Reasoning about multi-process systems with the box calculus

Greg Michaelson$_1$ and Gudmund Grov$_2$

$_1$Heriot-Watt University
$_2$University of Edinburgh

Abstract. The box calculus is a formalism for reasoning about the properties of multi-process systems which enables account to be taken of pragmatic as well as computational concerns. It was developed for the programming language Hume which explicitly distinguishes between coordination, based on concurrent boxes linked by wires, and expressions, based on polymorphic recursive functions. This chapter introduces Hume expressions and surveys classic techniques for reasoning about functional programs. It then explores Hume coordination and the box calculus, and examines how Hume programs may be systematically transformed while maintaining computational and pragmatic correctness.

1 Overview

Having constructed programs that meet their specifications, we often want to change them to take advantage of changing operating environments. For example, we might want to migrate a program from environments with smaller to larger numbers of processors to improve performance, in particular as the number of cores grows in new generations of the same CPU architecture. In changing programs, we want to ensure not only they still meet their original specifications, but also that their *pragmatic* (i.e. time, space, sequencing) behaviours change in well understood ways. In particular, in making what appear to be local improvements to a program, for example by introducing parallelism, we want to avoid unexpected global impacts that make overall performance worse, for example as a result of unanticipated additional communication or scheduling costs. Thus, we need some means of *reasoning* about changes to programs that can account for pragmatic as well as computational program properties.

Most software is constructed in imperative programming languages; abstractions from von Neumann architectures[1] based on sequences of state changes mediated by mutable memory. Here, different program components interact by manipulating the same memory areas. Thus, changing the order of component execution often changes the sequence of memory manipulation, changing what the program does. This complicates reasoning about imperative programs because the state change sequencing must be made explicit in the reasoning

[1] And also Harvard architectures.

rules. While there are mature systems like Floyd-Hoare logic[Hoa69] and weakest preconditions[Dij75] for establishing properties of imperative programs, they require considerable mathematical sophistication, scale poorly with program size and lack mature automated or semi-automated tool support.

In contrast, declarative languages do not have any notion of state change through mutable memory. Instead, program components interact by passing each other values. Thus, in principle, components may be executed in arbitrary order without changing what the program does. This is alleged to make it simpler to reason about declarative programs, compared with imperative ones, as the reasoning rules do not require any notion of sequenced state change. Nonetheless, reasoning about functional programs is not really much easier than for imperative programs: once again, the mathematics is hard, scalability is poor and tools are lacking.

The introduction of parallelism further complicates reasoning about programs. Parallelism requires interaction between processors, either through shared mutable memory access or distributed memory message passing, and that interaction must take place in some order and at some additional performance cost. The development of tools and technologies for reasoning about parallelism is hampered by factors quite orthogonal to those constraining reasoning about sequential programs.

In particular, despite the explosive growth in deployment of multi-processor architectures, there is effectively no standardisation of parallel programming languages. This is hardly surprising. Given the vast investments in software tools and technologies for sequential imperative languages, it is really hard to make a commercial case for adapting unproven extensions to extant languages, let alone new parallel languages: the rise and fall of occam[Inm88] is an object lesson. Instead, extant languages tend to be augmented with libraries like MPI[MF94], for message passing, and OpenMP[CJP07], for shared memory. As yet, there is little formalisation of these libraries and so scant theoretical or practical support for formal reasoning about practical parallelism.

There are, of course, mature formalisms for reasoning about abstract parallelism, for example CSP[Hoa78], CCS[Mil82] and the π calculus[Mil99]. However, while well suited to reasoning about coordination, these take little account of the computations that are being coordinated, and share the same constraints as sequential formalisms.

It has long been claimed [Weg68] that declarative programs are ideal for parallelism as the absence of sequences state change enables implicit parallelism at all levels of programs. Indeed, parallelism formalisms share strong roots with declarative languages. In practice, such implicit parallelism is almost invariably too fine grain to be exploited efficiently. That is, the cost of the interaction between newly parallel program components outweighs any benefit from executing them in parallel. Thus, there is considerable research into developing new declarative languages for parallelism, such as Eden[BLOMP97], or extending extant languages, such as Haskell[eAB$^+$99], again without any wide adoption of a single language or stable standardisation of extensions. Nonetheless, as we shall see, declarative languages do offer valuable abstractions for parallelism in *higher order functions (HOFS)* which generalise common patterns of computation enabling their efficient realisation as standard patterns of coordination.

Hume[HM03] is a general purpose programming language which was designed to enable the construction and analysis of systems where strong assurances are required that resource bounds are met. This language has deep roots in contemporary functional languages and is based on concurrent finite state automata with transitions controlled by pattern matching over inputs to invoke recursive expressions to generate outputs.

To meet Hume's design objectives, an explicit distinction is made amongst the expression, coordination and control *layers*:

- The *expression* layer is based on a strict, polymorphic functional language with a rich type system, reminiscent of Standard ML[MTHM97] or Haskell. This layer is used to define computations that return values for use in box transitions.
- The *coordination* layer is based on concurrent generalised finite state automata consisting of *boxes* linked by *wires*. This layer is used to define boxes and wiring, and box and wiring templates.
- The *declaration* layer is used to define common auxiliary constructs for use throughout programs, for example: constants; functions; type aliases; type signatures; exceptions and constructed data types.

To further facilitate resource analysis complementing system development, Hume supports the notion of *language level* with different formal properties, depending on the types of values on wires between boxes, and the forms of expressions within boxes. Thus:

- *full Hume* is a Turing complete language with undecidable time and space behaviour;
- *PR-Hume* restricts recursion to primitive recursion. Thus, time and space are decidable though not necessarily well bound;
- Template-Hume prohibits user defined recursion but provides a repertoire of higher order functions with well characterised behaviours. Here, time and space bounds may be well bound.
- *FSM-Hume* corresponds to a richly typed finite state machine abstraction. There is no recursion and all repetition is through iteration over boxes. Furthermore, only types of known size may be passed on wires. FSM-Hume enables accurate time and space analysis.
- *HW-Hume* is an impoverished language oriented to hardware at the bit levels, supporting pattern matching on tuples of bits to produce tuples of bits. HW-Hume enables precise time and space analysis.

It is unrealistic to expect programmers to restrict themselves to one level. Instead, Hume supports a methodology of transformational software development. An initial system is built and analysed. If satisfactory bounds cannot be established then problematic loci may be changed into a lower level, typically by moving activity from within boxes to between boxes. Clearly, reasoning about and changing programs at the coordination layer almost invariably requires reasoning and change at the expression layer. Thus, the box calculus strongly reflects this language design, and is novel in enabling movements between layers in search of optimal programs.

In the rest of this chapter we will:

- introduce the Hume expression layer;
- survey classical techniques for reasoning about functional programs;
- introduce the Hume coordination layer;
- explore the foundations of the box calculus;
- apply the box calculus to systematically deriving a range of multi-box programs from single box programs.

2 Hume expression language

2.1 Base types and expressions

Base types For our purposes, the main Hume base types are integer, floating points, words, characters and booleans. All base types are *sized*, that is they have fixed ranges of values which are related to the number of bits that instances occupy. For numeric types, the size is specified explicitly. For example, the integer type constructor is:

- `int` *size* - signed integer;

where *size* is some multiple of 8[2].

Type aliases In practice, it is usual to use *type aliases*, rather than raw type constructors, of the form:

`type` *id* `=` *type*`;`

where *id* is an identifier composed of upper and lower case letters and digits, starting with a letter or _, and *type* is a type expression, in the first instance a type constructor. Thereafter, *id* may be used in any context where a type expression is appropriate. For example:

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

defines `integer` to be an alias for `int 64`.

Base values Hume has the standard base value representations:

- integers are sequences of possibly negative decimal digits: e.g. `12345`, `-678910`;
- floats are sequences of decimal digits separated by a decimal point: e.g. `123.456`, `-789.1011`;
- words are sequences of hexadecimal digits preceded by `0x`: e.g. `0xabcdef`;
- characters are letters or escaped letters within single quotes: e.g. `'a'`, `'\n'`(newline);
- booleans are `true` or `false`.

[2] In practice, current Hume implementations tie all sizes to some C equivalent.

Base expressions All expressions may be structured by brackets `(...)`.

The integer infix operators are `+` (addition), `-` (subtraction), `*` (multiplication), `div` (division) and `mod` (remainder). All integer operators take two integer operands. The precedence, in descending order, is: `(...)`; unary `-`; `+`, `-`; `*`, `mod`, `div`.

The float infix operators are `+` (addition), `-` (subtraction), `*` (multiplication) and `/` (division). All float operators take two float operands. The precedence, in descending order, is: `(...)`; unary `-`; `+`, `-`; `*`, `/`.

The boolean operators are `not` (prefix negation), `&&` (infix conjunction) and `||` (infix disjunction). All boolean operators take boolean operands. The precedence, in descending order, is: `(...)`, `not`, `||`, `&&`.

The comparison operators are `==` (equals), `!=` (not equals), `<` (less than), `<=` (less than or equal), `>=` (greater than or equal) and `>` (greater than). All comparison operators take operands of the same type.

Constant declaration Constants may be declared by:

id `=` *expression*`;`

Here, *id* is associated with the value of *expression* and may be used in subsequent *expression*s. For example:

```
cost = 35;
quantity = 12;
total = cost * quantity;
```

associates `cost` with `35`, `quantity` with `12` and `total` with `420`.

2.2 Functions

Function declaration At simplest, Hume functions are declared as:

id pattern `=` *expression*`;`

where *id* names the function, *pattern* introduces formal parameters and *expression* is the function body. To begin with, a *pattern* may just be an *id*. For example:

```
inc x = x+1;
isZero y = y==0;
```

declares `inc` to be function that increments its argument `x` and `isZero` to be a function that checks if it's argument `y` is zero.

Function type and type signature If *pattern* is $type_1$ and *expression* is $type_2$ then *id* is: $type_1$ `->` $type_2$.

Types may be nominated explicitly through a *type signature* of the form:

id `::` *type*`;`

For example, we could make the types of `inc` and `isZero` explicit as:

```
inc :: int 32 -> int 32
isZero :: int 32 -> bool
```

Hume supports polymorphic type inference and it is not a requirement to specify the function type if it is inferable from the context of declaration or use. However, it may be necessary to provide a type signature to disambiguate overloaded operators which may appear in different type contexts. For example:

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

In `sq`, `*` is overloaded so `x`'s type cannot be inferred. Here, the type signature makes it explicit that `sq` operates on integers so `*` must be an integer operator.

Explicit parameter type An alternative to deploying a type signature is to explicitly nominate the type of a formal parameter using:

(*pattern*`::`*type*)

instead of *pattern*. For example:

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

makes it explicit that `x` is `integer` in `sq`.

Function call Functions are called with expressions of the form:

id expression

Here, *id* is a name associated with a function and *expression* is the actual parameter. If *id* is associated with a $type_1$`->`$type_2$ function and *expression* is $type_1$ then the call returns a value of $type_2$.

For a function call: where *id*'s function has formal parameter *pattern*, consisting in the first instance of a single id_1, and body $expression_1$, then:

- the actual parameter *expression* is evaluated to *value*;
- *value* is matched with *pattern* i.e. id_1 is bound to *value*;
- the function body $expression_1$ is evaluated with all free occurrences of the formal parameter id_1[3] replaced by *value*.

Note that Hume does not support anonymous functions.

Note that this is a *substitutive* model of function call evaluation. For example:

1. `inc 41`
2. $\rightarrow$ `x+1` with `x` bound to `41`
3. $\rightarrow$ `41+1`

[3] i.e. occurrences outwith the scope of some other declaration of id_1 in $expression_1$

4. → `42`

For example:

1. `isZero (inc 3)`
2. → `x+1` with `x` bound to `3`
3. → `3+1`
4. → `4`
5. → `y==0` with `y` bound to `4`
6. → `4==0`
7. → `false`

A function call has precedence higher than numeric operators and lower than `(...)`.

2.3 Tuples

Tuple form A tuple is a fixed sized sequence of elements of possibly different types. A tuple has the form:

`(`exp_1`,`$exp_2 \ldots exp_N$`)`

If exp_i is of $type_i$ then this tuple has type:

`(`exp_1`,`$exp_2 \ldots exp_N$`) :: (`$type_1$`,`$type_2$`,`$\ldots type_N$`)`

For example:

```
(1,2.0,true) :: (int 32,float 32,bool)
(1,(2,3),4) :: (int 32,(int 32,int 32),int 32)
```

Tuple pattern Tuple patterns may be used for multi-parameter functions. These take the form:

`(`$patt_1$`,`$patt_2 \ldots patt_N$`)`

In:

id `(`$patt_1$`,`$patt_2 \ldots patt_N$`) =` $expression$

if $patt_i$ is $type_i$ and $expression$ is $type_{N+1}$ then the function type is:

`(`$type_1$`,`$type_2 \ldots type_N$`) ->` $type_{N+1}$

Then, a function call:

$id\ expression_1$

proceeds as:

– evaluate $expression_1$ to:
`(`$value_1$`,`$value_2 \ldots value_N$`)`

– match $patt_i$ with $value_i$ i.e. bind id_i from $patt_i$ to $value_i$;
– evaluate the body *expression* with these bindings.

For example, given:

```
quad :: (integer,integer,integer,integer) -> integer;
quad (a,b,c,x) = a*x*x+b*x+c;
```

then:

```
1. quad (1,2,1,3)
2. → a*x*x+b*x*c with a=1, b=2, c=1 and x=3
3. → 1*3*3+2*3+1
4. → 16
```

2.4 Multi-case functions

Multi-case function declaration Multi-case functions may be declared as:

id $pattern_1$ `=` $expression_1$`;`
id $pattern_2$ `=` $expression_2$`;`
`...`
id $pattern_N$ `=` $expression_N$`;`

All cases must have same *id*. All $pattern_i$ must be same $type_1$. All $expression_i$ must be the same $type_2$. The function then has type: $type_1$`->`$type_2$.

As we shall see, case order is significant. The $pattern_i$ should be disjoint and cover all possible values of $type_1$. Thus, it is usual, for functions that do not have exhaustive cases, to provide a final case with a catch-all *id* pattern.

Constant pattern Patterns may include constant values in any positions where *id* may appear. For example, we might define boolean negation as:

```
Not false = true;
Not true = false;
```

and natural number decrement as:

```
natDec 0 = 0;
natDec x = x-1;
```

For a constant pattern match to succeed, the same constant must appear in the same structural position in the formal parameter pattern and actual parameter value.

