and discuss in the next, Hume is novel in providing implicit *automagic I/O* of most sized types.

2.3.1 Simple types

Booleans

Booleans have the type constructor `bool` and values `true` and `false`.

Characters

Characters have the type constructor `char` and values of the form:

'*character*'

where *character* is a single symbol, for example:

```
'a' '9' '@' ')'
```

or a standard escaped symbol, for example:

'\n' - newline
'\t' - tab
'\'' s ingle quote

Words

Words are fixed sized unsigned bit sequences. They have the type constructor:

`word` *size*

where *size* denotes a fixed number of bits, for example:

```
word 64
```

Words values are hexadecimal:

`0x` *hexdigits*

for example:

```
0xFFFF0001
```

Integers

Integers are fixed sized, twos complement signed. They have the type constructor:

`int` *size*

as for `word`. Integer values are decimal[3], for example:

```
42351 (-147)
```

Floats

Floats are fixed sized and signed. They have the type constructor:

`float` *size*

as for `word`. Float values may be floating point or exponential[4], for example:

```
41.4243 44.45e46 4748e-49 (-50e51)
```

[3] TP3: negative integers need to be bracketed
[4] TP4: see TP3

Enumerated discriminated unions

Enumerated discriminated unions provide a way for users to define new types with a fixed number of values. They are defined by:

`data` *constructor* `=` $identifier_1$ `|` `...` `|` $identifier_i$`;`

There after, the values $identifier_1 ... identifier_N$ all have type *constructor*. For example, after:

```
data LIGHTS = RED | RED_AMBER | AMBER | GREEN;
```

`RED`, `RED_AMBER`, `AMBER` and `GREEN` all have type `LIGHTS`.

Note that Hume cannot input or output dicriminated unions.

2.3.2 Structured types

We will here consider tuples and strings. We will discuss lists and structured discriminated types types later.

Tuples

Tuples are fixed length sequences of arbitrary type. A tuple of types $type_1 ... type_N$ itself has type:

`(`$type_1$`,...,`$type_N$`)`

Tuples may be arbitrarily nested. For example:

```
(true,'a') ::  (bool,char)
(('b',false),'1') ::  ((char,bool),char)
```

Strings

Arbitrary sized strings have the type constructor `string`. Sized strings have constructor `string` *size*.

String values are arbitrary length sequences of symbols within double quotes:

`"`$symb_1 \ldots symb_N$`"`

using the same escape conventions as characters. For example:

```
"Hume dispels doubt\n"
```

Vectors

Vectors are fixed sequences of the same type. Vector types take the form:

`vector` *constant* `of` *type*

where *constant* is a natural number. For this type, values take the form:

<<$value_1$,...,$value_N$>>

where $value_i$ is of type *type* and N is the *constant*. For example:

```
<<'a','b','c'>> ::  vector 3 of char
<<('1',true),('2',false)>> ::  vector 2 of (char,bool)
<<<<false,false>>,<<false,true>>,
  <<true,false>>,<<true,true>>>> ::
   vector 3 of vector 2 of bool
```

2.3.3 Type aliases

Type aliases of the form:

type *identifier* = *type*;

may be used to provide simplified and meaningful identifiers. The subsequent use of *identifier* in any context where a type may appear is equivalent to the in situ substitution of *type*. For example, we will extensively use:

```
type integer = int 64;
```

in the rest of this book.

2.4 Patterns

Recall that boxes are composed of sequences of transitions of the form:

pattern -> *expresion*

On each box execution cycle, as discussed in more detail below, input wires are *matched* against successive transition patterns, with a successful match triggering evaluation of the associated expression.

In general, a pattern is a tuple with one element for each input wire. In turn, each element must be consistent with the type and denotation structure of the corresponding wire's type.

Pattern elements look like value denotations but may also have *variable* identifiers in place of arbitrary value components. For the types we've met so far, a pattern element may be:

- a constant boolean, word, integer, character, sized string or enumerated discriminated union value;
- a variable;
- the wild card _;
- a tuple of pattern elements;
- a vector of pattern elements

`<`	less than
`<=`	less than or equal
`==`	equal
`>=`	greater than or equal
`>`	greater than
`!=`	not equal

Figure 2.1: Comparison operators.

Note that patterns may not contain floats or unsized strings: float equality is not well defined; strings of arbitrary size cannot be distinguished in inputs.

Patterns elements are matched left to right with the values on the corresponding wires. Here we will assume that every wire has a value: we will consider patterns where wires may be empty in another chapter.

For a pattern element match with a wire value to succeed, it must correspond in structure with the value on the corresponding wire. That is, if the pattern element is a:

- constant, the corresponding wire value must be the same;
- variable, the match succeeds *and the variable is bound to the corresponding wire value*;
- wildcard, the match succeeds;
- tuple or vector, the elements are matched with the corresponding elements of the tuple or vector on the wire;

After a successful match, the variables in the pattern are bound to the corresponding values from the wires for use in the associated expression.

2.5 Expressions

We will next quickly survey the operations and expressions for the types we have met so far. We will meet all of these repeatedly in the rest of this book.

2.5.1 Comparison

The comparison operators are shown in Figure 2.1. All are infix, take two arguments of the same type and return a `bool`.

These operators apply directly to simple types, and recursively, left to right, to structured types. Character based constructs are compared in alphabetic order. For example:

```
"cat" < "catamaran" ⇒ true
("a",1,1.0) >= ("a",2,1.0) ⇒ false
```

operator	action	precedence
`||`	infix disjunction (or)	3
`&&`	infix conjunction (and)	2
`not`	prefix negation	1

Figure 2.2: Boolean operators.

operator	action	precendence
`+`	infix addition	3
`-`	infix subtraction	3
`*`	infix multiplication	2
`div`	infix division	2
`mod`	infix remainder	2
`**`	infix power	2
`-`	prefix negation	1

Figure 2.3: Integer operators.

2.5.2 Boolean expressions

Boolean expressions may be constructed using the operators shown in Figure 2.2. They may be structured with arbitrarily nested bracket sub-expressions in `(...)`.

2.5.3 Integer expressions

The integer arithmetic operators are shown in Figure 2.3. All take one or more integer argument of the same size and return an integer of the same size.

Integer expressions may be structured using `(...)`.

2.5.4 Float expressions

The float arithmetic operators are shown in Figure 2.4. All take one or more float arguments of the same size and return a float of the same size.

There are also the customary trigonometric and other infix operators shown in Figure 2.5.

operator	action	precendence
`+`	infix addition	3
`-`	infix subtraction	3
`**`	infix power	2
`*`	infix multiplication	2
`/`	infix division	2
`-`	prefix negation	1

Figure 2.4: Float operators.

operator	action
`sin`	sine
`cos`	cosine
`tan`	tangent
`asin`	inverse sine
`acos`	inverse cosine
`atan`	inverse tangent
`sinh`	hyperbolic sine
`cosh`	hyperbolic cosine
`tanh`	hyperbolic tangent
`atan2`	angle of point $a1/a2$ to x axis
`log`	natural logarithm
`log10`	logarithm base 10
`exp`	exponent
`sqrt`	square root

Figure 2.5: Float trigonometric and other operators.

operator	action	precendence
`+`	infix addition	3
`-`	infix subtraction	3
`*`	infix multiplication	2
`div`	infix division	2
`mod`	infix remainder	2
`**`	infix power	2
`-`	prefix negation	1

Figure 2.6: Word operators.

Float expressions may be structured using `(...)`.

2.5.5 Word expressions

The word arithmetic operators are shown in Figure 2.6. All take one or more word arguments of the same size and return a word of the same size.

The infix word manipulation operators are shown in Figure 2.7.

The bitwise word operators are shown in Figure 2.8.

Word expressions may be structured using `(...)`.

2.5.6 String expressions

String elements are selected with infix `@`:

```
"hello"@3 ⇒ 'l'
```

Strings are joined with infix `++`:

operator	action
`lshl`	logical shift left
`lshr`	logical shift right
`ashl`	arithmetic shift left
`ashr`	arithmetic shift right
`rotl`	rotate left
`rotr`	rotate right
`bittest`	for word $a1$ test bit $a2$
`bitset`	for word $a1$ set bit $a2$
`bitclr`	for word $a1$ clear bit $a2$

Figure 2.7: Word manipulation operators.

operator	action
`^&`	and
`^|`	or
`^`	not and

Figure 2.8: Word bitwise operators.

```
"hello"++" "++"there" ⇒ "hello there"
```

The length of string is found with infix `length`:

```
length "hello" ⇒ 5
```

2.5.7 Vector expressions

The infix operator v`@`i selects element i of vector v:

```
<<2,4,6,8,10>>@3 ⇒ 6
```

The prefix operator `length` returns the number of elements in a vector.

