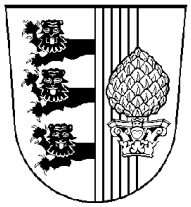
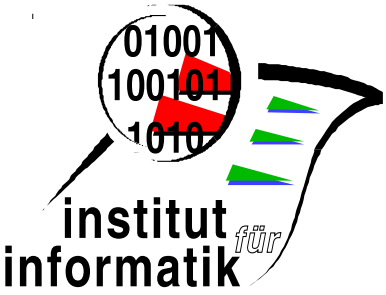
UNIVERSITÄT AUGSBURG



An Algebraic Approach to Systolic Circuits

Bernhard Möller

Report 1998-01 Januar 1998



INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

An Algebraic Approach to Systolic Circuits[1]

Bernhard Möller

Institut für Informatik
Universität Augsburg

Abstract. The goal of deductive design is the systematic construction of a system implementation starting from its behavioural specification according to formal, provably correct rules. We use Gofer/Haskell to formulate a functional model of directional, synchronous and deterministic systems with discrete time. We develop an algebra of slowdown. The associated laws are then employed in deductive hardware design of two systolic circuits, a convolver and a recognizer for regular expressions. The strategy used is applicable to the derivation of other systolic circuits as well.

1. Deductive Design

The goal of deductive design is the systematic construction of a system implementation
- starting from its behavioural specification,
- according to formal, provably correct rules.

The main advantages are the following.
- The resulting implementation is correct by construction;
- The rules can be formulated schematically, independent of the particular application area;
- Hence they are re-usable for wide classes of similar problems;
- Being formal, the design process can be assisted by machine.
- Implementations can be constructed in a modular way.
  - The first emphasis lies on correctness;
  - Subsequently transformations can be used to increase the performance.
- A formal derivation serves as a record of the design decisions that went into the construction of the implementation.
  - It is an explanatory documentation and
  - eases revision of the implementation upon modification of the system specification.

Note that we do not view deductive design as alternative to, but complementary to verification.

[1] This research was partially sponsored by Esprit Working Group 8533 *NADA – New Hardware Design Methods.*

There is a variety of approaches to deductive design, e.g.,

- refinement calculus,
- program extraction from proofs,
- transformations.

We shall follow the latter (see e.g. Bauer et al. 89, Partsch 90) and use mainly

- equational reasoning,
- algebraic laws,
- structural induction,
- fixpoint induction for recursive definitions.

In this paper we present the deductive design of two systolic circuits, a convolver and a recognizer for regular expressions. The approach uses an algebra of functions based on the semantics of the lazy functional language *Gofer*. In Möller 1998 that approach is extended to a general framework for deductive hardware design in the particular area of

- directional,
- synchronous and
- deterministic systems with
- discrete time.

The approach generalises with varying degrees of complexity to adirectional systems, asynchrony, non-determinacy or continuous time.

2. *Gofer* as a Hardware Description Language

We model hardware functionally in *Gofer/Haskell*. The reasons for this are the following.

- Functional languages supports various views of streams directly.
- Polymorphism allows generic formulations and hence supports re-use.
- Since all specifications are executable, direct prototyping is possible.
  An adaptation of the transformation system CIP-S (see Bauer et al. 87) for *Gofer/Haskell* is being constructed at the University of Ulm under H. Partsch. This will allow direct replay of the paper and pencil derivations done here to check their correctness by machine. Moreover, the set of transformation rules given here can then be re-used for further derivations directly on the system.
- Functional languages are being considered for their suitability as bases of modern hardware description languages; an example is the (unforunately abandoned) language *MHDL* (see Rhodes 95).
- Many approaches to hardware specification and verification also use higher-order concepts to good advantage (see e.g. Gordon 86).

2.1 Basic Types and Functions

For those not familiar with *Gofer*, we briefly repeat the essential elements of *Gofer*.
Basic types are Int for the integers and Bool for the Booleans with elements True and False . The type of functions taking elements of type a as arguments and producing elements of type b as results is a -> b . The fact that a function f has this type is expressed as f :: a -> b .
Function application is denoted by juxtaposing function and argument, separated by at least one blank, in the form f x . Functions of several arguments are mostly used in curried form f x_1 x2 ... x_n . In this case f has the higher-order type f :: t_1 -> (t_2 -> ... (t_n -> t) ...) or, abbreviated, f :: t_1 -> t_2 -> ... t_n -> t (the arrow -> associates to the right, whereas function application associates to the left).
Functions are defined by equations of the form f x = E or as (anonymous) lambda abstractions. Instead of λx.E one uses the notation \x -> E .
A two-place function f :: a -> b -> c may also be used as an infix operator in the form x `f` y ; this is equivalent to the usual application f x y .
Consider now some binary operator # . By supplying only one of its arguments we obtain a *residual function* or *section* of the form (x #) = \y -> x # y or (# y) = \x -> x # y .