Multi-case function call For:

id expression

- *expression* is evaluated to some *value*;
- *value* is matched against each *pattern* in turn from first to last.
- if a match with $pattern_i$ succeeds then $expression_i$ is evaluated;

For example, for:

```
Not true
```

then:

1. try `Not false` - `false` does not match `true`;
2. try `Not true` - `true` matches `true`;
3. $\rightarrow$ `false`

For example, for:

```
natDec 3
```

then:

1. try `natDec 0` - `0` does not match `3`;
2. try `natDec x` - `x` binds to `3`;
3. $\rightarrow$ `x-1`
4. $\rightarrow$ `3-1`
5. $\rightarrow$ `2`

2.5 Recursion

Recursive function declaration As in all functional languages, Hume functions may call themselves. At simplest, a recursive function has the typical structure:

- **base case** match constant and return final value;
- **recursion case** match *id* and call function again with modified *id*.

The recursion case should make progress towards the base case by changing the recursion parameter *id*.

For example, consider summing a sequence of integers from N to 0:

$N + (N-1) + ... + 2 + 1 + 0$

We can write this as:

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

so:

```
sum 3 → 3+sum 2 → 3+2+sum 1 → 3+2+1+sum 0 → 3+2+1+0 → 6
```

For example, consider summing a sequence of squares from N to 0:

$N^2 + (N-1)^2 + ... + 2^2 + 1^2 + 0$

We can write this as:

```
sumSq 0;
sumSq N = sq n+sumSq (N-1);
```

so:

```
sumSq 3 → sq 3+sumSq 2 → sq 3+sq 2+sumSq 1 →
sq 3+sq 2+sq 1+sumSq 0 → sq 3+sq 2+sq 1+0 → 9+4+1+0 → 14
```

2.6 Higher order functions 1

As a first definition, a higher order function takes other functions as parameters. For example, consider summing some function f over the range from N to 0:

$f\ N + f\ (N-1) + ...f\ 2 + f\ 1 + 0$

We may write this as:

```
sumF :: (integer->integer,integer)->integer;
sumF (f,0) = 0;
sumF (f,N) = f N+sumF (f,n-1);
```

Then we may sum squares from N to 0 with:

```
sumF (sq,3) → sq 3+sumF (sq,2) → sq 3+sq 2+sumF (sq,1) →
sq 3+sq 2+sq 1+sumF (sq,0) → sq 3+sq 2+sq 1+0 → 9+4+1+0 →
14
```

2.7 Curried functions

Functions with tuple formal parameters may be written as nested or *Curried* functions. Thus:

id`::(`$type_1$`,`$type_2 ... type_N$`)->`$type$`;`
id `(`$patt_1$`,`$pat_2 ... patt_N$`) =` $expression$`;`

has equivalent nested function:

id`::`$type_1$`->`$type_2 ... type_N$`->`$type$`;`
$id\ patt_1\ patt_2\ ...\ patt_N$ `=` $expression$`;`

For example:

```
quad::(integer,integer,integer,integer)->integer;
quad (a,b,c,x) = a*x*x+b*x+c;
⇔
quadC::integer->integer->integer->integer->integer;
quadC a b c x = a*x*x+b*x+c;
```

Curried functions are called as:

$id\ exp_1\ exp_2\ \ldots\ exp_N$

For example:

```
quadC 1 2 1 3;
```

Currying is a matter of style. It's use lies in support for *partial application* where a function of N parameters is applied to $M < N$ parameters to return a function of $N - M$ parameters with the first M parameters bound to specific values. However, Hume does not support partial application.

2.8 Constructed types 1

New types with distinct constant values may be declared by:

`data` *id* `=` *id*$_1$ `|` *id*$_2$ `| ... ;`

Here, *id* is the new type, and the id_i are new constructors returning values id_i of type *id*.

For example:

```
data STATE = ON | OFF;
```

declares a new type `STATE` with values `ON` and `OFF`.

Type constructors may be used as constants in patterns. For example, to flip `STATE`:

```
change ON = OFF;
change OFF = ON;
```

Here, `change` has type `STATE->STATE`.

2.9 Higher Order Functions 2

A second definition of a higher order function is one that returns a function. For example, consider:

```
data FN = INC | SQ;
getFN :: FN -> (integer->integer);
getFN INC = inc;
getFN SQ = sq;
```

Here, `getFN` returns either `inc` or `sq`, depending on the actual parameter. Thus:

```
getFn SQ 3 → sq 3 → 9
```

As always, the functions returned in different cases must have the same type, in this instance `integer->integer`.

2.10 Polymorphism

The ability to abstract over types is termed *polymorphism*, from the Greek for “many forms”. In Hume, type expressions may include *type variables* with single lower-case letter identifiers: `a`, `b`, `c` etc.

In a type expression, all occurrences of a type variable must be capable of being replaced consistently with the same type. For example, consider the identity function:

```
identity :: a -> a;
identity x = x;
```

In:

`identity 42` $\rightarrow$ `42`

the type variable `a` is replaced consistently by `int 32`. In:

`identity ('a','b','c')` $\rightarrow$ `('a','b','c')`

the type variable `a` is replaced consistently by `(char,char,char)`.

2.11 Lists

List representation A *list* is an arbitrary length sequence of the same type. A list whose elements are of *type* has type `[`*type*`]`.

Lists are formed using the infix concatenation operator `:` of effective type `(a,[a])->[a]`. That is, if exp_1 is of *type* and exp_2 is a list of *type*, i.e. `[`*type*`]`, then exp_1`:`exp_2 is of type `[`*type*`]`.

The *empty list* is `[]` of effective type `[a]` and must end every list.

For example:

```
1:2:3:[] :: [int 32]
('a',true):('b',false):('c',false):[] :: [(char,bool)]
inc:sq:[] :: [int 32->int 32]
```

In exp_1`:`exp_2, exp_1 is called the list *head* and exp_2 the list *tail.*

The simplified notation:

exp_1`:`exp_2`:...:`exp_N`:[]` $\Leftrightarrow$ `[`exp_1`,`exp_2`,...,`exp_N`]`

is often used. For example:

```
1:2:3:[] ⇔ [1,2,3]
('a',true):('b',false):('c',false):[] ⇔
  [('a',true),('b',false),('c',false)]
inc:sq:[] ⇔ [inc,sq]
```

Note that:

`[`*exp*`]` $\Leftrightarrow$ *exp*`:[]`

List pattern and list recursion Formal parameter patterns may include the forms:

$patt_1$`:`$patt_2$
`[`exp_1`,`exp_2`...`exp_N`]`

For both forms, the actual parameters must have corresponding structures. Then, elements of the list pattern are matched against corresponding values in the actual parameter.

List recursion then has the typical structure:

- **base case** `[]` - return final value;
- **recursion case** `(`id_1`:`id_2`)` - recurse on id_2 and combine with modified id_1.

For example, to find the length of a list:

```
Length :: [a] -> integer;
Length [] = 0;
Length (h:t) = 1+Length t;
```

so:

```
Length [1,2,3] → 1+Length [2,3] → 1+1+Length [3] → 1+1+1+Length
[] → 1+1+1+0 → 3
```

For example, to sum the elements of a list:

```
sumL [] = 0;
sumL (h:t) = h+sumL t;
```

so:

```
sumL [1,2,3] → 1+sumL [2,3] → 1+2+sumL [3] → 1+2+3+sumL []
→ 1+2+3+0 → 6
```

To square all in a list:

```
sqList [] = [];
sqList (h:t) = sq h:sqList t;
```

so:

```
sqList [1,2,3] → sq 1:sqList [2,3] → sq 1:sq 2:sqList [3] →
sq 1:sq 2:sq 3:sqList [] → sq 1:sq 2:sq 3:[] → 1:4:9:[] →
[1,4,9]
```

2.12 List higher order functions

We will now survey a number of list higher order functions which we will use in later sections.

Sum function over list To sum a function over a list:

```
sumFL :: (a->integer)->[a]->integer;
sumFL f [] = 0;
sumFL f (h:t) = f h+sumFL f t;
```

so:

> `sumFL sq [1,2,3]` → `sq 1+sumFL sq [2,3]` → `sq 1+sq 2+sumFL sq [3]` → `sq 1+sq 2+sq 3+sumFL sq []` → `sq 1+sq 2+sq 3+0` → `1+4+9` → `14`

map To *map* a function over a list, that is apply a function to every element:

```
map :: (a->b)->[a]->[b];
map f [] = [];
map f (h:t) = f h:map f t;
```

so to square every element in a list:

> `map sq [1,2,3]` → `sq 1:map sq [2,3]` → `sq 1:sq 2:map sq [3]` → `sq 1:sq 2:sq 3:map sq []` → `sq 1:sq 2:sq 3:[]` → `1:4:9:[]` → `[1,4,9]`

append To *append* one list onto another, that is join the lists end to end:

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

For example:

> `append [1,2,3] [4,5,6]` → `1:append [2,3] [4,5,6]` → `1:append [2,3] [4,5,6]` → `1:2:append [3] [4,5,6]` → `1:2:3:append [] [4,5,6]` → `1:2:3:[4,5,6]` → `[1,2,3,4,5,6]`

`append` *l*1 *l*2 may be written *l*1`++`*l*2.

2.13 String

A *string* is a sequence of letters within `"..."`. For example:

> `"this is not a string"`

The string type constructor is `string`. A string is the same as a list of `char` so:

> `"hello"` ⇔ `'h':'e':'l':'l':'o':[]` ⇔ `['h','e','l','l','o']`

and:

> `"hello"++" "++"there"` ⇔ `"hello there"`

2.14 Conditional expression

Pattern matching can only determine the presence or absence of a constant value. To establish other properties of values a *conditional expression* may be used:

`if` $expression_1$ `then` $expression_2$ `else` $expression_3$

$expression_1$ must return a `bool`, and $expression_2$ and $expression_3$ must return the same type.

$expression_1$ is evaluated. If it is `true` then $expression_2$ is evaluated. Otherwise $expression_3$ is evaluated.

For example, to select all the even values in an integer list:

```
isEven y = y div 2==0;
getEven [] = [];
getEven (h:t) =
 if isEven h
 then h:getEven t
 else getEven t;
```

so:

`getEven [1,2,3,4]` → `getEven [2,3,4]` → `2:getEven [3,4]` → `2:getEven[4]` → `2:4:getEven []` → `2:4:[]` → `[2,4]`

Filter For example, to find all the elements of a list satisfying some property:

```
filter :: (a->bool)->[a]->[a];
filter p [] = [];
filter p (h:t) =
 if p h
 then h:filter p t
 else filter p t;
```

so:

`filter isEven [1,2,3,4]` → `filter isEven [2,3,4]` → `2:filter isEven [3,4]` → `2:filter isEven[4]` → `2:4:filter isEven []` → `2:4:[]` → `[2,4]`

2.15 Case expression

The *case expression* provides an expression form which is equivalent to a multi-case function declaration. For:

`case` *expression* `of`
$pattern_1$ `->` $expression_1$ `|`
$pattern_2$ `->` $expression_2$ `|`
`...`
$pattern_N$ `->` $expression_N$

expression and all $pattern_i$ must have the same type, and all $expression_i$ must have the same type. As with multi-case functions, the *pattern*s should be disjoint and there should be full coverage for the corresponding type, so a final case with a catch-all variable pattern is common.

expression is evaluated and matched against each *pattern* in turn from 1 to N. If the match with $pattern_i$ succeeds then the value of $expression_i$ is returned.

For example:

```
fib 0 = 1;
fib 1 = 1;
fib n = fib (n-1)+fib (n-2);
⇔
fib n =
 case n of
 0 -> 1 |
 1 -> 1 |
 n -> fib (n-1)+fin (n-2);
```

As we will see, the case expression is used in the box calculus to move activity between functions and boxes.

2.16 Constructed types 2

The constructed type form is generalised to enable the declaration of structured types. For:

`data` id `=` id_1 $type_1$ `|` id_2 $type_2$ `|` $\ldots$ id_N $type_N$`;`

each id_i $type_i$ is a type constructor of $type_i$`->`id.

The equivalent form id_i $pattern_i$ may then be used in patterns. For a match to succeed, the actual parameter id_j $expression_j$ must have the same constructor id_j as id_i and the $expression_j$ must match $pattern_i$.

For example, to declare integer lists:

```
data LIST = NIL | CONS (integer,LIST);
```

Here, the new type is `LIST` with constant value `NIL` and structured values of the form `CONS (`h`,`t`)` where h is an `integer` and t is a `LIST`. For example:

```
CONS(1,CONS(2,CONS(3,NIL)))
```

Then, we might declare a function to flatten a `LIST` into a `[integer]` as:

```
flatten NIL = [];
flatten (LIST(h,t)) = h:flatten t;
```

so:

```
flatten (CONS (1,CONS(2,CONS(3,NIL)))) →
1:flatten (CONS (2,CONS(3,NIL))) →
1:2:flatten (CONS(3,NIL)) →
1:2:3:flatten NIL → 1:2:3:NIL → [1,2,3]
```

3 Reasoning about functional programs

3.1 Introduction

Our starting point was that we have a correct program, that is one that satisfies its specification, and we wish to change it in various ways without compromising that correctness. In the widely used Floyd-Hoare paradigm, we assume that we have *proved* that:

$$\{P\}program\{Q\}$$

That is, for some *program*, given a precondition P, which is true at the start of the program, then we can prove that some post-condition Q is true at the end of the program, using an appropriate proof theory. If we then change *program* to *program′* we then need to prove:

$$\{P\}program'\{Q\}$$

That is we must prove that the new program still satisfies the original specification.

Proving program correctness requires considerable sophistication in both constructing the specification and deriving the proof. This is a very time consuming process, despite partial automation through theorem provers.

An alternative is to deploy formal program *transformation*, using rules that are known to preserve correctness. That is, given a transformation T, if we can prove that:

$$\forall\ P,Q,program\text{: } \{P\}program\{Q\} \Rightarrow \{P\}T(program)\{Q\}$$

then we can deploy T to change programs without any further need to re-prove the changes.

In practice, the deployment of transformation assumes *referential transparency*[Qui64], that is that *substitution of equalities* preserves meaning. So, mathematical or logical techniques are used to establish that transformations establish equality and then the transformations may be applied to localised program fragments.

For functional programs, reasoning about program transformation draws on classical propositional and predicate calculi, set theory, Peano arithmetic and the theory of computing. We will next survey these sources and then carry out a number of proofs of basic transformations for use when we meet the box calculus proper.