The prefix operator `update` v i e makes a copy[5] of vector v with element i replaced with the value of expression e:

```
update <<1,2,3,4,5>> 3 33 ⇒ <<1,2,33,4,5>>
```

We will consider operators for creating and mapping over vectors after we have met functions in the next chapter.

2.5.8 Type casting and coercion

While many types share the same lexical form of operators, Hume insists that this is merely a convenience and that each form actually denotes a different operation depending on the type context. Furthermore, binary operators only take operands of the same type so implict mixed type operations are not permitted.

[5] TP5: details of destructive update are available on request

Thus, where mixed type operand are required at least one must be converted explicitly to be consistent with the other.

There are type forms of type conversion, *casting* and *coercion*.

Casting takes the form:

expression `::` *type*

Here the value of *expression* will be converted to *type* provided it may be accomplished without loss of information and at no run-time cost.

The more general[6] coercion takes the form:

expression `as` *type*

Here, there may be both a loss of information or a run-time cost.

It is usually possible to coerce an arbitrary non-function type to string and thence to another non-function type. This may become clearer when we discuss automatic input/output in the next chapter.

2.5.9 Putting it all together

To summarise, an arbitrary expression may be:

- a base constant;
- a variable;
- a tuple or vector where each element is an expression;
- a conversion expression;
- a prefix operator applied to an appropriate expression;
- an infix operator applied to appropriate operand expressions;
- a bracketed expression.

We will, of course, see lots of concrete instances of these in the rest of this book. We will also meet functions, lists and recursive discriminated unions in the next chapter.

2.6 Program execution

So, now we have a program made up of named boxes wired to each other and to named streams connected to the operating environment. Each box has named and typed inputs and outputs, and a sequence of transitions from patterns over inputs to expressions over outputs.

[6] TP6: and more useful?

To begin with, every box is potentially executable (*RUNNABLE*) and all initial values are on the wires. Note that the order of box execution is not specified[7].

Execution is carried out in two stages. In the first pattern match stage, each box attempts to match the values on its inputs against each transition pattern in turn until one match succeeds. For now, we assume that matching always starts with the first transition and proceeds in sequence through the transitions. Such matching is termed *unfair*.

After a successful match, variables in the pattern are bound to corresponding input values and the associated transition expression is then evaluated to produce the output values. At the end of this stage, all local memory allocated to the pattern and expression is reclaimed. However, no inputs are consumed from wires and no outputs are asserted to wires.

Once all boxes have completed the first stage, in the second *super-step* stage, wire transactions are resolved. For each box, values on successfully matched wires are removed. Then, if all of the output wires for which new values have been generated are empty then each wire is set to the coresponding output value and the box is RUNNABLE for the next execution cycle.

However, if at least one output wire still has unconsummed values, then no output values are asserted for that box. Instead, the new output values are held until the next cycle. Such a box is said to be *BLOCKED-OUTPUT*.

On all subsequent cycles, only RUNNABLE boxes are eligible for pattern matching, but both succesfully matched and BLOCKED-OUTPUT boxes are considered in the super-step.

2.7 Comments

A comment is a line starting with `--`. For a multi-line comment, every line must be so marked.

2.8 Importing files

A directive:

`import` *path*

will insert the file at *path*, along with nested `import`s, prior to parsing.

Recursive `import`s are ignored.

2.9 Standard prelude

We will assume a minimal standard prelude:

[7] TP7: though in some implementations it is (possibly reverse) alphabetic order of box name

```
type integer = int 64;
stream input from "std_in";
stream output to "std_out;
```

in a file `prelude.hume` and that every subsequent program begins with:

```
import prelude.hume
```

We will sometimes redefine these definitions in situ as needed.

2.10 Summary

In this somewhat dense and jumbled chapter, we have met almost all of the key ideas behind Hume. In the next few chapters, we will begin to systematically explore Hume programming, starting with single boxes of one transition before developing single boxes with multiple transitions and multiple box programs. As well as considering further Hume constructs such as functions, lists, structured discriminated unions and exceptions, we will also discuss Hume programming techniques, in particular how to optimise what goes on inside and outside of boxes. Finally, we will look at more elaborate ways of connecting to the operating enviroment through sockets and foreign functions.

Chapter 3

One box, one transition

3.1 Starting with nothing...

In this chapter we are just going to look at Hume programs that consist of a single box with a single transition, with output to standard output and, usually, input from standard input[1].

Let's begin with a program that accepts single integers and outputs them, one at a time:

```
box oneint
in (n::integer)
out (n'::integer)
match
 n -> n;

wire oneint (input) (output);
```

Note that, while we use unique identifiers for inputs and outputs, it is purely "coincidental" if a pattern contains an identifier which also names an input or output[2]; the name spaces are disjoint.

Note in the transition the special cases of a single variable input and of a non-tuple output, which are not bracketed[3].

When we compile and run this program:

```
$ humec -lotsaspace oneint.hume
...
$ oneint
...
```

[1] TP8: given arbitrary recursive expressions, this form, is of course Turing Complete.

[2] TP9: there is a plausible convention of pattern identifiers always reflecting associated input/output identifiers but that seems over prescriptive

[3] TP10: we should probably have used different brackets to distinguish a sequence of patterns for multiple input wires and a tuple pattern for a single wire.

```
0
0 1
1 2
2
```

it will repeatedly read successive integers from the keyboard and write them to the display.

Note that the program "knows" to expect an integer from standard input on the input wire, without any explicit invocation of input. Similarly, the program "knows" to display the integer on standard output without any explicit invocation of output. As we explore below, Hume has strong automagic I/O requiring no user formatting of input or output.

Having input and output on the same line without identification is untidy, so we will modify the program to prompt for input and display it on a separate line. To do this, we change the expression to a tuple of the integer followed by a newline and prompt. We must also change the output type:

```
box oneint2
in (n::integer)
out (n'::(integer,string))
match
 n -> (n,"\nNext> ");
```

Now, when we run the program:

```
1
1
Next>2
2
Next>3
3
Next>
```

we are prompted for input every time except the first.

Note that Hume automatically converts the tuple on the output into appropriate character representations for the elements separated by a space. This is quite unlike other languages where values must either be converted explicitly for textual display or the systems reconstructs the full syntactic value denotation.

Let us again change the program to print the integer and its square, again by modifying the output type and the transition expression:

```
box intsq
in (n::integer)
out (n'::(integer,string))
match
 n -> (n,n*n,"\nNext> ");
```

The output is now:

```
...
Next> 2
2 4
Next> 3
3 9
Next>
```

Now let us prompt for two integers and output the sum of their squares in a more informative format:

```
box sumsq
in (n::(integer,integer))
out (n'::(string,integer,string,integer,string,integer,string))
match
 (x,y) -> ("x:",x,"y:",y,"x*x+y*y:",x*x+y*y,"\nEnter x y> ");
```

with interaction:

```
Enter x y> 7 8
x:7 y:8 x*x+y*y:113
Enter x y>
```

The input `n` is now from a single wire carrying a tuple of two values, matched in the pattern by `x` and `y`. Once again, the system will automatically recognise appropriate values for the tuple elements from the input stream character sequence.

As always, the output type faithfully reflects the type of the expression tuple. However, an alternative here is to simplify the output type to a single `string` and then cast the tuple expression[4]:

```
box sumsq2
in (n::(integer,integer))
out (n'::string)
match
 (x,y) -> ("x:",x,"y:",y,"x*x+y*y:",x*x+y*y,"\nEnter x y> ") as string;
```

Note that Hume's polymorphic type system will, in this instance, ensure that the expression is type correct even though the output no longer corresponds in type and so provides no directly confirming type information. That is, `x` and `y` must be `integer` because `n` is a tuple of `integer` so `*` must be `integer` multiplication and `+` must be `integer` addition (and consistent with `integer 1`).

3.2 Simple functions

Hume provides functions as expression abstractions. At simplest, a function has the same form as a box transition:

[4] TP11: if we'd used different brackets for a tuple expression and a transition expression feeding mutiple wires then the cast wouldn't be necessary here.

identifier pattern `=` *expression*`;`

The *identifier* names the function. *pattern* is termed the formal parameter and *expression* the body. Variables in the *pattern* have scope and extent in the *expression*.

If *pattern* has type $type_1$ and *expression* has type $type_2$ then this function has type:

$type_1 \rightarrow type_2$

implying that *identifier* is a mapping from a $type_1$ to a $type_2$.

For example:

```
inc x = x+1;
```

is the integer increment function with type:

```
int 64 -> int 64
```

Note again that Hume can deduce that `x` is `integer`: `1` is `integer` so `+` must be `integer`.