2.2 Case Distinction and Assertions

Gofer offers several possibilities for doing case distinctions. One is the usual if-then-else construct. To avoid cascades of ifs, a function may also be defined in a style similar to the one used in mathematics. The notation is

```
f x
  | C1        =  E1
  ...
  | Cn        =  En
```

The result is the value of the first expression E_i for which the corresponding C_i evaluates to True . If there is none, the result is undefined.
We shall also use this to make functions intentionally partial in order to enforce assertions about their parameters (see Möller 96).
If one wants to avoid partiality one can use the predefined constant otherwise = True and add a final clause

```
  | otherwise  =  En+1 .
```

Yet another way of case distinction is provided by defining a function through *argument patterns*. Several equations indicate what a function does on inputs that have certain shapes. The equations are tried in textual order; if no pattern matches the current argument, the function is again undefined at that point.

Example: By the equations
f 0 = 5
f 1 = 7
function f :: Int -> Int is defined only for argument values 0 and 1 .

2.3 Lists

The type of lists of elements of type a is denoted by [a] . The list consisting of elements $x_1,...,x_n$ is written as $[x_1,...,x_n]$; in particular, [] is the empty list. Concatenation is denoted by ++ . The function length returns the length of a list.
A very useful specification feature is list comprehension in the form

[f x | x <- L, px]

where L is a list expression, f some function on the list elements and p a boolean function. The symbol <- may be viewed as a leftward arrow and pronounced as "drawn from" or as a form of element sign. In this latter view, the expression is the list analogue of the usual set comprehension { f x | x ∈ S, p x}. The meaning of the list comprehension expression is again a list, constructed as follows:

- The elements of list L are scanned in left-to-right order.
- On each such element x the test p is performed.
- If p x = True , f x is put into the result list.
- Otherwise, x is ignored.

The list [m, m+1, ... , n] of integers may be denoted by the shorthand [m..n] . The right bound n may be omitted; then the expression denotes the infinite list [m, m+1, ...] .

3. A Model of Sequential Circuits

3.1. Streams and Stream Transformers

A frequently used model of sequential hardware is that of *stream transformers*. Streams are used to model the temporal succession of values on the connection wires, whereas the modules are functions from (bundles of) input streams to (bundles of) output streams.
In this paper we deal with discrete time only. Even this leaves several options how to represent streams. One possibility would be to define

type Stream a = [a]

Since *Gofer/Haskell* employs a lazy semantics, this allows finite as well as infinite streams. Time remains implicit, but can be introduced using the list indexing operation (!!).
We use a version which explicitly refers to time:

```
type Time = Int
type Stream a = Time -> a
```

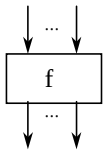
This will carry over easily to real time. On the other hand, this does not directly support finite streams. They have to be modeled by functions that become eventually constant, preferably yielding only bot after the “proper” finite part.
We will use bot also to “cut off” negative time points. To this end we define

```
nonneg :: (Time -> a) -> Stream a
nonneg f t
| t >= 0    =   f t
```

So nonneg f is a stream that is undefined for negative time points (i.e., enforces the assertion t >= 0) and on nonnegative time points agrees with f .

3.2 Functions as Modules

A combinational module will be modelled as a function taking a list of inputs to a list of outputs. Diagrammatically we represent such a module f as



Using lists of inputs and outputs has the advantage that the basic connection operators can be defined independent of the arities of the functions involved. The disadvantage is that we need uniform typing for all inputs/outputs. Conventional polymorphism is too weak here; one would need an extension to "tuples as first-class citizens" with concatenation of tuple types and also of tuples as primitives.
We now discuss briefly the role of functions as modules of a system. In a higher-order language such as *Gofer* there are two views of functions:

- as routines with a body expression that depends on the formal parameters, as in conventional languages;
- as "black boxes" which can be freely manipulated by higher-order functions (combinators).

The latter view is particularly adequate for functional hardware descriptions, since it allows the direct definition of various composition operations for hardware modules.
However, contrary to other approaches we do not reason purely at the combinator level, i.e. without referring to individual in/output values. While this has often advantages, it can become quite tedious in other places. So we prefer to have the possibility to switch.
The basis for reasoning about functions is the *extensionality rule*

f = g iff f x = g x for all x .

To show equality of two functional expressions F and G we may hence

- start with the expression F x ;
- *unfold* F , i.e., push the argument x through F till calls h x of usual functions h result;
- substitute x for the formal parameters of these functions;
- manipulate the resulting expression till it has the form G x .

Then the extensionality rule tells us F=G .
Many algebraic laws we use are equalities between functions, interpreted as extensional equalities.

Example: Function composition is defined in *Gofer* by

(f . g) x = f (g x)

with polymorphic combinator

(.) :: (b -> c) -> (a -> b) -> a -> c

A fundamental law is associativity of composition:

(f . g) . h = f . (g . h)

3.3 About Connections

We shall employ two views of connections between modules:

- that of "rubber wires", represented by formal parameters or implicitly by plugging in subexpressions as operands;
- that of "rigid wires", represented by special routing functions which are inserted using basic composition combinators.