3.2 Propositional calculus

Inference Propositional calculus[Nid62] is a system for reasoning about truth formula made up of:

- constants `true` and `false`;
- variables;

– operators such as: ¬ (not), ∧ (and), ∨ (or), ⇒ (implies), ≡ (equivalent);
– `(...)` (brackets)

Proofs are based on *axioms*, that is formula that are always true and *rules of inference* of the form: $\frac{assumptions}{conclusion}$

The proof that a *proposition* is a *theorem*, that is always true, then proceeds by starting from axioms, and established theorems, which have already been proved to be true, and applying rules of inference until the truth or falsity of the proposition is established.

Truth tables We may give semantics to propositional operators in terms of truth tables that spell out explicitly their values for all possible combinations of operands. Figure 1 shows the tables for ¬, ∧, ∨ and ⇒.

X	Y	$\neg X$	$X \wedge Y$	$X \vee Y$	$X \Rightarrow Y$
`false`	`false`	`true`	`false`	`false`	`true`
`false`	`true`		`false`	`true`	`true`
`true`	`false`	`false`	`false`	`true`	`false`
`true`	`true`		`true`	`true`	`true`

Fig. 1. Truth tables for ¬, ∧, ∨ and ⇒

We may then prove a theorem by constructing the truth table to demonstrate that it is true for all combinations or arguments. For example, Figure 2 shows a proof that:

$$X \Rightarrow Y \equiv \neg X \vee Y$$

X	Y	$\neg X$	$\neg X \vee Y$	$X \Rightarrow Y$
`false`	`false`	`true`	`true`	`true`
`false`	`true`	`true`	`true`	`true`
`true`	`false`	`false`	`false`	`false`
`true`	`true`	`false`	`true`	`true`

Fig. 2. $X \Rightarrow Y \equiv \neg X \vee Y$

Rewriting Rewriting involves using proven equivalences of the form:

$$formula_1 \equiv formula_2$$

by substituting instances of $formula_2$ for $formula_1$ and vice versa, consistently replacing common meta-variables. There are many well known equivalences for cancelling out, reordering and expanding/grouping terms - see Figure 3. We will

Constant
$P \wedge \texttt{true} \equiv P$
$P \wedge \texttt{false} \equiv \texttt{false}$

$P \vee \texttt{true} \equiv \texttt{true}$
$P \vee \texttt{false} \equiv P$

Negation
$P \vee \neg P \equiv \texttt{true}$
$P \wedge \neg P \equiv \texttt{false}$

Idempotency
$P \vee P \equiv P$
$P \wedge P \equiv P$

Associativity
$P \vee (Q \vee R) \equiv (P \vee Q) \vee R$
$P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$

Commutativity
$P \vee Q \equiv Q \vee P$
$P \wedge Q \equiv Q \wedge P$
$(P \equiv Q) \equiv (Q \equiv P)$

Distributivity
$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

De Morgan's Laws
$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

Implication
$P \Rightarrow Q \equiv \neg P \vee Q$
$P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P$
$P \wedge Q \Rightarrow R \equiv P \Rightarrow (Q \Rightarrow R)$

Equivalence
$(P \equiv Q) \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$
$(P \equiv Q) \equiv (P \wedge Q) \vee (\neg P \wedge \neg Q)$
$(P \equiv Q) \equiv (P \Rightarrow Q) \wedge (\neg P \Rightarrow \neg Q)$

Fig. 3. Propositional equivalences.

meet many of these forms again when we consider other roots for reasoning about functional programs.

3.3 Predicate calculus

Where propositional calculus is concerned with properties of propositions about truth values, *predicate calculus*[Hod77] is used to reason about properties of some universe of discourse. It extends propositional calculus with:

– constant values from a universe of some *type*;
– predicates capturing properties of values from the universe, from that *type* to boolean;
– functions between values in the universe, from *type* to *type*;

Most important, predicate calculus introduces *quantifiers* for expressing properties of the entire universe. *Universal* quantification (all):

$$\forall var : P(var)$$

states that P holds for all var from the universe. *Existential* quantification (exists):

$$\exists var : P(var)$$

states that P holds for some var from the universe.

Pure predicate calculus is used to establish properties of arbitrary universes and we will not consider it further here. However, we will look at *applied* predicate calculus in more detail.

3.4 Set theory

Set theory[Hal60] formalise properties of sets of constants, characterised either exhaustively or by some predicate. Finite sets are written as:

$$\{element_1, element_2, ...element_N\}$$

where each $element_i$ is some atomic entity. The empty set is $\{\}$. The principle set operations are: $\in$ (member), $\cup$ (union), $\cap$ (intersection), $\subset$ and $\subseteq$ (subset) and $\setminus$ (difference).

Set theory also offers a rich collection of equivalences for cancelling, reordering, expanding and grouping terms in set expressions, summarised in Figure 4.

Constant
$X \cup \{\} \equiv X$
$X \cap \{\} \equiv \{\}$
$X \setminus \{\} \equiv X$

Idempotency
$A \cup A \equiv A$
$A \cap A \equiv A$

Associativity
$A \cup (B \cup C) \equiv (A \cup B) \cup C$
$A \cap (B \cap C) \equiv (A \cap B) \cap C$

Commutativity
$A \cup B \equiv B \cup A$
$A \cap B \equiv B \cap A$

Distributivity
$A \cap (B \cup C) \equiv (A \cap B) \cup (A \cap C)$
$A \cup (B \cap C) \equiv (A \cup B) \cap (A \cup C)$

Fig. 4. Set equivalences.

Set theoretic predicate calculus Quantification may be specialised to specific sets, so:

$$\forall var \in S : P(var)$$

states that P holds for all var in S, and:

$$\exists var \in S : P(var)$$

states that P holds for some var in S.

We may then note the equivalence:

$$\forall var \in S : P(var) \wedge P(s) \equiv \forall var \in S \cup \{s\} : P(var)$$

which states that if P holds for all in S and for s, then P holds for all of S augmented with s.

Similarly

$$\exists var \in S : P(var) \vee P(s) \equiv \exists var \in S \cup \{s\} : P(var)$$

which states that if P holds for some member of S or for s, then P holds for some member of S augmented with s.

3.5 Peano arithmetic

We come even closer to functional reasoning with *Peano arithmetic*[Kne63] which formalises properties of natural numbers, that is numbers greater than or equal to zero. Peano arithmetic is based on constructing numbers from 0 and the successor function *succ* i.e. $succ(X) = X + 1$. The axioms are:

1. 0 is a natural number;
2. if N is a natural number then $succ(N)$ is a natural number;
3. if N is a natural number then $\neg(0 = succ(N))$;
4. if M and N are natural numbers and if $M = N$ then $succ(M) = succ(N)$.

Note that here we use the numeric notion of *equality* rather than boolean equivalence.

Induction Peano arithmetic introduces the fundamental technique of *proof by induction* which underlies all recursive proof techniques. If:

– $P(0)$ can be proved;
– assuming $P(N)$ then $P(succ(N))$ can be proved.

then it may be concluded that P holds for all natural numbers.

Recursive Complementing inductive proof, Peano arithmetic also introduces *recursive* already familiar from functional programming. That is, a recursive function may be defined in terms of:

– *base case* where the argument is 0 and some value is returned;
– *recursion case* where the argument is $succ(N)$ and the function is called with possibly modified N.

For example, we define addition and multiplication in Figure 5.

$$\begin{array}{ll}
X + 0 = X & X * 0 = 0 \\
X + succ(Y) = succ(X + Y) & X * succ(Y) = X + X * Y
\end{array}$$

Fig. 5. Addition and multiplication

Inductive proof We may then use *inductive proof* to establish properties of recursive functions. We first number and state the theorem. We next state and prove the base case, where one argument is 0. Then, we state the recursion case and the assumed *induction hyp.*, where one argument involves *succ*, before proving the recursion case. We write proof steps systematically from one side of the equality to the other of the equality we wish to establish, one step to a line, noting the justification for the step. The justification is usually a reference to the definition of a function, the induction hypothesis or a theorem.

For example:

Theorem 1. $0 + X = X$

Proof. By induction on X

$$\begin{array}{lll}
\text{Base case:} & 0 + 0 = 0 & \\
 & 0 & \rightarrow (+\ ) \\
 & 0 + 0 & \\
\text{Recursion case:} & 0 + succ(X) = succ(X) & \\
\text{Assumption} & 0 + X = X & \text{[induction hyp.]} \\
 & 0 + succ(X) & \rightarrow (+\ ) \\
 & succ(0 + X) & \rightarrow \text{(induction hyp.)} \\
 & succ(X) &
\end{array}$$

For example:

Theorem 2. $X + succ(Y) = succ(X) + Y$

Proof. By induction on Y

$$\begin{array}{lll}
\text{Base case:} & X + succ(0) = succ(X) + 0 & \\
 & X + succ(0) & \rightarrow (+\ ) \\
 & succ(X + 0) & \rightarrow (+\ ) \\
 & succ(X) & \rightarrow (+\ ) \\
 & succ(X) + 0 & \\
\text{Recursion case:} & X + succ(succ(Y)) = succ(X) + succ(Y) & \\
\text{Assumption} & X + succ(Y) = succ(X) + Y & \text{[induction hyp.]} \\
 & X + succ(succ(Y)) & \rightarrow (+\ ) \\
 & succ(X + succ(Y)) & \rightarrow \text{(induction hyp.)} \\
 & succ(succ(X) + Y) & \rightarrow (+\ ) \\
 & succ(X) + succ(Y) &
\end{array}$$

Theorem 3. *$X + Y = Y + X$:*

Proof. By induction on Y

Base case:	$X + 0 = 0 + X$	
	$X + 0$	$\to (+\ )$
	X	$\to$ (Theorem 1)
	$0 + X$	
Recursion case:	$X + succ(Y) = succ(Y) + X$	
Assumption	$X + Y = Y + X$	[induction hyp.]
	$X + succ(Y)$	$\to (+\ )$
	$succ(X + Y)$	$\to$ (induction hyp.)
	$succ(Y + X)$	$\to (+\ )$
	$Y + succ(X)$	$\to$ (Theorem 2)
	$succ(Y) + X$	

Non-inductive Peano arithmetic does not require us to stick to the induction form for defining functions. For example, we give recursive definitions of *comparison* operators in Figure 6, and could prove the usual transitive properties:

$$\begin{array}{ll} 0 < succ(N) & succ(N) > 0 \\ succ(X) < succ(Y) = X < Y & succ(X) > succ(Y) = X > Y \end{array}$$

Fig. 6. Comparison

$$\begin{array}{l} A = B \wedge B = C \Rightarrow A = C \\ A < B \wedge B < C \Rightarrow A < C \\ A > B \wedge B > C \Rightarrow A > C \end{array}$$

We may then use the comparison operators to qualify other definitions. For example, we define subtraction and division in Figure 7[4].

$$\begin{array}{ll} X \leq Y \Rightarrow X - Y = 0 & X < Y \Rightarrow X/Y = 0 \\ X - 0 = X & X/0 = 0 \\ succ(X) - succ(Y) = X - Y & X/Y = succ((X - Y)/Y) \end{array}$$

Fig. 7. Subtraction and division

Arithmetic equivalences We could then prove the standard arithmetic equalities shown in Figure 8, using $1 \equiv succ(0)$.

[4] Note that, to make / total, we define division by 0 to be 0 not $\perp$.

Constant

$X + 0 = X$

$X - 0 = X$

$X * 0 = 0$

$X * 1 = X$

$X/1 = X$

$X/X = 1$

$-(-X) = X$

Associativity

$(A + B) + C = A + (B + C)$

$(A * B) * C = A * (B * C)$

Commutativity

$A + B = B + A$

$A * B = B * A$

Distributivity

$A * (B + C) = A * B + A * C$

$A * (B - C) = A * B - A * C$

Fig. 8. Arithmetic equivalences.

3.6 λ calculus

With recursive function theory[Pet67], Church's λ calculus[Chu36] is the bedrock of functional programming. λ calculus is based on pure abstractions over names, with three very simple expression forms:

- id - variable;
- $(\lambda id.expression)$ - function abstraction: id is the bound variable (formal parameter) and $expression$ is the body;
- $expression_1$ $expression_2$ - function application: $expression_1$ is a function and $expression_2$ is the argument (actual parameter).

β reduction λ expressions are evaluated through a process of substitution of argument expressions for bound variables in function bodies called β reduction. Before we can formulate this, we need to clarify the notions of a variable being *bound* or *free* in an expression.

A variable id is bound in an expression if the expression is:

- $(\lambda id'.expressions)$ - id is the bound variable id' or id is bound in the body $expression$;
- $expression_1$ $expression_2$ - id is bound in $expression_1$ or in $expression_2$.

A variable id is free in an expression if the expression is:

- id' - id is id';
- $(\lambda id'.expressions)$ - id is not id' and id is free in the $expression$;
- $expression_1$ $expression_2$ - id is free in $expression_1$ or in $expression_2$.

Then to β reduce:

$(\lambda id.expression_1)$ $expression_2$

in *normal order*, where the actual parameter $expression_2$ is not evaluated:

– replace all free occurrences of *id* in *expression*$_1$ with *expression*$_2$;
– β reduce the resulting expression.

We indicate a β reduction step with $\rightarrow_\beta$.

For example:

$((\lambda x.\lambda y.x\ y)\ (\lambda s.s\ s))\ (\lambda z.z) \rightarrow_\beta$
$(\lambda y.(\lambda s.s\ s)\ y)\ (\lambda z.z) \rightarrow_\beta$
$(\lambda s.s\ s)\ (\lambda z.z) \rightarrow_\beta$
$(\lambda z.z)\ (\lambda z.z) \rightarrow_\beta$
$\lambda z.z$

For *applicative order* β reduction, the argument *expression*$_2$ is evaluated before substitution in the body *expression*$_1$.

α renaming A potential problem with β reduction lies in *free variable capture* where a free variable in an argument expression, which should not be the site of further substitutions, is moved into the scope of a bound variable with the same identifier, where it may subsequently be replaced. For example, in:

$$((\lambda x.\lambda y.x)\ y)\ (\lambda x.x) \rightarrow_\beta (\lambda y.y)\ (\lambda x.x) \rightarrow_\beta (\lambda x.x)$$

y was free in the original expression but is bound in the reduced expression $\lambda y.y$ and so is replaced by $\lambda x.x$.

To α rename an expression ($\rightarrow_\alpha$), all *id* free in an expression are replaced with a new unique *id*$'$. For example:

$$((\lambda x.\lambda y.x)\ y)\ (\lambda x.x) \rightarrow_\alpha ((\lambda x.\lambda y.x)\ a)\ (\lambda x.x) \rightarrow_\beta (\lambda y.a)\ (\lambda x.x) \rightarrow a$$

Here, $\lambda x.x$ is discarded as there are now no occurrences of the bound variable y in the renamed body a.