Where the type of *pattern* variables cannot be deduced in context, the function type must be defined explicitly. For example, in the square function:

```
sq x = x*x;
```

`*` is overloaded so `x` does not have an unambiguous type. To ensure `x` is integer requires:

```
sq :: integer -> integer;
sq x = x*x;
```

[5]:

identifier `::` *type*`;`

Thus, we might have characterised `sq` by:

```
sq :: integer;
sq x = x*x;
```

A simple function call has the form:

identifier $expression_1$

$expression_1$ is termed the actual parameter. Function application is strict and left to right. If:

identifier :: $type_1 \rightarrow type_2$

[5] TP12: One author, from the Haskell tradition, thinks uniform explicit function typing is the best style as it aids software development and documentation. The other author, from the SML tradition if lazy, prefers to trust polymorphic type inference to tell him function types.

and $expression_1$ evaluates to:

$value_1$ :: $type_1$

then the *pattern* is matched against $value_1$, binding any variables in *pattern* to the corresponding elements of *value*, and the function body *expression* is evaluated to return some:

$value_2$ :: $type_2$

Function calls are used in expressions, and have higher precedence than infix operators. For example, we may rewrite the core of the sum of squares program as:

```
box sumsq3
in (n::(integer,integer))
out (n'::integer)
match
 (x,y) -> sq x+sq y;
```

Brackets must be deployed round actual parameters which are infix expressions or function applications. Thus:

`sq 3+4` $\Rightarrow$ `13`

but:

`sq (3+4)` $\Rightarrow$ `49`

Tuples may be used to get the effect of multi-argument functions. For example:

```
sumsq (x,y) = sq x+sq y;
```

Thus, we could further rewrite sum of squares as:

```
box sumsq4
in (n::(integer,integer))
out (n'::integer)
match
 (x,y) -> sumsq (x,y);
```

The infix operator `o` is used to compose functions so:

$indentifier_1$`(`$identifier_2$`(`$expression$`))` $\Leftrightarrow$
`(`$identifier_1$ `o` $identifier_2$`)` $expression$

For example:

`inc (sq (3))` $\leftrightarrow$ `(inc o sq) 3`

Functions may also be defined in Curried form:

identifier `::` *type*$_1$ `->` *type*$_2$ `->` `...` `->` *type*$_N$
identifier *pattern*$_1$ *pattern*$_2$ `...` `=` *expression*

where *pattern*$_i$ has *type*$_i$. Here, function application has the form:

identifier *expression*$_1$ *expression*$_2$ ...

where *expression*$_i$ has *type*$_i$. From left to right, each *expression*$_i$ is evaluated and its value matched against *pattern*$_i$.

This is broadly equivalent to the un-Curried form:

identifier `::` `(`*type*$_1$`,`*type*$_2$ `...)` `->` *type*$_N$
identifier `(`*pattern*$_1$`,`*pattern*$_2$ `...)` `=` *expression*

For example:

```
sumsq x y = sq x+sq y ...  sumsq 3 4 ⇔
sumsq (x,y) = sq x+sq y ...  sumsq (3,4)
```

The Curried form is retained for aesthetic reasons but is not particularly useful as Hume does not support anonymous functions or partial application[6].

3.3 Self wiring and state

So far we have met very simple boxes that endlessly consume inputs and generate outputs. Each execution cycle is distinct and no information is held between cycles. However, by wiring a box to itself, information may be retained from one cycle to the next.

In general, a self-wired box has the form:

```
box identifier
in (identifier1::type1,identifier2::type2 ...)
out (identifier'1::type1,identifier3::type3 ...)
match transitions;

wire identifier
(identifier.identifier1 initially value, ...)
(identifier.identifier'1, ...);
```

illustrated in Figure 3.1.

Input *identifier*$_1$ and output *identifier*$'_1$ are wired reflexively to each other, and necessarily have the same type *type*$_1$. By convention, self-wired inputs and outputs share the same identifier.

Note that such wiring must have an initial value. This is a very common ommission in Hume programs, leading to the box blocking right from the start.

Now, on each cycle, the *pattern* will match the current wire value from input *identifier*$_1$, *expression* will generate a value for output *identifier*$'_1$, which will be available from input *identfier*$_1$ on the next cycle.

[6] TP13: this seemingly tiresome restriction aids resource analyses

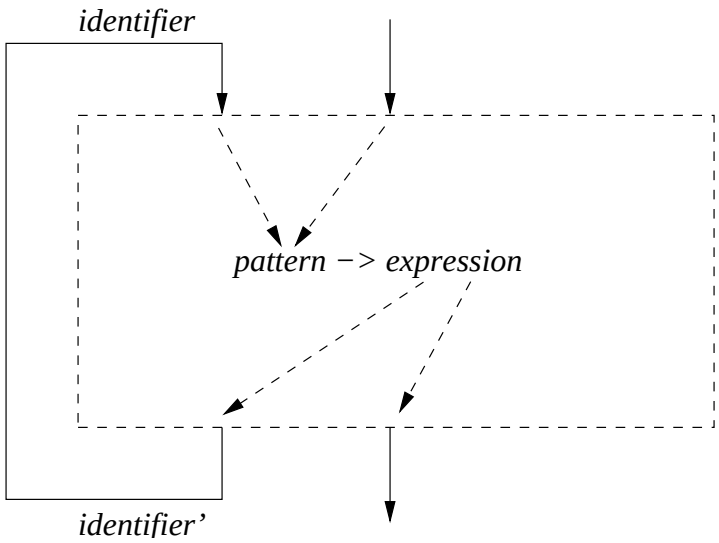

Figure 3.1: Self-wiring.

For example, consider a Hume program that maintains and displays a running total of input values:

```
box total
in (t,n::integer)
out (t'::integer,tot::(integer,string))
match
 (t,n) -> (t+n,(t+n,"\nNext> "));

wire total
(total.t' initially 0, input)
(total.t, output);
```

which runs as:

```
1
1
Next> 2
3
Next> 3
6
Next>
```

On each cycle, `t` matches the old running total, initially `0`, and `n` the next input. Then, `n is` added to `t` which is both passed to `n'` and displayed.

Note the short form in the input declaration for several inputs of the same type:

identifier$_1$,*identifier*$_2$, ... :: *type*

For example, consider the extremely useful generation box which produces a sequence of successive integers:

```
box gen
in (n::integer)
out(n',r::integer)
match
 n -> (n+1,n);

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

Here, successive values are held on the `n` to `n'` wire, and returned on `r`:

```
0 1 2 3 4 5 6 7 8 9 ...
```

Note that the box has no input and so the self-wire must be initialised.

Self-wiring of boxes is equivalent to accumulation variables in recursive functions. This observation underlies the linear recursive function to iterative box transformation discussed in a later chapter.

3.4 Local definition

In the `total` program, we calculate `t+n` twice. We can simplify this by using a local definition of the form:

`let` *identifier* `=` *expression*$_1$
`in` *expression*$_2$

Local definitions are used to introduce local variables to hold intermediate results. Thus, *identifier* is bound to the value of *expression*$_1$ throughout *expression*$_2$.

For example, we could rewrite `total` as:

```
box total2
in (t,n::integer)
out (t'::integer,tot::(integer,string))
match
 (t,n) -> let s = t+n
          in (s,(s,"\nNext> "));
```

Now, both references to `s` in `(s,(s,"\nNext> "))` will share the same value from `t+n`.

Note that only an *identifier*, and not an arbitrary *pattern*, may be used in a let definition[7]. Thus, to match tuple results from a function it is necessary to compose its call with another function whose formal parameter is an appropriate tuple pattern. For example, we might naturally attempt to write:

[7] TP14: See TP13.

```
sumdiff :: (integer,integer) -> (integer,integer)
sumdiff (x,y) = (x+y,x-y);
...
let (x,y) = sumdiff(p,q) in (y,x)
...
```

but instead must craft:

```
swap :: (integer,integer) -> (integer,integer);
swap (a,b) = (b,a);
...
swap(sumdiff(p,q))
...
```

3.5 Creating and mapping over vectors

The prefix operator `vecdef` n f creates a vector of length n where element i is function f applied to i:

`vecdef 5 sq` $\Rightarrow$ `<<1,4,9,16,25>>`

The prefix operator `vecmap` v f creates a new vector from function f applied to each element of vector v:

`vecmap <<1,2,3,4,5>> inc` $\Rightarrow$ `<<2,3,4,5,6>>`

3.6 `expression`

When developing functions, it is often useful to be able to test them independently of the boxes in which they're going to be used. Hume provides the "expression command" of the form:

`expression` *expression*`;`

which may appear arbitrarily amongst declarations.

At run time, the *expression* is evaluated and the value is displayed.

3.7 Automagic I/O

I/O provision is a common and long-standing weakness of programming language design. Many languages have no intrinsic I/O, which instead is provided through libraries with no standardisation. Furthermore, where I/O is part of the language it often feels like an afterthought. Typically, there is inconsistent orthogonality of I/O, often with structured types, and indeed some base types, not supported at all. A widely observed consequence is that I/O requires a disproportionate amount of effort in software development, distracting from a primary focus on how inputs are turned into outputs.

output :: *type* → *value* → *text*
output [`word` *size*] w = showword *size* w
output [`int` *size*] i = showint *size* i
output [`float` *size*] r = showfloat *size* r
output [`bool`] b = showbool b
output [`char`] c = showchar c
output [`string`] s = output [`[char]`] s
output [`[` *type* `]`] `[`e_1...e_n`]` =
    output *type* e_1++" "++ ... output *type* e_n
output [`(` $type_1$...$type_n$`)`] `(`e_1...e_n`)` =
    output $type_1$ e_1++" "++ ... output $type_n$ e_n
output [`vector` *size* `of` *type*] `<<`e_1...e_{size}`>>` =
    output *type* e_1++" "++ ... output *type* e_{size}

where 'showword', 'showint', 'showfloat', 'showbool' and 'showchar' are appropriate base type output functions, and '++' is a text concatenation operator.

Figure 3.2: Output by type.

Hume supports what is colloquially termed *automagic I/O*, where, as far as possible, there are implicit conversions to and from textual and internal representations of data values. This is driven entirely by type information in context without any programmer intervention.

Hume follows the imperative rather than the functional tradition in that values are represented as flat sequences of base values rather than in the syntactic denotation form. Denotation I/O may be useful once values have been constructed, for example as a basis for program tracing or simple implementation independent persistent data. Ultimately, however, all values that are not constructed within a program through abstracted denotations must result from the conversion of raw external textual input. It is inconvenient to have to prepare input data in syntactic denotation form, especially if that data has been already generated by other programs. It is also tiresome to read syntactic denotations of complex nested structures, especially if one is not familiar with the language in which they originate. Thus, we decided that Hume I/O would be based on flat textual representations.

3.7.1 Output

In Hume, all denotable values except discriminated union types have output representations. Generation of output is driven by the structure of the type of the value to be output, and may be defined recursively as shown in Figure 3.2.

Note that arbitrary sized structure values have non-unique text representations. For example, the text sequence:

input :: $type \rightarrow text \rightarrow value * text$
input [`word` $size$] t = getword $size$ t
input [`int` $size$] t = getint $size$ t
input [`float` $size$] t = getfloat $size$ t
input [`bool`] t = getbool t
input [`char`] t = getchar t
input [`(` $type_1$...$type_n$`)` `t` =
    let (e1,t1) = input $type_1$ t
     ...
        (en,tn) = input $type_n$ t_{n-1}
    in ((e1...en),tn)
input [`vector` $size$ `of` $type$] t =
    let (e1,t1) = input $type$ t
     ...
        (e_{size},t_{size}) = input $type$ t_{size}
    in (`<<`e1...e_{size}`>>`,t_{size})

where 'getword', 'getint', 'getfloat', 'getbool' and 'getchar' are appropriate base type input functions which return values of the required type and the text that follows their representation in the input.

Figure 3.3: Input by type.

```
1 2 3 4 5 6 7 8
```

represents the vectors:

```
<<1,2,3,4,5,6,7,8>> ::  vector 8 of integer;
<<<<1,2,3,4,5,6,7,8,9>>>> ::  vector 1 of vector 8 of integer
<<<<1,2,3,4>>,<<5,6,7,8>>>> ::  vector 2 of vector 4 of integer
<<<<1,2>>,<<3,4>>,<<5,6>>,<<7,8>>>> ::
  vector 4 of vector 2 of integer
```

amongst others.

3.7.2 Input

Hume input is restricted to fixed size types, and may be defined recursively on a required value's type signature as shown in figure 3.3.

This restriction is required because of the ambiguity of text representations discussed above: it is not possible to identify a unique value for an unsized type from a flat textual representation.

However, one benefit of the Hume approach is that the shape of structures may be changed transparently through I/O, or type conversion, precisely be-

cause of the non-uniqueness of representation. For example, a fixed length sequence of bits:

```
1011001010010010
```

may be input as vectors of varying size and dimension, of words of varying sizes:

```
<<1011,0010,1001,0010>> ::  vector 4 of word 4
<<10110010,10010010>> ::  vector 2 of word 8
<<<<1011,0010>>,
  <<1001,0010>>>> ::  vector 2 of vector 2 of word 4
<<<<1,0,1,1>>,
  <<0,0,1,0>>,
  <<1,0,0,1>>,
  <<0,0,1,0>>>> :  vector 4 of vector 4 of word 1
```

3.7.3 Limitations

The big benefit of automagic I/O is for input of uniform sized data from a continuous stream. However, input of non-uniform data or data sequences of finite but unknown length is curiously problematic.

For example, there is no easy way to input an arbitrary length sequence of characters ending with a newline. Instead, a sized vector of characters must be used to input sequences which are always of the same length. The vector is then coerced to string.

For example, inputing a data size followed by that quantity of data requires an explicit state machine with different phases for acquiring the size and the data. We will see an example of this in a subsequent chapter when we consider file I/O.

Otherwise, for data sequences with separator markers, indeed for any free form data, it is necessary to deploy a state machine that acquire a sequence of characters as one big string, followed by explicit parsing either through recursive functions or further state machines.

Chapter 4

One box, multiple transitions

4.1 Introduction

In the last chapter we looked at the construction of basic patterns and expressions for a box with just one transition. We will now look at more elaborate patterns and expressions, in particular for recursive data types, for a box with multiple transitions.

4.2 Multiple transitions

Our single transition boxes are unable to distinguish amongst different combinations of input values, for which multiple tarnsitions are required. In general, a box match has the form:

```
match
  pattern1 -> expression1 |
  pattern2 -> expression2 |
...
```

All the patterns must have the same type as the inputs. All the expressions must have the same type as the outputs.

Transitions do not need to be different. Repeated transitions, combined with fair matching discussed below, may be used to bias the match.

As noted in an earlier chapter, for this *unfair* matching, matching always starts with $pattern_1$ and proceeds pattern by pattern through the sequence of transitions. Collectively, the patterns should exhaustively cover the space of input values. This does not imply that one pattern must always succeed though: a match may fail if one input is not available at the point of match.

For example, consider inputing simple expressions of the form:

x	y	carry	sum	carry'
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Figure 4.1: Full adder.

integer operator integer

where the *operator* may be `+`, `-`, `*` or `/`:

```
box arith
in (e::(integer,char,integer))
out (r::(integer,string))
match
 (i1,'+',i2) -> (i1+i2,"\n") |
 (i1,'-',i2) -> (i1-i2,"\n") |
 (i1,'*',i2) -> (i1*i2,"\n") |
 (i1,'/',i2) -> (i1 div i2,"\n") |
 (_,op,_) ->  (0,"bad operator\n");
```

which runs as:

```
...
1 + 1
2
3 * 8
24
6 % 7
0 bad operator
...
```

We can directly read off a truth table as a Hume box, with one transitionfor each table row. For example, the full adder in Figure 4.1 becomes:

```
box adder
in (xyc::(bit,bit,bit))
out (sc::(bit,bit))
match
 (0,0,0) -> (0,0) |
 (0,1,0) -> (1,0) |
 (1,0,0) -> (1,0) |
 (1,1,0) -> (0,1) |
```

```
(0,0,1) -> (1,0) |
(0,1,1) -> (0,1) |
(1,0,1) -> (0,1) |
(1,1,1) -> (1,1);
```

4.3 Exceptions

Exceptions may be raised explicitly by an expression or by the system in response to some aberrant condition.

User defined exceptions are declared by:

`exception` *identifier type*;

and raised within expressions by:

`raise` *identifier value*

where the exception argument *value* is of *type*.

Exceptions are handled by augmenting a box declaration to identify both which exceptions the box accepts and how they are dealt with:

```
box ...
in ...
out ...
handles (identifier1,identifier2...)
match ...
handle
  identifier1 pattern1 -> expression1 |
  identifier2 pattern2 -> expression2 |
...
```

Thus, `handles` introduces recognised exceptions and `handle` says what to do with them. Here, $pattern_i$ must be of the requisite type for exception $identifier_i$.

When a handled exception has been raised, the pattern is matched with the exception argument and the value of the corresponding expression is returned. A raised but unhandled exception causes the program to fall over.

A box will only handle exceptions raised in its own matches. Thus, exceptions cannot be used for time-warp style inter-box communication.

For example, if we run the previous program:

```
...
22 / 0
Division By Zero
$
```

the `Div0` system exception is raised and the program stops.

We can handle this exception with:

```
box arith
in (e::(integer,char,integer))
out (r::(integer,string))
handles Div0
match
 (i1,'+',i2) -> (i1+i2,"\n") |
 (i1,'-',i2) -> (i1-i2,"\n") |
 (i1,'*',i2) -> (i1*i2,"\n") |
 (i1,'/',i2) -> (i1 div i2,"\n") |
 (_,op,_) ->  (0,"bad operator\n")
handle Div0 _ -> (0,"divide by zero\n");
```

for example:

```
...
33 / 0
0 divide by zero
33 / 3
11
...
```

Note that any data on self-wires is not accessible to an exception handler and should be assumed to be lost after the exceptions. An exception has no pattern to match a self-wire input. Furthermore an exception must generate some value for a self-wire output[1].

System exceptions are described in the Hume manual[2].

4.4 Star pattern and expression

The *star pattern* `*` will *ignore* the corresponding input. Thus a star pattern will *always succeed* even if the input is empty, and will not consume a non-empty input.

Similarly, the complementary *star expression*, also `*`, will produce a valid *empty* output.

The star pattern may only be used at the top level to match a whole wire. That is, star patterns may not be nested in structured patterns.

Similarly, the star expression may only be used to produce output for an entire tuple. That is, star expressions may not be nested in structured expressions. However, star expressions may appear at the top level of the body of a function, provided that ultimately produces an empty output for a whole wire. Note that `*` is a state of being not a value and so cannot be passed around, say as an argument.

[1] TP14: but what happens if an exception occurs before inputs are consumed on the superstep and asserts `*` on the self wire?

[2] TP15: It is not entirely well defined what system exceptions return. Possibilities include a string or the empty tuple.

Note that these constructs, which are central to Hume's versatility, mark an important point of break with the functional tradition where all functions are expected to be total.

We will make considerable use of these forms in the rest of this book.

4.5 Box iteration

A common Hume trope is a box which is self wired to repeat some activity until some condition is met:

```
box iterate
in (init::type1,accum::type2)
out (accum'::type2,result::type3)
match
 (*,base_pattern) -> (*,base_expression) |
 (*,iterate_pattern) -> (iterate_expression,*) |
 (initial_pattern,*) -> (initial_expression,*);

wire iterate (input,iterate.accum') (iterate.accum,output);
```

Note the input `accum` and output `accum'` which are wired to each other to accumulate a partial result.

The iteration starts with the third pattern which accepts an initial value from `input`, ignores the accumulation wire, passes the input value suitably modified to the accumulation wire and generates no output.

The iteration continues with the second pattern which ignores the input, further processes the accumulated value and again generates no output.

Finally, the terminating first pattern ignores the input, does not produce an accumulation value and outputs the final value.

For example, consider summing all the values from 1 to some input integer. In some imperative language we would have something like:

```
READ(N);
I := N;
S := 0;
WHILE I>0 DO
BEGIN
     SUM := SUM + I;
     I := I-1
END
WRITE(S)
```

In Hume we use the iterative form:

```
box sum
in (n::integer,a::(integer,integer))
out (a'::(integer,integer),r::integer)
```

```
match
 (*,(0,s)) -> (*,s) |
 (*,(i,s)) -> ((i-1,s+i),*) |
 (n,*) -> ((n,0),*) ;

wire sum (input,sum.a') (sum.a,output);
```

Starting with the second transition, we ignore the input, pick up the accumulated count so far (`i`) and sum so far (`s`), decrement the count and add it to the sum, and generate no output. Then, the last transition picks up the input and initialises the count and sum, while the first transition returns the final sum.

Note that the order of transitions is crucial. If we started with the input transition, it would succeed every time there was a new input, restarting the computation. And if we started with the iterating transition we would never recognise the termination case.

4.6 Tuple on wire or many wires?

We could have written `sum` with two self-wires each carrying a single value instead of one self-wire carrying a tuple of two values:

```
box sum2
in (n::integer,c,t::integer)
out (c',t'::integer,r::integer)
match
 (*,0,s) -> (*,*,s) |
 (*,i,s) -> (i-1,s+i,*) |
 (n,*,*) -> (n,0,*) ;

wire sum2 (input,sum2.c',sum2.t') (sum2.c,sum2.t,output);
```

This is computationally equivalent to `sum` and arguably easier to read. It is also less efficient as more wires must be maintained and matched.

4.7 Recursive functions

Recursive functions are a natural form in Hume:

$$\begin{array}{l} identifier\ pattern_{11}\ pattern_{12}\ \ldots\ \texttt{=}\ expression_1\texttt{;} \\ identifier\ pattern_{21}\ pattern_{22}\ \ldots\ \texttt{=}\ expression_2\texttt{;} \\ \ldots \end{array}$$

Patterns in corresponding positions in each case must have the same type. Expressions must have the same type.

For example, to sum the integers from 1 to `n`:

```
sum 0 = 0;
sum n = n+sum (n-1);
```

4.8 Recursion and iteration

We can now see that the general iterative box above has the equivalent recursive function:

```
identifier ::  type1 -> type2;
identifier base_pattern = base_expression;
identifier iterate_pattern = iterate_expression;
```

which may be embeded in a box:

```
box iterate
in (n::type1)
out (n'::type2)
match
 n -> identifier n;

wire iterate (...)  (...)
```

For example, we could rewrite the summation program as:

```
box sum3
in (n::integer)
out (n'::integer)
match
 n -> sum n;

wire sum3 (input) (output);
```

which is clearly considerably simpler.

However, the choice between a recursive function and an iterative box is not straightforward. A recursive form may consume more local memory if it is not tail recursive. The iterative box may be more memory efficient but must be scheduled for every iteration where the box with recursion is only scheduled once. However, if the recursion is costly then, in a multi-box program, this may delay execution of other boxes. Finally, there may be recursive forms for which it is hard to establish useful resource bounds.

Arguably, recursive functions should be used to prototype a program, which is then refined to the iterative form:

- if memory use is problematic;
- to ease balancing a multi box system;
- if resource bounds of adequate precision cannot be found;

4.9 Structured discriminated union

The discriminate union also extends naturally to user defined structures. First of all, the right hand side elements may also include constructor designated typed sub-elements:

`data` *constructor* `= ... |` *identifier* $type_1$ $type_2$ `... | ...;`

For example, we might define a mixed arithmetic type by:

```
type real = float 64;
data ARITH = INT integer | REAL real;
```

`ARITH` is a new type constructor with values `INT` *integer* and `REAL` *real*, for example:

```
INT 42 :: ARITH
REAL 42.42 :: ARITH
```

Structured discriminated union patterns may be used in transition matches and function declarations. For example, we can write a single function to perform both integer and real addittion:

```
add (INT i1) (INT i2) = INT (i1+i2);
add (REAL r1) (REAL r2) = REAL (r1+r2);
```

so:

```
add (INT 1) (INT 2) ⇒ INT 3
add (REAL 1.1) (REAL 2.2) ⇒ REAL 3.3
```

We can also define recursive discrimnated unions where the new constructor itself appears as a sub-element type. For example, we might define arbitrary length sequences of integers:

```
data INTSEQ = IEND | INEXT integer INTSEQ;
```

with values like:

```
INEXT 1 (INEXT 2 (INEXT 3 IEND)) :: INTSEQ
```

For example, to sum an `INTSEQ`:

```
sumintseq IEND = 0;
sumintseq (INEXT i is) = i+sumintseq is;
```

Discriminate unions may be parameterised:

`data` *constructor* $identifier_1$ $identifier_2$ `... =`
`... |` *identifier* $type_1$ $type_2$ `... | ...;`

constructor then has parameteric polymorphism: the parameters $identifier_i$ may be referred to in the sub-element *type*s and insantiated with arbitrary types.

For example, we may generalise `INTSEQ` to a sequence of arbitrary type:

```
data SEQ a = END | NEXT a (SEQ a);
```

construct sequences of `SEQ` for any type `a`:

```
NEXT "a" (NEXT "b" (NEXT "c" END)) :: SEQ string
NEXT (1,1.0) (NEXT (2,2.0) (NEXT (3,3.0) END))) :: SEQ (integer,real)
```

and explicitly specialise `SEQ` to a concrete type:

```
type INTSEQ = SEQ integer;
```

4.10 Lists

The `SEQ` type discussed above is, of course, a linked list. Hume provides lists as a standard type. A list of arbitary *type* has type:

`[`*type*`]`

The empty (null) list is `[]`.

Lists are constructed with the infix operator `:`, where the left operand (head) is of some type and the right operand (tail) is a list of the same type. For example:

```
1:(2:(3:(4:(5:[])))) :: [integer]
```

A null terminated list:

$element_1$`:(`$element_2$`(:...:(`$element_s N$`:[])...))`

may be represented in the simplified form:

`[`$element_1$`,`$element_2$`,...,`$element_N$`]`

for example:

```
[1,2,3,4,5] :: [integer]
```

Note that the singleton `[1]` is the null terminated list `1:[]`.

List elements are selected through pattern matching. Both forms may be used in patterns. For example, to check if a string list is in ascending order:

```
sorder [] = true;
sorder [_] = true;
sorder  (s1:(s2:t)) =  s1 <= s2 && sorder (s2:t);
```

Note that `s1` matches the list head, `s2` the head of the tail, and `t` the tail of the tail.

Lists of the same type may be appended end to end:

```
append [] l2 = l2;
append (h1:t1) t2 = h1:append t1 t2;
```

with the infix operator `++`.

4.11 Conditional expression

Transition and function pattern matching can only determine whether or not some value is present on an input or an argument: it cannot check other value properties. Where it is not possible to distinguish properties by pattern matching, a conditional expression may be used. This has the form:

`if` *expression*$_1$ `then` *expression*$_2$ `else` *expression*$_3$

where *expression*$_1$ returns a boolean, and *expression*$_2$ and *expression*$_3$ return the same type.

For example, to generate an ascending sequence of integers:

```
gen i n = if i<=n then i:gen (i+1) n else [];
```

Note that, in boxes, the conditional expression can only be used in expression evaluation after transition pattern matching, that is after a commitment has been made to consume inputs[3].

4.12 Example: sieve of Erastothenes

Consider finding generating all the prime numbers using the sieve of Erastothenes. We maintain a list `p` of all the primes we've found so far, starting with `[2]`, and generate successive integers `n`, starting with `3`. Then if none of the values in `p` divides the next integer `n`:

```
isprime _ [] = true;
isprime n (h:t) = if n mod h == 0 then false else isprime n t;
```

we add it to the end of the list and display it. Either way, we generate the next integer:

```
box sieve
in (n::integer,p::[integer])
out (n'::integer,p'::[integer],next::(integer,char))
match
 (n,p) -> if isprime n p
          then (n+1,p++[n],(n,'\n'))
          else (n+1,p,*);

wire sieve
(sieve.n' initially 3,sieve.p' initially [2])
(sieve.n,sieve.p,output);
```

Here we maintain the next integer and known primes on wires (`n` to `n'` and `p` to `p'`) between execution cycles. For a change, we have used separate self-wires rather than a single self-wire of a tuple.

[3] TP16: perhaps Hume need guarded match patterns

Chapter 5

Many boxes

5.1 Introduction

We've now reached the point where we can consider building multi box programs. It might seem that, like any other software system, we can identify the components, encapsulate them accordingly and stick them all together. However, multi-box programs are inherently *concurrent*: despite Hume's abstract aura of evaluation order independence and super step synchronisation, we still have the very concrete considerations of sequencing, race conditions, blocking, starvation, live lock, deadlock and so on. However, as with all programming, such problems can be minimised by careful design, and systematic construction and testing.

5.2 Multiple boxes

Let's start by building a half adder corresponding to the truth table in Figure 5.1, but using even lower level components as shown in Figure 5.2.

We receive a pair of bits on the input, fan them out to separate XOR and AND boxes, and join the resultant sum and carry:

```
box fanout
in (xy::(bit,bit))
out (xy1,xy2::(bit,bit))
```

x	y	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 5.1: Half adder.

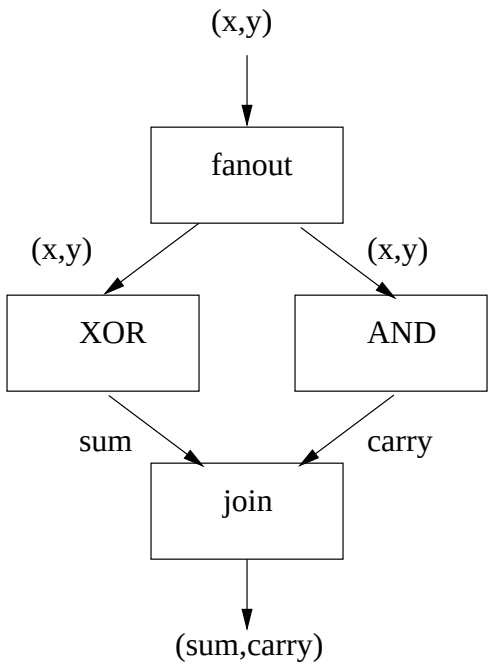

Figure 5.2: Half adder as XOR and AND.

```
match
 (x,y) -> ((x,y),(x,y));

box XOR
in (xy::(bit,bit))
out (sum::bit)
match
 (0,0) -> 0 |
 (0,1) -> 1 |
 (1,0) -> 1 |
 (1,1) -> 0;

box AND
in (xy::(bit,bit))
out (carry::bit)
match
 (0,0) -> 0 |
 (0,1) -> 0 |
 (1,0) -> 0 |
 (1,1) -> 1;
```

cycle	box	input	output
1.	`fanout`	$(x,y)_1$	$(x,y)_1$ and $(x,y)_1$
	`XOR`	MATCHFAIL	-
	`AND`	MATCHFAIL	-
	`join`	MATCHFAIL	-
2.	`fanout`	$(x,y)_2$	$(x,y)_2$ and $(x,y)_2$
	`XOR`	$(x,y)_1$	sum_1
	`AND`	$(x,y)_1$	$carry_1$
	`join`	MATCHFAIL	-
3.	`fanout`	$(x,y)_3$	$(x,y)_3$ and $(x,y)_3$
	`XOR`	$(x,y)_2$	sum_2
	`AND`	$(x,y)_2$	$carry_2$
	`join`	sum_1, $carry_1$	$(sum, carry)_1$

Figure 5.3: Half adder execution cycles.

```
box join
in (sum,carry::bit)
out (sc::(bit,bit))
match
 (s,c) -> (s,c);

wire fanout (input) (XOR.xy,AND.xy);
wire XOR (fanout.xy1) (join.sum);
wire AND (fanout.xy2) (join.carry);
wire join (XOR.sum,AND.carry) (output);
```

Now it is important to note that this is actually a three stage pipe line which takes three execution cycles to fully process an initial input, as shown in Figure 5.3.

5.3 Example: vending machine

Consider the vending machine cash handler design shown in Figure 5.4.

The *vending machine* itself keeps track of credit and the current purchase price, and requests change checks and coin release from the change mechanism. The *change mechanism* keeps track of coins and dispenses change.

Figure 5.5 shows the main vending machine in more detail. The *state* self-wire indicates I/O or change mechanism interaction. The *credit and price* self-wire keeps track of the customer's credit and the prices of items. The *code* wire from the input indicates whether the customer has inserted cash, requested a purchase or wants change. The *action and amount* wire goes to change mechanism which returns information on the *response and change* wire. Finally, the *message* is to the output.

Figure 5.6 shows the change mechanism in more detail.

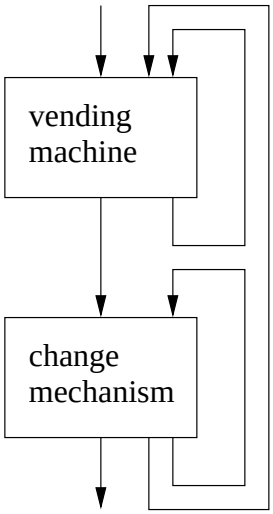

Figure 5.4: Vending machine cash handler design.

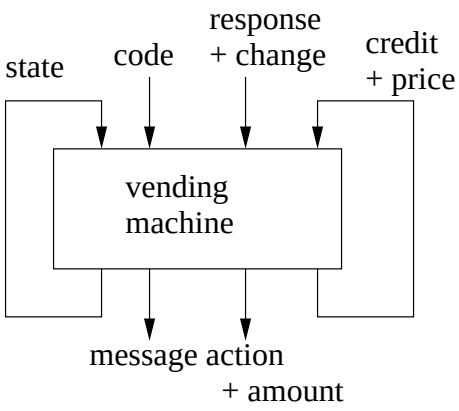

Figure 5.5: Vending machine.

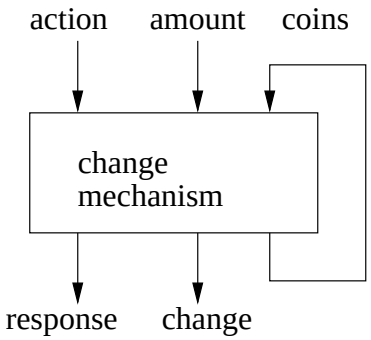

Figure 5.6: Change mechanism.

The *action and amount* wire is from, and the *response and change* wire is back to, the vending machine. The *current coins* self-wire keeps track of the number of each denomination of coin the machine holds.

We will assume that the cash handler accepts UK coins in the demoninations £2, £1, fifty pence, twenty pence, ten pence, five pence, two pence and one penny:

```
values = <<200,100,50,20,10,5,2,1>>;

MAXCOINS = 8;

type COINS = vector 1 .. 8 of integer;
```

It is useful to be able to display a set of coins as an appropriately formatted string:

```
showChange c i =
 if i>MAXCOINS
 then "\n"
 else
  if c@i==0
  then showChange c (i+1)
  else (c@i) as string++"* "++(values@i) as string++"p; "++showChange c (i+1);
```

First of all, let's consider adding a coin to some set of counts of coins:

```
exception BAD_COIN::integer;

add coin coins i =
 if i>MAXCOINS
 then raise BAD_COIN coin
 else
  if coin==values@i
```

```
  then update coins i (coins@i+1)
  else add coin coins (i+1);
```

We need to be able to check if some amount of money is satisfied by some set of coins:

```
data ACK = OK | FAIL;

check 0 _ _ = OK;
check amount coins i =
 if i>MAXCOINS
 then FAIL
 else
  let m = amount div (values@i)
  in
   if m>coins@i
   then check (amount-values@i*coins@i) coins (i+1)
   else check (amount-values@i*m) coins (i+1);
```

We also need to be able to return an amount of change from an initial set of coins, along with the remaining coins:

```
release 0 change coins _ = (change,coins);
release amount change coins i =
 let m = amount div (values@i)
 in
  if m>coins@i
  then release (amount-values@i*coins@i)
               (update change i (change@i+coins@i))
               (update coins i 0) (i+1)
  else release (amount-values@i*m)
               (update change i (change@i+m))
               (update coins i (coins@i-m)) (i+1);
```

The change mechanism may be asked to *ADD* some ammount to the coins it holds, *CHECK* if some amount is satisfied by its coins or *RELEASE* some amount of change from its coins:

```
data ACTION = ADD | CHECK | RELEASE;

box change
in (action::ACTION,amount::integer,coins::COINS)
out (response::ACK,change::COINS,coins'::COINS)
match
 (ADD,coin,coins) ->
  let coins' = add coin coins 1
  in (*,*,coins') |
 (CHECK,amount,coins) ->
```

```
  (check amount coins 1,*,coins) |
 (RELEASE,amount,coins) ->
  case release amount <<0,0,0,0,0,0,0,0>> coins 1 of
   (change,coins') -> (*,change,coins');
```

Note that the transitions selectively generate outputs. Thus: for *ADD*, no response or change are returned; for *CHECK*, no change is returned; for *RELEASE*, no response is returned.

The vending machine itself may *INPUT* a customer request of the form:

`c 0` $\Rightarrow$ release credit as change
`c` *coin* $\Rightarrow$ add *coin* to credit
`d` *amount* $\Rightarrow$ if credit is more than *amount*, debit credit

It may also be *CHECKING* to see if the change machine enough coins to satisfy a debit, or waiting for the change mechanism to *RELEASE* change:

```
data STATE = INPUT | CHECKING | CHANGE;

showCredit credit = "CREDIT "++credit as string++"\n";

box vending
in (s::STATE,code::(char,integer),credit::integer,price::integer,
    response::ACK,change::COINS)
out (s'::STATE,m::string,credit'::integer,price'::integer,
     action::ACTION,amount::integer)
match
(INPUT,('c',0),credit,*,*,*) ->
 (CHANGE,"GETTING CHANGE\n",0,*,RELEASE,credit) |
(INPUT,('c',coin),credit,*,*,*) ->
 (INPUT, showCredit (credit+coin),credit+coin,*,ADD,coin) |
(INPUT,('d',amount),credit,*,*,*) ->
 if amount>credit
 then (INPUT,"NOT ENOUGH CREDIT\n"++showCredit credit,credit,*,*,*)
 else (CHECKING,"CHECKING CHANGE\n",credit,amount,CHECK,amount) |
(CHECKING,*,credit,price,OK,*) ->
 (INPUT,"PURCHASE MADE\n"++showCredit (credit-price),credit-price,*,*,*) |
(CHECKING,*,credit,_,FAIL,*) ->
 (INPUT,"NO CHANGE\n"++showCredit credit,credit,*,*,*) |
(CHANGE,*,credit,*,*,change) ->
 (INPUT,showChange change 1++showCredit credit,credit,*,*,*);
```

The transactions here are somewhat more complicated, with considerable use of ignore patterns and expressions.

Finally, we wire the whole cash handler together. Initially, the change mechanism has no coins and the customer has nor credit:

```
wire change
```

```
(vending.action,vending.amount,change.coins' initially <<0,0,0,0,0,0,0,0>>)
(vending.response,vending.change,change.coins);

wire vending
(vending.s' initially INPUT,input,vending.credit' initially 0,vending.price',
 change.response,change.change)
(vending.s,output,vending.credit,vending.price,change.action,change.amount);
```

For example:

```
c5
CREDIT 5
c5
CREDIT 10
c10
CREDIT 20
c50
CREDIT 70
d15
CHECKING CHANGE
PURCHASE MADE
CREDIT 55
c0
GETTING CHANGE
1 * 50 p; 1 * 5 p;
CREDIT 0
...
```

We will later explore how to use sockets to connect the machine to a Java graphical user interface (GUI).

5.4 File I/O

File I/O is rather more complex than with standard I/O. Hume boxes are non-terminating, and standard I/O is potentially an endless stream of characters, whereas files are all too finite. The central problems are dealing with end of file for input and indicating end of file for output.

In Hume, when end of file is detected, the system `EndOfFile` exception is thrown. This implies that a box directly connected to a file cannot accumulate input data on a self-wire as it will not be accessible after the end of file exception.

However, file input may be managed by separating out the box consuming the file from the box that processes the data, and linking them with one wire to carry a tag indicating whether or not end of file has been reached, and another to carry the data. After end of file, the non-existent data can be ignored and so need not be sent. Similarly, the tag can be ignored so long as there is more data, and so need not be sent either.

For example, consider summing an unknown number of integer from a file:

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

data STATUS = EOF;

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

box sumnumbs
in (s::STATUS,n,sum::integer)
out (sum',result::integer)
match
 (*,n,sum) -> (sum+n,*) |
 (EOF,*,sum) -> (*,sum);

wire getnumbs (input) (sumnumbs.s,sumnumbs.n);
wire sumnumbs
(getnumbs.s,getnumbs.n',sumnumbs.sum' initially 0)
(sumnumbs.sum,output);
```

Files are closed when a program terminates. And programs terminate when no box is RUNNABLE i.e. when all boxes are BLOCKEDOUT[1].

5.5 Box templates

Consider building a full adder from two half adders and an OR, where the halfadders in turn are made from XOR and AND, as shown in Figure 5.7.

To simplify the presentation we have used a modified fanout that takes two separate inputs.

We now have two fanouts, two XORs and two ANDS. Rather than copying, pasting and renaming the boxes for these constructs, we can use a box template to define replicated copies.

Thus:

```
template identifier1
in (...)
out (...)
match ...;
```

defines a template for the $identifier_1$ family of identical boxes.

Then:

[1] TP 17: there may be a hack in some implementations where a file is closed if sent '\0'

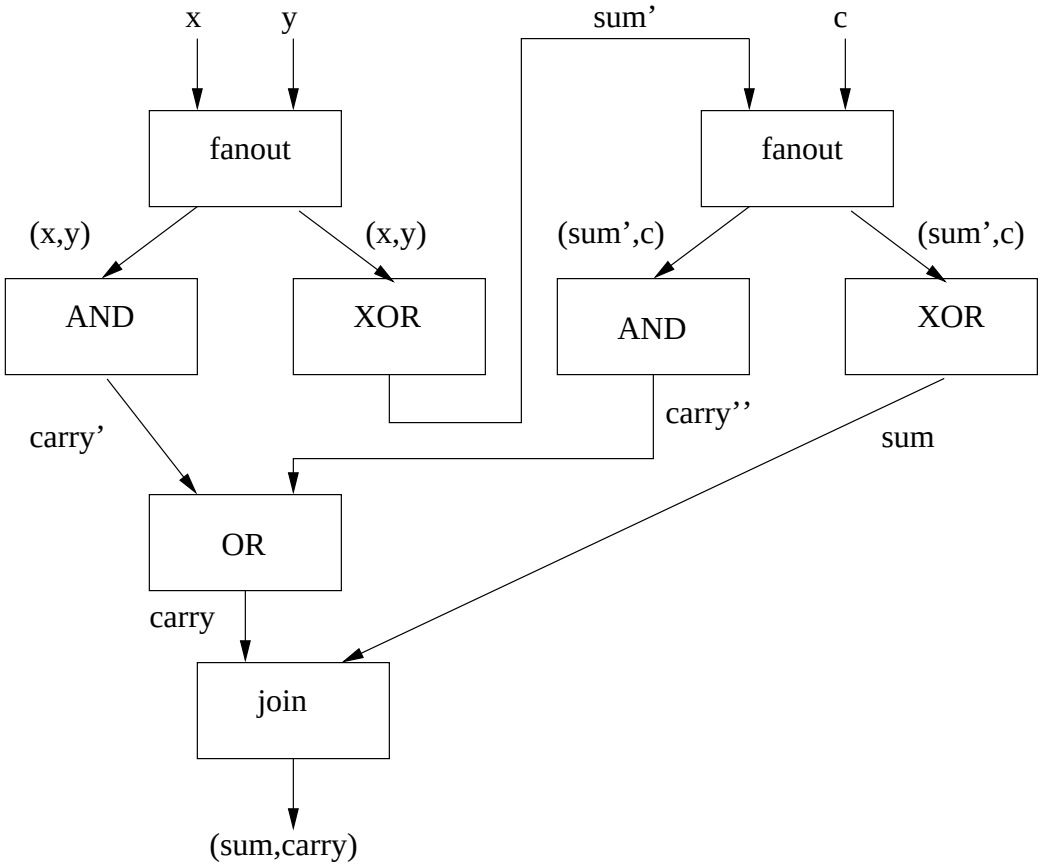

Figure 5.7: Full adder as gates.