α renaming assumes that we have an inexhaustible supply of new names.

η reduction η reduction is a common special case of β reduction:

$$\lambda\ x.f\ x \rightarrow_\eta f$$

where abstracting over applying some function f to some argument x is simply equivalent to just the function f.

Example proof We may use β reduction to carry out equivalence proofs for the λ calculus. For example, given functions to convert between Curried and un-Curried forms:

```
curry f x y = f (x,y)
uncurry f (x,y) = f x y
```

we may show:

Theorem 4. $\texttt{curry}(\texttt{uncurry}f) = f$

Proof.

$$\begin{array}{ll}
\texttt{curry}(\texttt{uncurry}f) & \rightarrow \text{(uncurry)} \\
\texttt{curry}((\lambda f.\lambda(x,y).f\ x\ y)\ f) & \rightarrow_\beta \\
\texttt{curry}(\lambda(x,y).f\ x\ y) & \rightarrow \text{(curry)} \\
(\lambda f.\lambda x.\lambda y.f\ (x,y))(\lambda(x,y).f\ x\ y) & \rightarrow_\beta \\
\lambda x.\lambda y.(\lambda(x,y).f\ x\ y)(x,y) & \rightarrow_\beta \\
\lambda x.\lambda y.f\ x\ y & \rightarrow_\eta \\
\lambda x.f\ x & \rightarrow_\eta \\
f &
\end{array}$$

and:

Theorem 5. $\texttt{uncurry}(\texttt{curry}f) = f$

Proof.

$$\begin{array}{ll}
\texttt{uncurry}(\texttt{curry}\ f) & \rightarrow \text{(curry)} \\
\texttt{uncurry}((\lambda f.\lambda x.\lambda y.f\ (x,y))\ f) & \rightarrow_\beta \\
\texttt{uncurry}(\lambda x.\lambda y.f\ (x,y)) & \rightarrow \text{(uncurry definitiion)} \\
(\lambda f.\lambda(x,y).f\ x\ y)(\lambda x.\lambda y.f\ (x,y)) & \rightarrow_\beta \\
\lambda(x,y).(\lambda x.\lambda y.f\ (x,y))\ x\ y & \rightarrow_\beta \\
\lambda(x,y).(\lambda y.f\ (x,y))\ y & \rightarrow_\beta \\
\lambda(x,y).f\ (x,y) & \rightarrow_\eta \\
f &
\end{array}$$

3.7 Structural induction

Burstall's widely used *structural induction*[Bur69] is a generalisation of McCarthy's recursion induction on recursive functions[McC62] to *compositional* recursive structures, that is structures whose properties may be characterised in terms of properties of their components. For example, lists are defined in terms of the empty list ([]) and the concatenation of a head and a tail ($h : t$), so proving $P(h : t)$ by structural induction involves:

– base case: prove $P([])$;
– recursion case: assume $P(t)$ and prove $P(h : t)$.

As we shall see, structural induction is a mainstay of reasoning about functional programs.

3.8 fold and unfold

The other mainstay of reasoning about functional programs is Burstall and Darlington's fold/unfold approach[BD77]. This is based on five rules:

1. *instantiation*: substitute for actual parameter in function body;

2. *unfolding*: replace function call in expression by equivalent instantiation of function body;
3. *folding*: replace instance of function body by equivalent function call;
4. *abstraction*: introduce `let` (or `where`) by replacing instance with variable and defining variable to be instance.

Rules 1. and 2. are reminiscent of β reduction and rule 3. of its reverse.

Of their fifth rule, termed *laws*, they say:

> "We may transform an equation by using on its right hand side any laws we have about the primitives K,l...(associativity, commutativity etc)...." (p48)

thus advocating use of the equivalences and equalities we have already surveyed.

3.9 Bird-Meertens Formalism

The Bird-Meertens Formalism (BMF)[BdM97] is a general calculi of functional programs. Here we will consider the theory of lists[Bir87], which applies rules drawn from fold/unfold and structural induction to programs built from higher order functions like map, fold, append and compose. [5]

For example, to prove the associativity of `++`:

Theorem 6. *a++$(b$++$c) = (a$++$b)$++c*

Proof. By induction on a

Base case:	$[]$++$(b$++$c) = ([]$++$b)$++c	
	$[]$++$(b$++$c)$	$\rightarrow$ (++)
	b++c	$\rightarrow$ (++)
	$([]$++$b)$++c	
Recursion case:	$(h : t)$++$(b$++$c) = ((h : t)$++$b)$++c	
Assumption	t++$(b$++$c) = (t$++$b)$++c	[induction hyp.]
	$(h : t)$++$(b$++$c)$	$\rightarrow$ (++)
	$h : (t$++$(b$++$c))$	$\rightarrow$ (induction hyp.)
	$h : ((t$++$b)$++$c)$	$\rightarrow$ (++)
	$(h : (t$++$b))$++c	$\rightarrow$ (++)
	$((h : t)$++$b)$++c	

We will now carry out a number of proofs which we will use when we explore the box calculus.

We start with a generic definition of `fold` which applies some function `f` to the head of a list and the result of doing so recursively to the tail of the list, given some initial value `r`:

```
fold :: (b->a->b)->b->[a]->b
fold f r [] = r
fold f r (x:xs) = fold f (f r x) xs
```

[5] Note that in presenting proofs we assume that all variables are universally quantified.

We assume that `f` is associative so:

$$f\ a\ (f\ b\ c) = f\ (f\ a\ b)\ c$$

We next introduce additional functions to take the first `n` elements of a list:

```
take _ [] = []
take 0 xs = []
take (1+n) (x:xs) = x:take n xs
```

drop the first `n` elements of a list:

```
drop _ [] = []
drop 0 xs = xs
drop (1+n) (x:xs) = drop n xs
```

There now follow a number of simple BMF proofs which we will use when we come to consider a substantive box calculus example below.

First of all, we show that `take` and `drop` cancel:

Theorem 7. $\texttt{take}\ n\ xs\texttt{++}\texttt{drop}\ n\ xs = xs$

Proof. By induction on xs

Base case:	$\texttt{take}\ n\ []\texttt{++}\texttt{drop}\ n\ [] = []$	
	$\texttt{take}\ n\ []\texttt{++}\texttt{drop}\ n\ []$	$\rightarrow$ (`take`/`drop`)
	$[]\texttt{++}[]$	$\rightarrow$ (`++`)
	$[]$	
Recursion case:	$\texttt{take}\ n\ (x : xs)\texttt{++}\texttt{drop}\ n\ (x : xs) =$ $x : xs$	
Assumption	$\texttt{take}\ n\ xs\texttt{++}\texttt{drop}\ n\ xs = xs$	[induction hyp.]
By induction on n		
Base case:	$\texttt{take}\ 0\ (x : xs)\texttt{++}\texttt{drop}\ 0\ (x : xs) =$ $x : xs$	
	$\texttt{take}\ 0\ (x : xs)\texttt{++}\texttt{drop}\ 0\ (x : xs)$	$\rightarrow$ (`take`/`drop`)
	$[]\texttt{++}(x : xs)$	$\rightarrow$ (`++`)
	$x : xs$	
Recursion case:	$\texttt{take}\ (y + 1)\ (x : xs)\texttt{++}$ $\texttt{drop}\ (y + 1)\ (x : xs) = x : xs$	
	$\texttt{take}\ (y + 1)\ (x : xs)\texttt{++}$ $\texttt{drop}\ (y + 1)\ (x : xs)$	$\rightarrow$ (`take`/`drop`)
	$(x : \texttt{take}\ y\ xs)\texttt{++}$ $\texttt{drop}\ y\ xs$	$\rightarrow$ (`++`)
	$x : (\texttt{take}\ y\ xs\texttt{++}$ $\texttt{drop}\ y\ xs)$	$\rightarrow$ (induction hyp.)
	$x : xs$	

Note that we could have established the second induction by case analysis with n equal to 0.

We also show:

Theorem 8. $f\ a\ (\texttt{fold}\ f\ r\ b) = \texttt{fold}\ f\ a\ (r : b)$

Proof. By induction on b

$$\begin{array}{lll}
\text{Base case:} & f\ a\ (\texttt{fold}\ f\ r\ []) = \texttt{fold}\ f\ a\ (r : []) & \\
 & f\ a\ (\texttt{fold}\ f\ r\ []) & \rightarrow (\texttt{fold}\ ) \\
 & f\ a\ r & \rightarrow (\texttt{fold}\ ) \\
 & \texttt{fold}\ f\ (f\ a\ r)\ [] & \rightarrow (\texttt{fold}\ ) \\
 & \texttt{fold}\ f\ a\ (r : []) & \\
\text{Recursion case:} & f\ a\ (\texttt{fold}\ f\ r\ (x : b)) = & \\
 & \quad \texttt{fold}\ f\ a\ (r : (x : b)) & \\
\text{Assumption} & f\ a\ (\texttt{fold}\ f\ r\ b) = \texttt{fold}\ f\ a\ (r : b) & [\text{induction hyp.}] \\
 & f\ a\ (\texttt{fold}\ f\ r\ (x : b)) & \rightarrow (\texttt{fold}\ ) \\
 & f\ a\ (\texttt{fold}\ f\ (f\ r\ x)\ b) & \rightarrow (\text{induction hyp.}) \\
 & \texttt{fold}\ f\ a\ ((f\ r\ x) : b) & \rightarrow (\texttt{fold}\ ) \\
 & \texttt{fold}\ f\ (f\ a\ (f\ r\ x))\ b & \rightarrow (\text{f associativity}) \\
 & \texttt{fold}\ f\ (f\ (f\ a\ r)\ x)\ b & \rightarrow (\texttt{fold}\ ) \\
 & \texttt{fold}\ f\ (f\ a\ r)\ (x : b) & \rightarrow (\texttt{fold}\ ) \\
 & \texttt{fold}\ f\ a\ (r : (x : b)) &
\end{array}$$

Finally, we show that `fold` distributes over `++`, assuming that `e` is an identity element for `f` so:

$$f\ \texttt{e}\ x\ =\ x\ =\ f\ x\ \texttt{e}$$

Theorem 9. $\texttt{fold}\ f\ r\ (a{+}{+}b)\ =\ f\ (\texttt{fold}\ f\ r\ a)\ (\texttt{fold}\ f\ \texttt{e}\ b)$

Proof. By induction on a

$$\begin{array}{lll}
\text{Base case:} & \texttt{fold}\ f\ r\ ([]{+}{+}b)\ = & \\
 & \quad f\ (\texttt{fold}\ f\ r\ [])\ (\texttt{fold}\ f\ \texttt{e}\ b) & \\
 & \texttt{fold}\ f\ r\ ([]{+}{+}b) & \rightarrow (\texttt{++}\ ) \\
 & \texttt{fold}\ f\ r\ b & \rightarrow (\texttt{e}\ \text{identity}) \\
 & \texttt{fold}\ f\ (f\ r\ \texttt{e})\ b & \rightarrow (\texttt{fold}\ ) \\
 & \texttt{fold}\ f\ r\ (\texttt{e} : b) & \rightarrow (\text{Theorem 8}) \\
 & f\ r\ (\texttt{fold}\ f\ \texttt{e}\ b) & \rightarrow (\texttt{fold}\ ) \\
 & f\ (\texttt{fold}\ f\ r\ [])\ (\texttt{fold}\ f\ \texttt{e}\ b) & \\
\text{Recursion case:} & \texttt{fold}\ f\ r\ ((x : a){+}{+}b)\ = & \\
 & \quad f\ (\texttt{fold}\ f\ r\ (x : a))\ (\texttt{fold}\ f\ e\ b) & \\
\text{Assumption} & \texttt{fold}\ f\ r\ (a{+}{+}b)\ = & \\
 & \quad f\ (\texttt{fold}\ f\ r\ a)\ (\texttt{fold}\ f\ \texttt{e}\ b) & [\text{induction hyp.}] \\
 & \texttt{fold}\ f\ r\ ((x : a){+}{+}b) & \rightarrow (\texttt{++}\ ) \\
 & \texttt{fold}\ f\ r\ (x : (a{+}{+}b)) & \rightarrow (\texttt{fold}\ ) \\
 & \texttt{fold}\ f\ (f\ r\ x)\ (a{+}{+}b) & \rightarrow (\text{induction hyp.}) \\
 & f\ (\texttt{fold}\ f\ (f\ r\ x)\ a)\ (\texttt{fold}\ f\ \texttt{e}\ b) & \rightarrow (\texttt{fold}\ ) \\
 & f\ (\texttt{fold}\ f\ r\ (x : a))\ (\texttt{fold}\ f\ \texttt{e}\ b) &
\end{array}$$

We will see in subsequent sections both how the BMF enables us to reason about computational aspects of parallel programs and its limitations in accounting for pragmatic effects of parallel program transformation.

4 Hume coordination layer

We have met the Hume expression layer as a pure functional programming language and surveyed techniques for reasoning about functional programs. We will now look at the Hume coordination layer before considering requirements for reasoning about coordination in the next section.

4.1 Boxes and wires

As we noted above, Hume programs are built from *boxes* connected to each other, the wider environment and themselves by input and output *wires*.

Boxes are generalised finite state automata which pattern match on their inputs and generate corresponding outputs with expression layer constructs. Boxes are repeated one-shot and stateless. So a box can be thought of as a non-terminating while loop which loses all the values of its local variables on each iteration.

Wires connect input and output *links* on boxes and *streams* to the operating environment. Wires are uni-directional, single buffered FIFOs which can hold any matchable construct. They retain information between box iterations and so are the sole locus of state in Hume programs.

4.2 Execution model

Boxes execute repeatedly for ever in a two-phase execution cycle. In the *local* phase, each box attempts to match its inputs and generate outputs. Then in the *global* super-step phase, the consumption of input values from, and assertion of output values to, wires is resolved.

At the start of each execution cycle a box may be:

– *READY*: all outputs from the previous cycle have been consumed and so new inputs may be sought;
– *BLOCKED*: some outputs from the previous cycle have not been consumed and so new outputs cannnot be generated.

Then the execution model is:

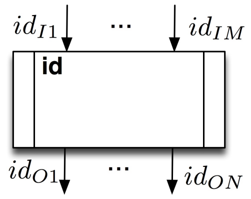
```
for each box:
  STATE $\leftarrow$ $READY$
forever
  for each $READY$ box:
    if match inputs then:
      consume inputs from wires
      generate and buffer outputs
      STATE $\leftarrow$ $BLOCKED$
  for each $BLOCKED$ box:
    if previous outputs consumed then:
      assert outputs from buffer on wires
      STATE $\leftarrow$ $READY$
```

In the model, the local and global phases may be conducted concurrently with an intervening barrier synchronisation.

4.3 Box, wire and steam declarations

Box declaration A box is declared by:

box *id*
in (id_{I1}::$type_{I1}$,...id_{IM}::$type_{IM}$)
out (id_{O1}::$type_{O1}$,...id_{ON}::$type_{ON}$)
match
  $pattern_1$ -> $expression_1$ |
...
  $pattern_P$ -> $expression_P$;



id is the name of the box, and $id_{Ix}/type_{Ix}$ and $id_{Oy}/type_{Oy}$ are the names and types of the input and output links.

All the $pattern_i$ must have the same type as the input links and all the $expression_i$ must have the same type as the output links.

Note that the link name space and the pattern name space are disjoint so the same identifiers may be used in both.

Wire declaration A wire declaration takes the form:

wire *id* ($link_{I1}$,...$link_{IM}$) ($link_{O1}$,...$link_{ON}$);

id is the name of the box, and $link_{Ix}$ and $link_{Oy}$ are the names of the links (and streams) to which the corresponding box inputs and outputs are connected.

A link name may be $box_{id}.in - out_{id}$, where box_{id} is the name of a box and $in - out_{id}$ is the name of one of that box's input or output links, or the name of a stream $stream_{id}$.

All wires must make type-consistent connections.

Wire initialisation Wires which are not connected to an environmental input may require an initial value to enable pattern matching to commence. This may be achieved by using a wiring link of the form:

id.id initially *constant*

Stream declaration Streams convey character sequences from and to the operating environment. In principle they may be associated with arbitrary sources and sinks but currently only files and sockets are supported.

Streams are declared by:

input stream: stream *id* from "*path*";
output stream: stream *id* to "*path*";

where *id* is the stream name and *path* is a path.

Automatic input/output Stream text is automatically converted to and from appropriate representations for any bounded type associated with a box link. For an input stream, the type is used as a grammar to parse the text to the corresponding value. For an output stream, the type is used to guide flat, unbracketed pretty printing of the value. The conventions are the same as for the expression layer type conversion: *expression* `as string`.

4.4 Examples

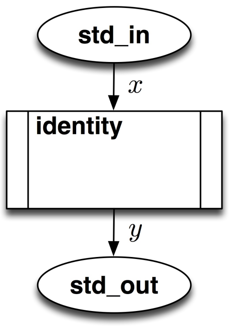
Consider a box that copies input from the keyboard to output on the display without change:

```
1. type integer = int 32;

2. box identity
3. in (x::integer)
4. out (y::integer)
5. match
6.  p -> p;

7. stream input from "std_in";
8. stream output to "std_out";

9. wire identity (input) (output);
```



where line numbers are purely to aid narrative.

Lines 2 to 6 declare a box called `identity` with input link `x` and output link `y` which both carry `integer`s as declared in line 1. The match in line 6 indicates that pattern `p` matches input on `x` to generate the corresponding value on output `y`.

Lines 7 and 8 declare the streams `input` and `output` connected to standard input and standard output respectively.

Line 9 wires link `identity.x` to `input` and `identity.y` to `output`.

When the program is run, the interaction is as follows:

```
$ identity
1
1 2
2 3
3...
```

We may next change the program to generate squares of successive outputs on new lines:

```
1. type integer = int 32;

2. sq::integer -> integer;
3. sq x = x*x;
```

```
 4. box square2
 5. in (x::integer)
 6. out (y::(integer,char))
 7. match
 8.  p -> (sq p,'\n');

 9. stream input from "std_in";
10. stream output to "std_out";

11. wire square2 (input) (output);
```

Now, output from line 8 is a tuple of a square and a newline character so the type of the output link on line 6 changes accordingly.

Interaction is now as:

```
$ square2
1
1
2
4
3
9...
```

Finally, consider inputting a simple sum of the form: *number operator number* where *operator* is one of `+`,`-`,`*` or `/`, and displaying the sum and result:

```
box sums
in (xy::integer,char,integer)
out (s::(integer,char))
match
 (x,'+',y) -> (x+y,'\n') |
 (x,'-',y) -> (x-y,'\n') |
 (x,'*',y) -> (x*y,'\n') |
 (x,'/',y) -> (x div y,'\n') ;
```

so interaction is as:

```
$ sums
1 + 1
2
6 / 2
3
4 * 8
32 ...
```

4.5 Feedback wiring

It is often useful to wire a box back to itself to maintain intermediate state.

For example, to generate successive integers from 0:

```
box gen
in (n::integer)
out (n'::integer,s::(integer,char))
match
 (x) -> (x+1,(x,'\n'));

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

Here, input `n` is wired to `n'` and initialised to `0`. Note that output `n'` is also wired back to `n`.

On each execution cycle, `n` gets the next integer from `n'`, sends `n+1` to `n'` and outputs `n`:

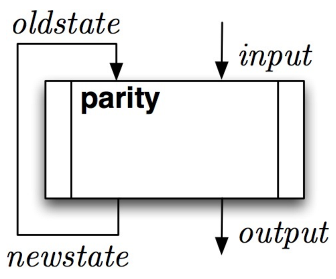
```
$ gen
0
1
2
...
```

For example, consider parity checking a sequence of bits to show at each stage if there is an odd or even number of `1`s:

```
type BIT = word 1;

data STATE = ODD | EVEN;

box parity
in (oldstate::STATE,input::BIT)
out (newstate::STATE,output::string)
match
 (ODD,0) -> (ODD,"ODD\n") |
 (ODD,1) -> (EVEN,"EVEN\n") |
 (EVEN,0) -> (EVEN,"EVEN\n") |
 (EVEN,1) -> (ODD,"ODD\n") ;
```



```
wire parity
(parity.newstate initially EVEN, input)
(parity.oldstate, output);
```

Here, `oldstate` and `newstate` are wired reflexively to each others, with `newstate` initialised to `EVEN`.

On each execution cycle, the output to `newstate` flips if the input value is a `1`. This which runs as:

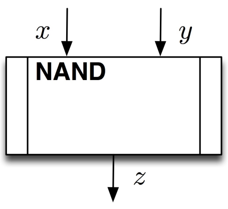
```
$ parity 1 0 1 1 0
ODD
ODD
EVEN
```

ODD
ODD
...

4.6 From one box to multiple boxes

Consider a box to find $X\ NAND\ Y$:

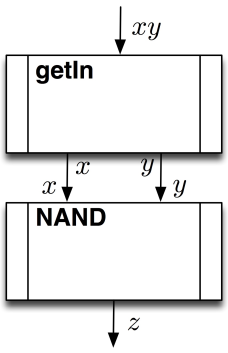
```
box NAND
in (x::BIT,y::BIT)
out (z::BIT)
match
 (0,0) -> 1 |
 (0,1) -> 1 |
 (1,0) -> 1 |
 (1,1) -> 0;
```



We could wire this into a test program that receives a pair of bits from a single wire from standard input and passes them on two wires to `NAND`:

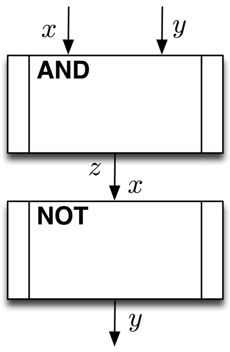
```
box getIn
in (xy:(BIT,BIT))
out (x::BIT,y::BIT)
match
 (x,y) -> (x,y);

wire getIn (input) (NAND.x,NAND.y);
wire NAND (getIN.c,getIn.y) (output);
```



We might, as an alternative, implement this as an *AND* box:

```
box AND
in (x::BIT,y::BIT)
out (z::BIT)
match
 (0,0) -> 0 |
 (0,1) -> 0 |
 (1,0) -> 0 |
 (1,1) -> 1;
```



feeding a *NOT* box:

```
box NOT
in (x::BIT)
out (y::BIT)
match
 0 -> 1 |
 1 -> 0;
```

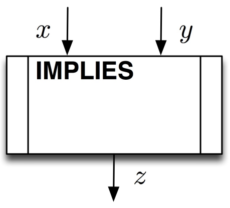
wired with:

```
wire getIn (input) (And.x,AND.y);
wire AND(getIn.x,getIn.y) (NOT.x);
wire NOT (AND.x) (output);
```

In a later section, we will look at using the box calculus to justify this change.

Now, consider a program for *IMPLIES*:

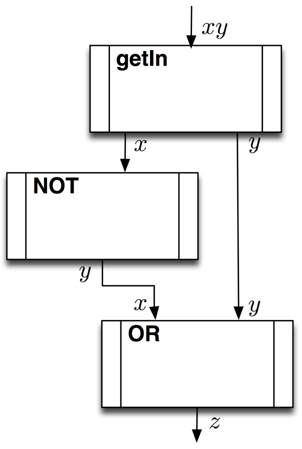
```
box IMPLIES
in (x::BIT,y::BIT)
out (z::BIT)
match
 (0,0) -> 1 |
 (0,1) -> 1 |
 (1,0) -> 0 |
 (1,1) -> 1;
```



$X \Rightarrow Y$ is equivalent to $\neg X \vee Y$, so we could also implement this as an *OR* box fed with a *NOT*ed X and unchanged Y:

```
box OR
in (x::BIT,y::BIT)
out (z::BIT)
match
 (0,0) -> 0 |
 (0,1) -> 1 |
 (1,0) -> 1 |
 (1,1) -> 1;

wire getIn (input) (NOT.x,OR.y);
wire NOT (getIn.x) (OR.x);
wire OR (NOT.y,getIN.y) (output);
```



Again, in a later section, we will look at using the box calculus to justify this change.

4.7 * pattern and expression

The programs we've considered so far are synchronous where boxes execute in locked-step. To enable asynchronicity, Hume introduces the * pattern and expression.

The * pattern ignores its input. If there is no input then the match succeeds and if there is input then the match succeeds but the input is not consumed. The * pattern may only be used in box matches, not in functions.

Similarly, the * expression ignores its output. This may be used in expressions returning final values from box transitions, including in conditional and case expressions, and in function bodies. However, there is no associated value and so it is meaningless to attempt to pass a * as, say, an actual parameter to a function call.

4.8 Recursive functions and iterative boxes

An important feature of boxes is that all space is retrieved on each execution cycle. If space is at a premium, for example in an embedded application, then it may be advantageous to convert stack consuming recursion within a box to constant space iteration using a feedback wire.

For example, consider finding the sum of the integers from 1 to N:

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

This may be re-written to use an accumulation variable:

```
sum' s 0 = s;
sum' s N = sum' (s+N) (N-1);
```

```
sum N = sum' 0 N;
```

where the partial sum `s` is passed from call to call. In turn, this is equivalent to the iteration:

```
sum(N)
{  int s;
   s = 0;
   while(N>0)
   {  s = s+N;
      N = N-1;
   }
   return s;
}
```

where `s` is the partial sum and `N` is the next value to be summed.

We may reformulate this as a box with feedback wires for the partial sum and the next value. Suppose the inputs are `i` for the original input, `s` for the partial sum and `N` for the next value. Suppose the outputs are `o` for the final result, `s'` for the incremented partial sum and `N'` for the decremented next value. Then we may distinguish three possibilities:

1. there is a new initial value with no partial sum or current value. The sum is initialised to 0, the current value is initialised to the initial value, and there is no final output:
   `(i,*,*) -> (*,0,i)`
2. the current value is 0. Any next initial value is ignored. The partial sum is the final output and there are no new values for the partial sum or current value:
   `(*,s,0) -> (s,*,*)`
3. the current value is not 0. Any next initial value is ignored. The current value is added to the partial sum and the current value is decremented, with no final output:
   `(*,s,N) -> (*,s+N,N-1)`

Thus, the final program is:

```
box itersum
in (i::integer,s::integer,N::integer)
out (o::integer, s'::integer,N::integer)
match
 (*,s,0) -> (s,*,*) |
 (*,s,N) -> (*,s+N,N-1) |
 (i,*,*) -> (*,0,i);

wire itersum
(input,itersum.s',itersum.N')
(output,itersum.s,itersum.N);
```

Note the order of the matches. We start with the "termination case" when N is 0, followed by the "iteration case" when N is non-zero, followed by the case for a new input.

5 The box calculus

The box calculus[GM08,Gro09,GM11] contains rules for transforming the coordination layer of a Hume program. In most cases this involves changes to the expression layer through functional reasoning techniques, especially rewriting.

5.1 Rules of the calculus

We will now describe the rules used in the examples in the next section. Some of these rules are atomic, i.e. they can be seen as the axioms, while other are derived. Note that some of them have rather complicated formulations for which we will not give the formal syntax and semantics in full detail – for this we refer to [GM11].

Rename The simplest family of rules are those that just rename a component. The calculus has two such rules: **Rename** which renames a box; and **RenameWire** which renames a given input or output wire of a box. Renaming of functions is considered to be independent of the calculus, and can be incorporated by the **ESub** rule.

Expression substitution (ESub) The first rule of the calculus enables the use of the BMF reasoning discussed in Section 3.9. For any match

```
p -> e1
```

if we can show that

$$\texttt{e1} = \texttt{e2}^{6}$$

then we can replace `e1` with `e2`, which results in the following new match

```
p -> e2
```

Expression/function folding A special, but very common, case of the **ESub** rule is folding and unfolding as discussed in 3.8. Here we may create or delete functions during this process. Assume you have a match of the following form:

```
p -> e p
```

We then fold `e` into a function `f` (with a parameter `x`):

[6] We rely on extensional equality, meaning that two function `f` and `g` are the same if they return the same values: $\forall x.\ \texttt{e1}\ x = \texttt{e2}\ x$.