`instantiate` $identifier_1$ `as` $identifier_2$`*`N

where N is some integer, will create N copies of $identifier_1$ with names $identifier_2 1$, $identifier_2 2$, ... , $identifier_2 N$.

In the full adder, we can now define:

```
template fanout
in (x,y::Bit)
out (x1,y1,x2,y2::Bit)
match
 (x,y) -> (x,y,x,y);

template xor
in (x,y::Bit)
out (z::Bit)
match ...;

template and
in (x,y::Bit)
out (z::Bit)
match ...;
```

and create the requisite copies with:

```
instantiate fanout as f*2;
instantiate xor as x*2;
instantiate and as a*2;
```

Wiring of the new boxes `f1`, `f2`, `x1`, `x2`, `a1` and `a2` then proceeds as usual.

5.6 Wiring loops and macros

Consider a very simple simulation of a circular railway track on which trains may only travel in a clockwise direction. The circuit is built from track sequences of the form shown in Figure 5.8.

The idea is that a train may enter from the previous track and exit to the next track. It can see the next track's entry signal and it displays its own entry signal to the previous track.

When a train is on a track and the next track's signal is red then the track's signal remains red. If the next track's signal is green then the train may exit the track and it's signal goes green. If a track is empty then a train may enter it and its signal goes red. Otherwise an empty track retains a green signal:

```
template track
in (s::STATE,entry::STATE,nextsignal::SIGNAL)
out (s'::STATE,exit::STATE,signal::SIGNAL)
match
```

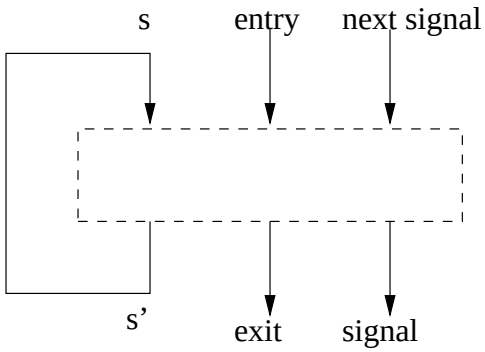

Figure 5.8: Track.

```
 (TRAIN,*,RED) -> (TRAIN,*,RED) |
 (TRAIN,*,GREEN) -> (EMPTY,TRAIN,GREEN) |
 (EMPTY,TRAIN,_) -> (TRAIN,*,RED) |
 (EMPTY,*,_) -> (EMPTY,*,GREEN) ;

instantiate track as t*4;
```

Now, wiring even four pieces of track becomes tedious and error prone:

```
wire t1
(t1.s' initially TRAIN,t4.exit,t2.signal initially GREEN)
(t1.s,t2.entry,t4.nextsignal);
wire t2
(t2.s' initially EMPTY,t1.exit,t3.signal initially GREEN)
(t2.s,t3.entry,t1.nextsignal);
wire t3
(t3.s' initially EMPTY,t2.exit,t4.signal initially GREEN)
(t3.s,t4.entry,t2.nextsignal);
wire t4
(t4.s' initially EMPTY,t3.exit,t1.signal initially GREEN)
(t4.s,t1.entry,t3.nextsignal);
```

Hume offers number of forms which greatly simplify wiring. First of all, simple *macros*:

> `macro` $identifier\ identifier_1$ `=` $expression$`;`

may be called as:

> $identifier$`(`$expression_1$`)`

within $\{\ldots\}$ in wire declarations.

Hume also provides *wiring loops* to simplify repetitive wiring:

for *identifier* = *expression*$_1$ to *expression*$_2$
[except(*expression*$_3$,...)]
wire − declaration;

where *wire − declaration* may contain references to {*identifier*}. This is then equivalent to a sequence of *wire − declaration* for all integer values from *expression*$_1$ to *expression*$_N$, optionally except for *expression*$_3$ etc.

Wiring loops are further augmented with *wiring macros*:

wire *identifier* (*identifier*$_1$,...) = *wire − declaration*;

which are called as:

wire *identfier*(*expression*$_1$,...)

For example, we might simplify the wiring above to:

```
constant TRACKS = 4;

wire t1
 (t1.s' initially TRAIN,t4.exit,t2.signal initially RED)
 (t1.s,t2.entry,t4.nextsignal);

wire Track (this,prev,next) =
 wire {this}
 ({this}.s' initially EMPTY,{prev}.exit,{next}.signal initially GREEN)
 ({this}.s,{next}.entry,{prev}.nextsignal);

for i = 2 to TRACKS-1
 wire Track (t{i},t{i-1},t{i+1});

wire Track (t4,t3,t1);
```

Note the special cases for the tracks at the "ends" of the circuit: `t1` and `t4`. Note the even more special case for `t1` as it has a different initial state.

And we can't actually observe this program doing anything unless we trace it.

5.7 Sockets

5.8 Foreign functions

5.9 Fair matching

The Hume matching we have met so far is *unfair*: on each execution cycle, transitions are considered in sequence starting with the first. Thus, the same rules may always succeed with subsequent rules never being considered.

Unfair matching may be subverted using an explicit state self-wire with an enumerated constructed type to ensure that each match is considered as required.

For example, suppose we wish to send alternate input values to two outputs:

```
data STATE = LEFT | RIGHT;

box alternate
in (s::STATE,n::integer)
out (s'::STATE,l,r::integer)
match
 (LEFT,n) -> (RIGHT,n,*) |
 (RIGHT,n) -> (LEFT,*,n);

wire alternate (alternate.s' initially LEFT,...)
               (alternate.s,...,.
```

Here the alternating value on the `STATE` self-wire value results in alternating transition match success.

Where it is important for every match to have an equal probability of success, Hume also provides *fair* matching:

> `box` *identifier*
> `in (`*identifier*`::`*type* `...)`
> `out (`*identifier*`::`*type* `...)`
> `fair` *transitions*`;`

For example, we can rewrite our example as:

```
box alternate1
in (n::integer)
out (l,r::integer)
fair
 n -> (n,*) |
 n -> (*,n);

wire alternate1 (...) (,...,...)
```

With fair matching, transition selection can be biased by replicating transitions. Thus, in our example, to send twice as many inputs to the left as to the right:

```
box alternate2
in (n::integer)
out (l,r::integer)
fair
 n -> (n,*) |
 n -> (n,*) |
 n -> (*,n);
```

One fair match implementation technique is *round robin* where on each excecution cycle, matching starts with the first transition after the one which most recently matched successfully on a previous cycle.

Bibliography

[1] B. Nordström, K. Petersson and J. M. Smith. *Programming in Marton-Löf's Type Theory: An Introduction*. Oxford, 1990.

[2] A. Bundy, G. Grosse, and P. Brna. A recursive techniques editor for Prolog. *Instructional Science*, 20:135–172, 1991.

[3] R. Burstall. Inductively Defined Functions in Functional Programming Languages. Technical Report ECS-LFCS-87-25, LFCS, University of Edinburgh, April 1987.

[4] M. Davis. *Computability and Decidability*. McGraw-Hill, 1958.

[5] S. Peyton Jones (Ed). *Haskell 98 Languages and Libraries: the Revised Report*. Cambridge, 2003.

[6] K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*. Springer, 1999.

[7] K. Hammond, G. Michaelson, and R. Pointon. The Hume Report 1.1. Technical report, March 2007. `http://www-fp.cs.st-andrews.ac.uk/hume/report/hume-report.pdf`.

[8] G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[9] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, 1969.

[10] I. Horrocks. *Constructing the User Interface with Statecharts*. Addison-Wesley, 1998.

[11] S. Kleene. *Introduction to Meta-Mathematics*. North-Holland, 1952.

[12] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999.

[13] R. Milner, M. Toft, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT, 1997.

[14] A. Salomma. *Theory of Automata*. Pergamon, 1969.

[15] P. Stevens and R. Pooley. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.

[16] D. Turner. Elementary Strong Functional Programming. In *1st International Symposium on Functional Programming Languages in Education, Nijmegen, The Netherlands*, number 1022 in LNCS. Springer, December 1995.