```
  f x = e x;
```

It is then trivial to show that `f` $x =$ `e` x, thus creating the new match:

```
  p -> f p
```

This rule, which creates a new function `f` from the expression and replaces the expression with a call to `f`, is called **expression folding introduction (EFoldI)**.

Its dual, which unfold the function definition of `f` in the expression is called **expression folding elimination (EFoldE)**.

Note that in applications where **EFoldI** creates a function which is equivalent to an existing function, then a new function is created. A subsequent step can then replace this new function with the existing one using `ESub` and BMF reasoning.

Match composition We can fold a set of adjacent matches

```
  p1 -> e1 | p2 -> e2 | ... | pn -> en
```

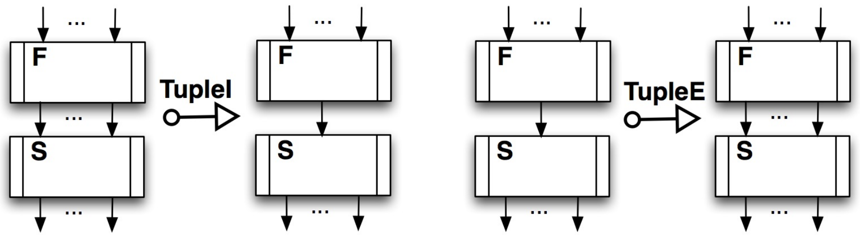
into one match with a case expression for each match

```
 p -> case p of  p1 -> e1 | p2 -> e2 | ... | pn -> en
```

if and only if `p` will always match (and consume) whenever `p1`, $\cdots$, `pn` will. This rule is called **match composition introduction (MCompI)**. A special case of this rule is when the `*` pattern is not used in any pattern, and the matches are *total* (will never fail if input are available on all wires). In this case we can fold all matches into one match,which we will do in several of the example below.

The dual of this rule, **match composition elimination (MCompE)** turns a case expression into a match for each case. The same precondition applies to this rule.

Tuple composition If there are more than one wire from a box to another then these wires can be combined into a single wire containing a tuple, where each element of this wire correspond to one of the original wires. A proviso for this is that the `*` pattern is used either for each or none of the tuple elements for each expression/pattern. This rule is called **tuple composition introduction (TupleI)**, and is illustrated on the left below:

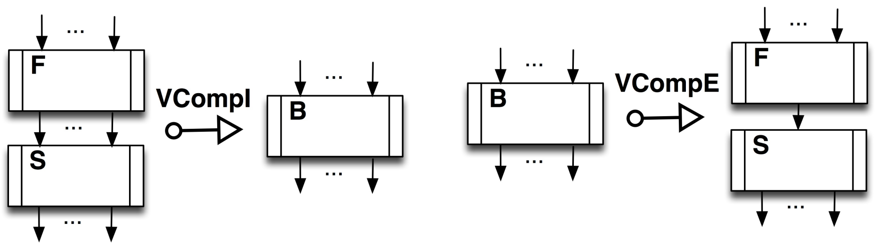


In order to apply the rule the patterns of `S` must change accordingly, and so must the expression of `F`. For simplicity, we can only apply this rule when each expression of `F` can be decomposed syntactically (e.g. the expression cannot be a function application)

Dually, **tuple composition elimination (TupleE)** is the process of splitting a tuple into mant wires – one for each element of the tuple. This is illustrated on the right above.

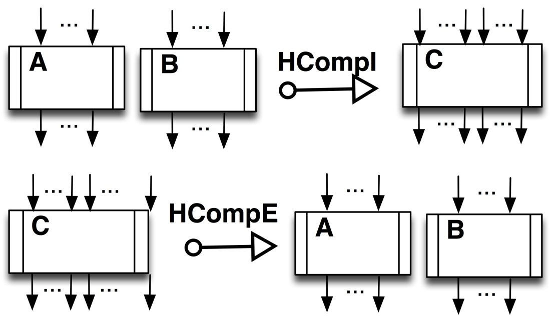
Vertical box composition **Vertical box composition introduction (VCompI)** lifts function composition to the box level, and dually, **vertical box composition elimination) (VCompE)** lifts function de-composition into the box level.

VCompI takes two connected boxes and turn them into one. A proviso for such transformation is that all outputs of the first box are connected to the inputs of the second[7]. This is shown on the left hand side below:



The proviso for it's dual, **VCompE**, is that the box being transformed only has one match which is of the form of a function composition (i.e. `f O g`). This is illustrated on the r.h.s. above.

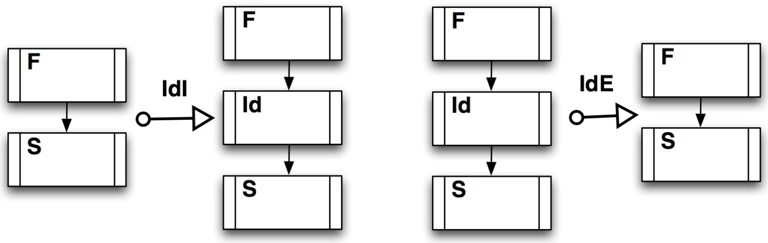
Horizontal box composition Two independent boxes can be joined together if it is never the case that one of them is blocked when the other is not. This is illustrated on the top below, where `A` and `B` are composed into box `C`. This rule is called **horizontal box composition introduction (HCompI)**.



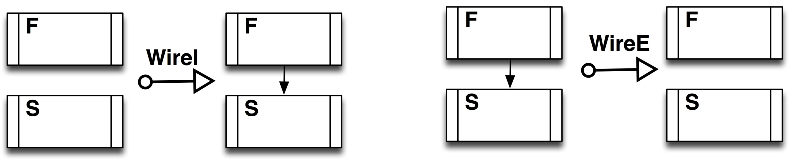
[7] This can in principle be relaxed but will add more complexities.

Its dual, **horizontal box composition elimination (HCompE)**, shown on the bottom above, decomposes a box `C` into two separate boxes `A` and `B`. In order to do this, we need to be able decompose input/output wires of the box so that they are independent of each others, as well as ensuring that the boxes blocks at the same time. This principle is best illustrated by example, which we will show in the following section.

Identity boxes An identity box is a box with one input and one output with one match of the form `x -> x`, that is a box which introduces a one step latency for one wire. If such delay has no impact on the rest of the program, an arbitrary number of identity boxes can be introduced and eliminated, called **identity box introduction (IdI)** and **identity box elimination (IdE)** respectively:

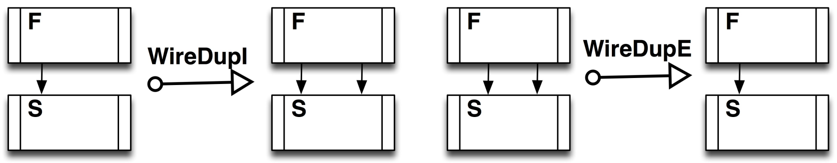


Introducing wires Wires which does 'nothing' can be eliminated. Syntactically, these are cases where all the source expressions and target patterns only contains *. Semantically, this can be generalised to cases where there are not only *, but it is provable that this behaves like *. This rule is known as **wire elimination (WireE)** and is shown on the right below.



Dually, we can introduce a wire where both the source and targets are only *s without changing the semantics of the program. This rule is known as **wire introduction (WireI)** and is shown in the left above.

Wire duplication We can also duplicate wires and eliminate such duplications, known as **wire duplication introduction (WireDupI)** and **wire duplication elimination (WireDupE)**, shown on the left and right below, respectively.



In these cases we must show that the wires are indeed proper duplications, i.e. initially the wires are the same, the same values are always written and consumed.

5.2 A note on preconditions

We have informally discussed some of the preconditions of applying the rules, however there are additional complications in the presence of asynchronous wires (the use of `*`) due to the concurrent nature of the box scheduling. To illustrate, consider a box with matches

```
1.  (x,y) -> ... |
2.  (*,y) -> ... | ...
```

In a configuration where the match at line 1 always succeeds, introduction of an identity box on the first input wire may result in the value on this wire arriving a step later, which may cause match 2 to succeed instead. This can have global impact on correctness. The `*` pattern is not used for any examples in the following section, so we have ignored this potentially complicated issue.

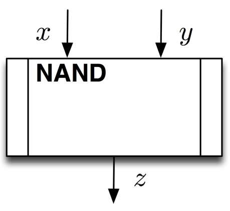
6 Reasoning about Hume programs with the box calculus

In this section we will illustrate use of the box calculus. Sections 6.1 and 6.2 use the calculus for the transformations informally shown in Section 4.6, while Section 6.3 shows how to parallelise the `fold` function for multi-core applications. This examples uses many of the properties we proved about `fold` in Section 3.9.

6.1 Decomposing NAND into AND and NOT

Consider again the `NAND` box from Section 4.6:

```
box NAND
in (x::BIT,y::BIT)
out (z::BIT)
match
 (0,0) -> 1 |
 (0,1) -> 1 |
 (1,0) -> 1 |
 (1,1) -> 0;
```



Firstly note that all cases are handled by the matches. Our first transformation step composes all matches into one match (with a `case` expression) by applying the **MCompI** rule. This creates the following box:

```
box NAND
in (x::BIT,y::BIT)
out (z::BIT)
match
   (x,y) -> case (x,y) of
                  (0,0) -> 1 |
                  (0,1) -> 1 |
                  (1,0) -> 1 |
                  (1,1) -> 0;
```

Next we fold the case expression into a function using the **EFoldI** rule, which creates the new function[8]:

```
nand (x,y) =  case (x,y) of
                  (0,0) -> 1 |
                  (0,1) -> 1 |
                  (1,0) -> 1 |
                  (1,1) -> 0;
```

and the new box:

```
box NAND
in (x::BIT,y::BIT)
out (z::BIT)
match
   (x,y) -> nand (x,y);
```

Using BMF style reasoning, similar to **MCompE** at the expression layer, the case expression can be replaced by pattern matching:

```
nand (0,0) = 1;
nand (0,1) = 1;
nand (1,0) = 1;
nand (1,1) = 0;
```

Logical AND can be represented by the following function:

```
and (0,0) = 0;
and (0,1) = 0;
and (1,0) = 0;
and (1,1) = 1;
```

[8] Strict calculus use is unlikely to come up with names that “makes sense”, so many steps are followed by a renaming application. However, to ease the reading we are using more descriptive names directly.

and logical NOT by the following function:

```
not 0 = 1;
not 1 = 0;
```

Next, we can see that NAND is the same as AND followed by NOT

Theorem 10.

$$\mathit{nand}(x, y) = \mathit{not}(\mathit{and}(x, y))$$

Proof. This equality can easily be proven by drawing up the truth table:

x	y	nand (x,y)	and(x,y)	not(and(x,y))
0	0	1	0	1
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

With Theorem 10 we can apply the **ESubst** rule to replace `nand (x,y)` by `not(and(x,y))` in the `NAND` box:

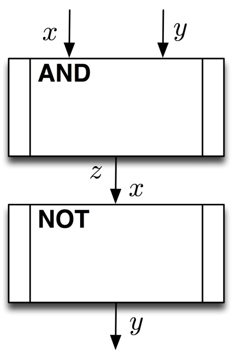
```
box NAND
in (x::BIT,y::BIT)
out (z::BIT)
match
   (x,y) -> not(and (x,y));
```

`not(and(x,y))` is the same as `(not o and) (x,y)`, thus we can apply the **VCompE** rule to decompose this box into two sequential boxes. We then rename the boxes (by rule **Rename**) to `AND` and `NOT`, and input and output wires (using **RenameWire**), which creates the following configuration:

```
box AND
in (x::BIT,y::BIT)
out (z::BIT)
match
   (x,y) -> and (x,y) ;

box NOT
in (x::BIT)
out (y::BIT)
match
   x -> not x;

wire AND(...) (NOT.x);
wire NOT (AND.z) (...);
```



Next we unfold the definition of the `not` function into a case expression using BMF style reasoning

```
not x = case x of 0 -> 1 | 1 -> 0;
```

and show that it is identical to the original (pattern matching) version. We then unfold this function in the `NOT` box using the **EFoldE** rule:

```
box NOT
in (x::BIT)
out (x'::BIT)
match
   x -> case x of 0 -> 1 | 1 -> 0;
```

and split the `case` expression up into several matches using the **MCompE** rule:

```
box NOT
in (x::BIT)
out (x'::BIT)
match
  0 -> 1 |
  1 -> 0;
```

The exact same strategy can be applied to the `AND` box, giving:

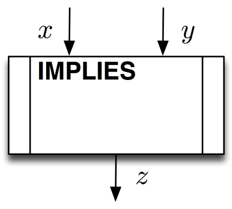
```
box AND
in (x::BIT,y::BIT)
out (z::BIT)
match
   (0,0) -> 0 |
   (0,1) -> 0 |
   (1,0) -> 0 |
   (1,1) -> 1;
```

which completes the transformation.

6.2 Decomposing IMPLIES into NOT and OR

We will now apply the box calculus to the transformation of IMPLIES into NOT followed by OR as illustrated in Section 4.6. We start with the IMPLIES box:

```
box IMPLIES
in (x::BIT,y::BIT)
out (z::BIT)
match
 (0,0) -> 1 |
 (0,1) -> 1 |
 (1,0) -> 0 |
 (1,1) -> 1;
```



First we apply the **MCompI** rule to turn the matches into a `case` expression. We then apply the **EFoldI** rule to separate this `case` expression into a function we call `implies` and unfolds the `case` expression in the function using BMF reasoning. As a result the `IMPLIES` box is replaced by the following box:

```
box IMPLIES
in (x::BIT,y::BIT)
out (z::BIT)
match
   (x,y) -> implies (x,y);
```

which uses the function:

```
implies (0,0) = 1;
implies (0,1) = 1;
implies (1,0) = 0;
implies (1,1) = 1;
```

Next, we define OR in the expression layer as a function:

```
or (0,0) = 0;
or (0,1) = 1;
or (1,0) = 1;
or (1,1) = 1;
```

and introduce the following auxiliary function:

```
negatefirst (0,0) = (1,0);
negatefirst (0,1) = (1,1);
negatefirst (1,0) = (0,0);
negatefirst (1,1) = (0,1);
```

We then show that

Theorem 11.

$$\mathtt{implies}(x,y) = \mathtt{or}(\mathtt{negatefirst}(x,y))$$

Proof. This can be shown by a truth table:

x	y	implies (x,y)	negatefirst(x,y)	or(negatefirst(x,y))
0	0	1	(1,0)	1
0	1	1	(1,1)	1
1	0	0	(0,0)	0
1	1	1	(0,1)	1

By applying Theorem 11 we can replace `implies` with `or(negatefirst(x,y))` in the `IMPLIES` box using the **ESubst** rule:

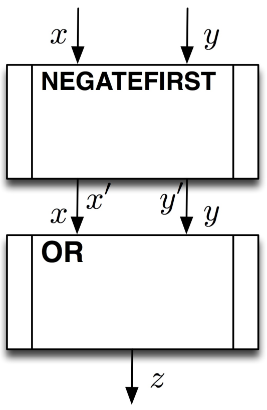
```
box IMPLIES
in (x::BIT,y::BIT)
out (z::BIT)
match
   (x,y) -> or (negatefirst (x,y));
```

We then apply sequential decomposition (**VCompE**):

```
box NEGATEFIRST
in (x::BIT,y::BIT)
out (x'::BIT,y'::BIT)
match
   (x,y) -> negatefirst (x,y);

box OR
in (x::BIT,y::BIT)
out (z::BIT)
match
   (x,y) -> or (x,y);

wire NEGATEFIRST (..) (OR.x, OR.y);
wire OR (NEGATEFIRST.x', NEGATEFIRST.y') (..);
```



Next, we apply BMF reasoning to replace the patterns in the `or` function with a `case` expression. We then unfold this function in the `OR` box by applying the **EFoldE** rule. Then we move the `case` expression into the match with the **MCompE** rule, creating the following new `OR` box:

```
box OR
in (x::BIT,y::BIT)
out (z::BIT)
match
   (0,0) -> 0 |
   (0,1) -> 1 |
   (1,0) -> 1 |
   (1,1) -> 1;
```

We now want to transform the `NEGATEFIRST` box. Firstly, we observe that the first argument is always negated. We then observe that the second argument is left unchanged. Thus, we have the following fact:

Theorem 12.

$$\mathtt{negatefirst}\ (x, y) = (\mathtt{not}\ x, y)$$

Proof. This can be easily shown by a truth table.

Using Theorem 12 we apply **ESub** to the `NEGATEFIRST` box, which gives us the following box:

```
box NEGATEFIRST
in (x::BIT,y::BIT)
out (x'::BIT,y'::BIT)
match
   (x,y) -> (not x,y);
```

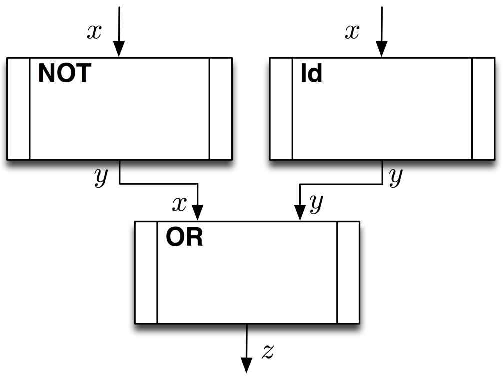
We then observe that the box has one match with two arguments, and the expression can be decomposed such that the first argument in the expression only uses the first input, and the second only uses the second input. This means that we can horizontally decompose the box into two parallel boxes using the **HCompE** rule. We then rename the boxes accordingly (using the **Rename** rule), creating the following box configuration:

```
box NOT
in (x::BIT)
out (y::BIT)
match
   x -> not x;

box Id
in (x::BIT)
out (y::BIT)
match
   y -> y;

box OR ...

wire NOT (..) (OR.x);
wire Id (..) (OR.y);
wire OR (NOT.y,Id.y) (..);
```



Next we unfold the definition of the `not` function into a case expression using BMF style reasoning and show that it is identical to the original (pattern matching) version. We then unfold this function in the `NOT` box using the **EFoldE** rule, before we move the `case` expression into the match by **MCompE** creating the new `NOT` box:

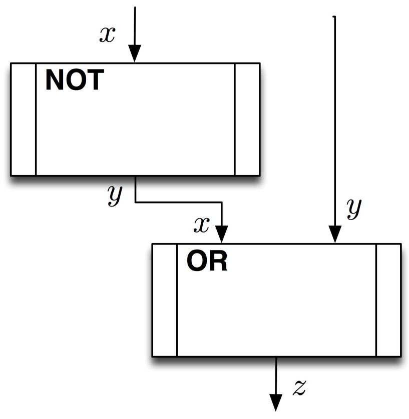
```
box NOT
in (x::BIT)
out (y::BIT)
match
  0 -> 1 |
  1 -> 0;
```

Finally, we observe that, as the name implies, the `Id` box behaves as an identity box and can therefore be eliminated by the **IdE** rule. We then have the following box configuration:

```
box NOT
in (x::BIT)
out (y::BIT)
match
  0 -> 1 |
  1 -> 0;

box OR
in (x::BIT,y::BIT)
out (z::BIT)
match
   (0,0) -> 0 |
   (0,1) -> 1 |
   (1,0) -> 1 |
   (1,1) -> 1;

wire NOT (..) (OR.x);
wire OR (NOT.y,..) (..);
```



which completes the transformation.

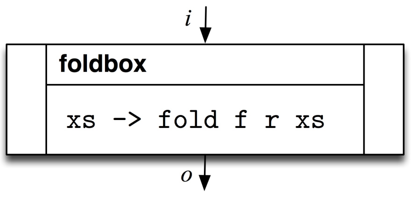
6.3 Parallelising `fold`

Our final application of the box calculus relates to the very timely problem of parallelising programs, for example to explore multi-core architectures. We will address the problem of parallelising the `fold` combinator which we described previously. We make the same assumptions about the function being folded as in Section 3.9 and we utilise several of the theorems proved there.

Two-box fold First we will address the problem of splitting one fold into two parallel applications, and after that generalise to N boxes. Obviously, there is a cost of parallelisation, thus this would only make sense when the function `f :: a -> b -> b` which we fold over performs some heavy and time-consuming computations over a list `xs` and we want to fold the result of each computation. Assuming an initial value `r::b`, we start with the following box:

```
box foldbox
in (i :: [a])
out (o ::  b)
match
  xs -> fold f r xs;
```

Henceforth we will not give such Hume code for boxes and separate diagrams, but integrate the matches into the diagram. In this notation the `foldbox` looks as follows:



Firstly, we define some abbreviations to make the code more readable :

```
  append' (xs,ys) = append xs ys;
  left xs = take ((Length xs) div 2) xs;
  right xs = drop (Length xs) div 2) xs;
  split2 xs = (left xs, right xs);
```

With these definition it follows from Theorem 7 that:

$$\texttt{fold}\ f\ r\ xs = \texttt{fold}\ f\ r\ (\texttt{append'}(\texttt{left}\ xs, \texttt{right}\ xs))$$

Then, from Theorem 9 we have that:

$$\texttt{fold}\ f\ r\ (\texttt{append'}(\texttt{left}\ xs, \texttt{right}\ xs)) =$$

$$f\ (\texttt{fold}\ f\ r\ \texttt{left}\ xs)\ (\texttt{fold}\ f\ e\ \texttt{right}\ xs)$$

Thus, by transitivity of = we know that

$$\texttt{fold}\ f\ r\ xs = f\ (\texttt{fold}\ f\ r\ (\texttt{left}\ xs))\ (\texttt{fold}\ f\ e\ (\texttt{right}\ xs)) \qquad (1)$$

Further from the definition of `uncurry` we know that

$$f\ x\ y = (\texttt{uncurry}\ f)\ (x, y)$$

from before. Using this (1) becomes by BMF:

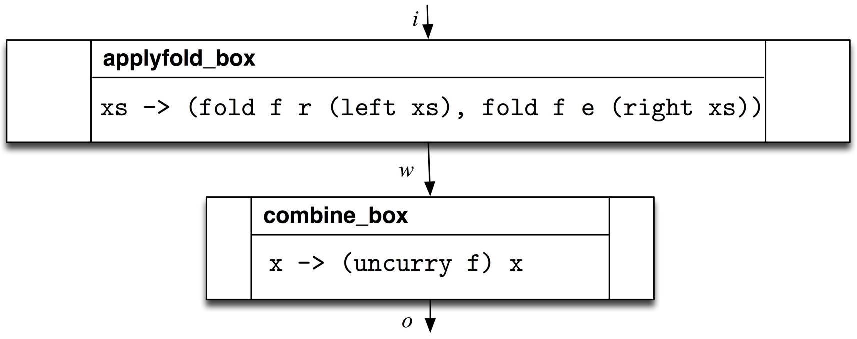
$$\texttt{fold}\ f\ r\ xs = (\texttt{uncurry}\ f)\ (\texttt{fold}\ f\ r\ (\texttt{left}\ xs),\ \texttt{fold}\ f\ e\ (\texttt{right}\ xs))$$

Using this fact, we apply the **ESub** rule creating the following new match for `foldbox`:

```
  xs -> (uncurry f) (fold f r (left xs), fold f e (right xs))
```

This is sequential application of two function, which we lift to the box level by the **VCompE** rule, together with some box renaming[9]:

[9] Note that a single wire label means that this name is used for both the output of the first box (`applyfold_box`) and input of the second box (`combine_box`).



We know that the output wire of the `applyfold_box` is a pair which can be split by the **TupleE** rule. After this split the match of `combine_box` becomes:

```
(a,b) -> (uncurry f) (a,b)
```

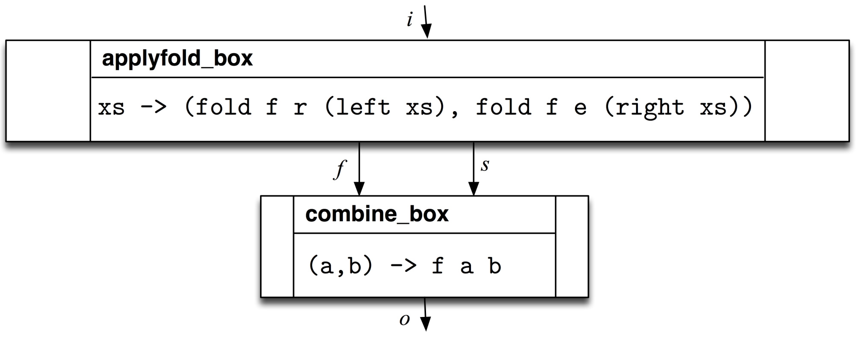
Next, we curry this function, which by the **ESub** rule gives us the match:

```
(a,b) -> (curry(uncurry f)) a b
```

From Theorem 4 we have that:

$$\texttt{curry}(\texttt{uncurry}\ f) = f$$

Using this theorem we apply the **ESub** rule to get the following new configuration:



We have now completed the transformation of `combine_box`. Next we start transforming `applyfold_box`. First we apply **EFoldI** to fold the expression into a new function `h`:

```
h xs  = (fold f r (left xs), fold f e (right xs));
```

and the box match becomes:

```
xs ->  h xs
```

Next, we define a new function

```
  g (a,b) = (fold f r a, fold f e b);
```

and show that

$$\mathtt{h}\ xs = \mathtt{g}\ (\mathtt{left}\ xs, \mathtt{right}\ xs)$$

which hold by unfolding both function definitions. We apply **ESub** to create a new match:

```
  xs -> g (left xs, right xs)
```

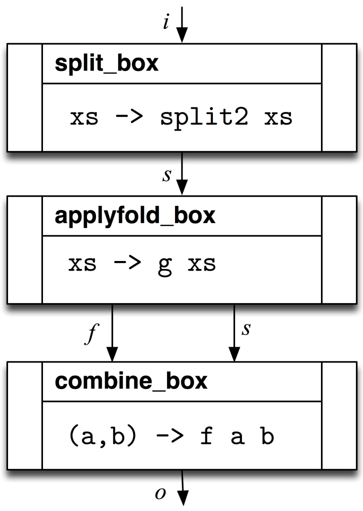
We know that:

$$(\mathtt{left}\ xs, \mathtt{right}\ xs) = \mathtt{split2}\ xs$$

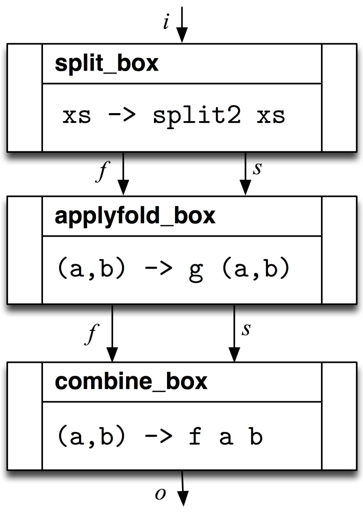
by the definition of `split2`. Using this we apply the **ESub** rule creating the following new match:

```
  xs -> g (split2 xs)
```

Again, this is sequential composition of two function, which we can lift to the box level by rule **VCompE**. By making some suitable renaming we obtain the following box configuration:



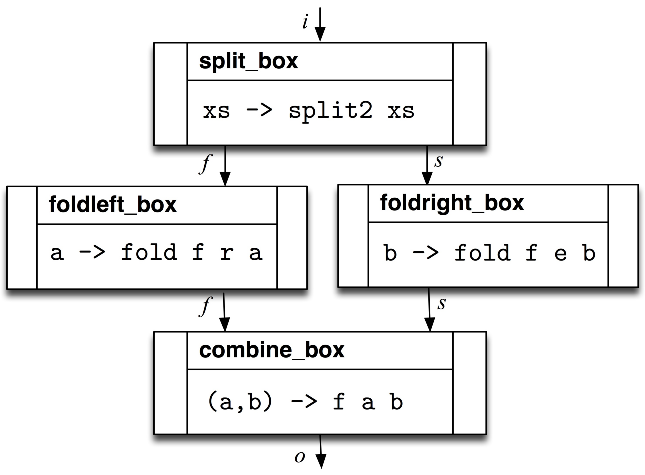
As above, the wire between the two new boxes is a pair which we can split by the **TupleE** rule:



The transformation of the split_box box is now completed. and we start on applyfold_box. First, we unfold the g function by rule **EFoldE** creating the new match:

```
  (a,b) -> (fold f r a, fold f e b)
```

Next we observe that the first element of the expression only uses the first pattern, while the second element only uses the second pattern, hence we can apply the horizontal box decomposition, that is rule **HCompE**, creating the following final box configuration



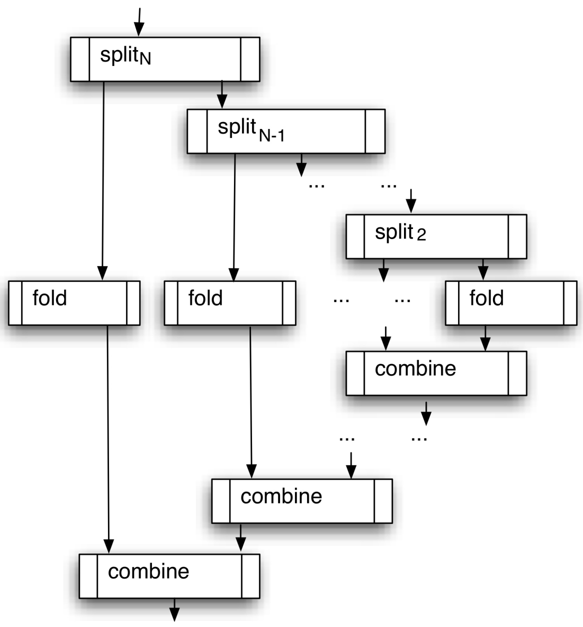
which completes the transformation.

N-box fold We now outline how to generalise this into N cores. We assume that N is fixed for ease of presentation (but we can also abstract over N). First we generalise `left`, `right` and `split2` with an additional argument `n` specifying the size of the chunks:

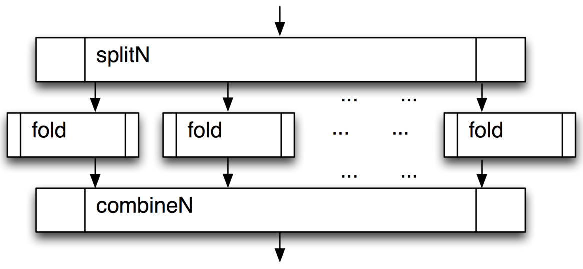
```
left n x = take ((length x) div n) x;
right n x = drop ((length x) div n) x;
splitN n x = (left n x,right n x);
```

Note that the previous definitions would be equivalent to setting `n` to `2`. The idea is that we we want to split a given list `x` into $\frac{\texttt{length x}}{\texttt{N}}$ chunks and apply (a slight adaption of) the transformation from above $N-1$ times, so that each chunk is executed on a core with equal load balancing.

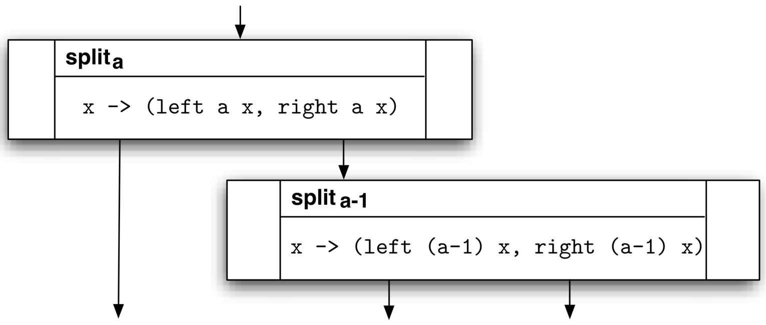
The difference for each transformation is that we use the more general version `splitN`. Firstly, we apply the transformation with N given as argument, i.e. `splitN N`. The first output wire will then contain the first $\frac{1}{N}$ parts of the list and this will be sent to the first core. The remaining $\frac{N-1}{N}$ parts of the list are sent down the second wire. Since we have used one core (the first wire), we have $N-1$ cores left. Thus, we reapply this transformation to the second wire with `splitN (N-1)` to get a chunk equal to to the first wire. This transformation is recursively applied $N-1$ times, which will create an "arrow-headed" shape



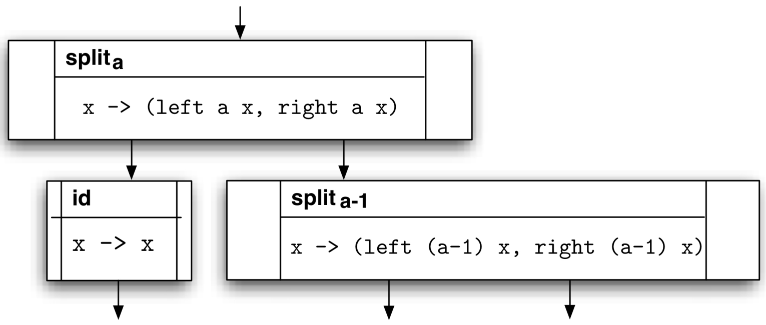
where we need to flatten the "top" and "bottom", creating the following shaped box configuration:



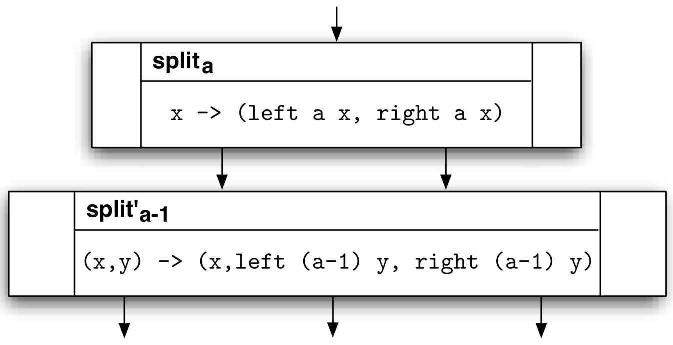
The top can be flatten by a sequence of the transformations discussed next, which shows how two boxes at the a^{th} $(2 < a \leq N)$ step can be combined (after unfolding `splitN`):


split a
x -> (left a x, right a x)
split a-1
x -> (left (a-1) x, right (a-1) x)

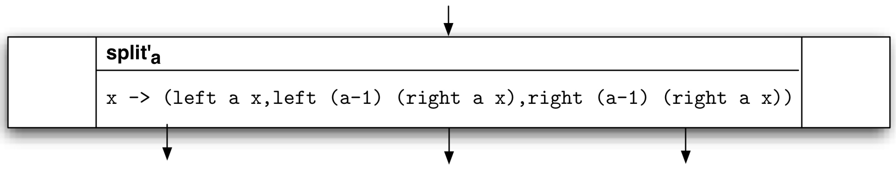
First we apply **IdI** to introduce an identity box on the first output wire:


split a
x -> (left a x, right a x)
id
x -> x
split a-1
x -> (left (a-1) x, right (a-1) x)

Then we horizontally merge this box with the second box by the **HCompI** rule which creates the following configuration:


split$_a$
x -> (left a x, right a x)
split'$_{a-1}$
(x,y) -> (x,left (a-1) y, right (a-1) y)

We can then vertically compose these two boxes with **VCompI**, creating


split'$_a$
x -> (left a x,left (a-1) (right a x),right (a-1) (right a x))

Notice that the result of the transformation is to make a copy of the last element `e` of the tuple in the expression of the box, so `(...,e)` becomes `(...,e,e)`. If we are at level k, then the first `e` is replaced by `left (N-k) e` and the second `e` is replaced by `right (N-k) e` – so the tuple is now `(...,left (N-k) e,right (N-k) e)`.

Reverting to our transformation. If we assume that $a > 3$, the result of incorporating the “split box” at the next level down, is the following match:

```
x -> (left a x,
        left (a-1) (right a x),
          left (a-2) (right (a-1) (right a x)),
            right (a-2) (right (a-1) (right a x)))
```

Note that we start at the top of the “arrow-head” diagram when combining boxes.

When merging the results, the match of each box looks as follows:

```
(x,y) -> f x y
```

Two such boxes can be combined with exactly the same approach of introducing an identity box (**IdI**), followed by horizontal (**HCompI**) and then vertical composition **VCompI**, giving a new box with the match

```
(x1,x2,x3) -> f x1 (f x2 x3)
```

Since we assume that `f` is associative this can be rewritten to:

```
(x1,x2,x3) -> f (f x1 x2) x3
```

Applying the associative rewrite to the next box it becomes

```
(x1,x2,x3,x4) -> f (f (f x1 x2) x3) x4
```

and so on. At the end the box will have the following match schema:

```
(x1,x2,x3,x4,...,xn) -> f (...(f (f x1 x2) x3) x4 ...) xn
```

This is the result of applying the `fold` function!

Theorem 13.

$$f\ x\ y = \texttt{fold}\ f\ x\ [y]$$

Proof. Firstly, remember that $[y]$ is shorthand for $(y : [])$. The proof follows from two unfoldings of the definition of `fold`:

$$\texttt{fold}\ f\ x\ (y : []) \rightarrow \texttt{fold}\ f\ (f\ x\ y)\ [] \rightarrow f\ x\ y$$

Next we prove the following property about `fold`:

Theorem 14.

$$f\ (\texttt{fold}\ f\ x\ ys)\ z = \texttt{fold}\ f\ x\ (ys\texttt{++}[z])$$

Proof. By structural induction on ys:

Base case:	$f\ (\texttt{fold}\ f\ x\ [])\ z = \texttt{fold}\ f\ x\ ([]\texttt{++}[z])$	
	$\texttt{fold}\ f\ x\ ([]\texttt{++}[z])$	$\rightarrow$ (`++`)
	$\texttt{fold}\ f\ x\ [z]$	$\rightarrow$ (`fold` def (twice))
	$f\ x\ z$	$\rightarrow$ (`fold` def)
	$f\ (\texttt{fold}\ f\ x\ [])\ z$	
Recursion case:	$f\ (\texttt{fold}\ f\ x\ (y : ys))\ z =$ $\texttt{fold}\ f\ x\ ((y : ys)\texttt{++}[z])$	
Assumption:	$f\ (\texttt{fold}\ f\ x\ ys)\ z = \texttt{fold}\ f\ x\ (ys\texttt{++}[z])$	[induction hyp.]
	$\texttt{fold}\ f\ x\ ((y : ys)\texttt{++}[z])$	$\rightarrow$ (++)
	$\texttt{fold}\ f\ x\ (y : (ys\texttt{++}[z]))$	$\rightarrow$ (`fold`)
	$\texttt{fold}\ f\ (f\ x\ y)\ (ys\texttt{++}[z])$	$\rightarrow$ (induction hyp.)
	$f\ (\texttt{fold}\ f\ (f\ x\ y)\ ys)\ z$	$\rightarrow$ (`fold`)
	$f\ (\texttt{fold}\ f\ x\ (y : ys))\ z$	

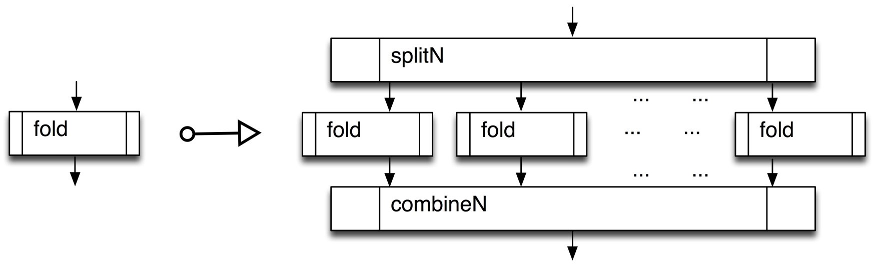
By first applying Theorem 13 to the innermost `f` application, and then reapplying Theorem 14 until we are left with a large `fold` expression we end up with the following property:

```
f (...(f (f x1 x2) x3) x4 ...) xn =  fold f x1 [x2,...,xn]
```

With this we apply **ESub** to get the following match:

```
(x1,x2,x3,x4,...,xn) -> fold f x1 [x2,...,xn]
```

This completes the following transformation:



7 Conclusion

We have surveyed the Hume programming language and shown how the explicit separation of coordination and computation aids formal reasoning about programs. In particular, we have introduced the box calculus for reasoning about coordination and shown how, in conjunction with extant reasoning systems for computation, it is possible to construct robust proofs of substantial program constructs for parallelism through systematic transformation.

We envisage two important avenues for future activity. First of all, effective box calculus deployment clearly requires a graphical tool-set to support, and ultimately automate, scalable program transformation. Secondly, there are excellent opportunities in complementing the box calculus with resource analysis to enable resource directed program development in a “costing by construction” style. Here again, this should be supported by an integrated tool-set which can flexibly account for different resource modalities such as time, space and power consumption.

Acknowledgements

We would like to thank our collaborators Kevin Hammond of the University of St Andrews, Hume’s co-designer, Andrew Ireland of Heriot-Watt University, who helped develop the box calculus, and the anonymous reviewer for very constructive feedback.

Hume resources

The Hume home page is:

`http://www-fp.cs.st-andrews.ac.uk/hume/index.shtml`.

Hume tools and documentation may also be found at:

`http://www.macs.hw.ac.uk/~greg/hume/`.

The Hume Report is at:

```
http://www.macs.hw.ac.uk/~greg/hume/hume11.pdf
```

and the Hume Manual is at:

```
http://www.macs.hw.ac.uk/~greg/hume/manual.pdf
```

References

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):4467, 1977.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming.* Prentice-Hall, 1997.

[Bir87] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[BLOMP97] S. Breitinger, R. Loogen, Y. Ortega-Mallen, and R. Pena. The Eden Coordination Model for Distributed Memory Systems. In *Proc. High-Level Parallel Prog. Models and Supportive Envs. (HIPS)*, number 1123 in LNCS. Springer-Verlag, 1997.

[Bur69] R. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[CJP07] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* MIT, 2007.

[Dij75] E. W. Dijkstra. Guarded commands, non-determinacy and derivation of programs. *Commuications of the ACM*, 18(8):453–457, 1975.

[eAB$^+$99] S.L. Peyton Jones (ed.), L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, K. Hammond, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, J.C. Peterson, A. Reid, and P.L. Wadler. Report on the Non-Strict Functional Language, Haskell (Haskell98). Technical report, Yale University, 1999.

[GM08] G. Grov and G. Michaelson. Towards a Box Calculus for Hierarchical Hume. In M. Morazan, editor, *Trends in Functional Programming Volume 8*, pages 71–88. Intellect, 2008.

[GM11] G. Grov and G. Michaelson. Hume box calculus: robust system development through software transformation. *Higher Order Symbolic Computing*, July 2011. DOI 10.1007/s10990-011-9067-y.

[Gro09] G. Grov. *Reasoning about correctness properties of a coordination language.* PhD thesis, Heriot-Watt University, 2009.

[Hal60] P. R. Halmos. *Naive Set theory.* Van Nostrand, 1960.

[HM03] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *In Proc. of GPCE'03*, pages 37–56. LNCS 2830, Sep. 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *communications of the ACM*, 12:576–583, October 1969.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hod77]	W. Hodges. *Logic*. Pelican, 1977.
[Inm88]	Inmos. *Occam Reference Manual*. Prentice-Hall, 1988.
[Kne63]	G. Kneebone. *Mathematical Logic and the Foundations of Mathematics*. Van Nostrand, 1963.
[McC62]	J. McCarthy. A basis for a mathematical theory of computation. Technical Report Memo 31, MIT, 1962.
[MF94]	MPI-Forum. MPI: A message passing intrface standard. *International Journal of Supercomputer Application*, 8(3–4):165–414, 1994.
[Mil82]	R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982.
[Mil99]	R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.
[MTHM97]	Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
[Nid62]	P. H. Nidditch. *Propositional Calculus*. Routledge and Kegan Paul, 1962.
[Pet67]	R. Peter. *Recursive Functions*. Academic Press, 1967.
[Qui64]	W.V. Quine. *Word and Object*. MIT, 1964.
[Weg68]	P. Wegner. *Programming Languages, Information Structures, and Machine Organization*. McGraw-Hill, 1968.