We proceed in two stages:

- We start at the level of rubber wiring to get a first correct implementation.
- Then we (mechanically) get rid of formal parameters by *combinator abstraction* to obtain a version with rigid wiring.

Note, however, that many approaches *start* at the latter level and have to carry the complications of wiring all through the derivation. This is tedious and obscures the essential steps.

In drawing diagrams we shall be liberal and use views in between rubber and rigid wiring. In particular, we shall use various directions for the input and output arrows.

We now sketch how to step from the logical connection to a topology with rigid wires, crossings and fan-out.

3.4 Basic Wiring Elements, Sequential and Parallel Composition

The basic wiring elements are a straight wire, modeled by the identity function, the fan-out of degree 2 (fork), the crossing (swap) and the sink:

```
id [x] = [x]      fork [x] = [x,x]      swap [x,y] = [y,x]      sink [x] = []
```

These operations are extended to wire bundles:

```
bfork m n xs
  | length xs == n  =  foldr (++) [] (copy m xs)
  ■  undefined otherwise

bswap m n xs
  | length xs == n = drop m xs ++ take m xs

bsink n xs = []
```

The identity id is predefined polymorphically by id y = y and hence doesn't need to be extended to wire bundles.
Based on this we define selection nets:

```
sel n i j =                    -- for i `below` n && j `below` n
  par i (bsink i (par j (bid j) (bsink (n-j))))
```

We have the following fusion rule:

```
bfork 2 |> par (j-i) (sel n i j) (sel n j k) = sel n i k
```

Finally, we have the *invisible module* ide with 0 inputs and 0 outputs:

```
ide [] = []
```

Sequential composition simply is reverse function composition. We are a bit sloppy here about the arities of the functions; this has again to do with the already mentioned absence of tuples as fisrt-class citizens. For parallel composition we need to tell the operator how many inputs are to be distributed to the first function; the remaining ones go to the second function.

```
(f  |> g ) xs    =   g  (f xs)
par k f g xs    =  f  (take k xs) ++ g (drop k xs )
```

We abbreviate par 1 by the infix operator ||| .

These operations enjoy a number of natural properties, such as associativity

f |> (g |> h) = (f |> g) |> h ,

par (m+k) (par m f g) h = par m f (par k g h) ,

neutrality

id |> f = f = f |> id ,

par m f ide = f = par 0 ide f ,

idempotence

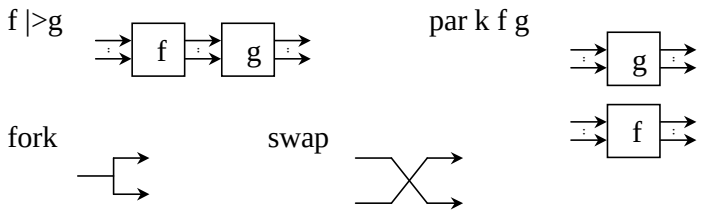
swap |> swap = id

and the abiding law

par m (f |> g) (h |> k) = (par m f h) |> (par n g k) .

These laws should hold for all semantic models of graph-like networks; they reflect the abstraction that lies in the graph view. A systematic account of these properties has been given in Ştef→nescu 94. Whereas associativity and abiding just allow "parenthesis-free layouts", use of neutrality or idempotence means simplification/complexification of abstract layouts.

Frequently we will use pictorial representations of our operations:



The input/output streams are numbered from bottom to top in the respective lists.

3.5 Combinator Abstraction

We have already discussed the need to pass from rubber wiring to rigid wiring. Formally this is achieved by eliminating all formal parameters from functional expressions in favour of parallel and sequential composition and the basic wiring elements. The resulting expression is called the *combinator abstraction* CA E of the original expression E .
For its construction, we need the list ID E of the formal parameters occurring in expression E . This list is organized in textual order of appearance of the parameters and kept repetition free.
The abstraction rules for expressions with formal parameters in list $[x_0,...,x_{n-1}]$ are as follows:

- CA $[x_i]$ = sel n i (i+1)
- CA f = f where f = \xs -> [f (xs!!0) ... (xs!!(k-1))] if f :: t_0 -> ... t_{k-1} -> t
- CA (f E1 ... En) = (CA E1 ||| ... ||| CA En) |> CA f
- CA (E1 ++ ... ++ En) = bfork n |> (CA E1 ||| ... ||| CA En)

These basic rules may lead to circuits involving very high fan-outs. More refined rules avoid this (see Möller 1998). For further details on wiring we refer to Hotz et al. 86 and Molitor 91.

4. Lifting and Constant

We introduce *liftings* of operations on data to streams. A “unary” operation takes a singleton list of input data and produces a singleton list of output data. This is lifted to a function from a singleton list of input streams to a singleton list of output streams. It is the analogue of the apply-to-all operation map on lists. Since streams are functions themselves, the lifting may also be expressed using function composition. We have

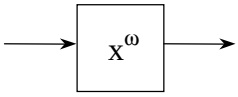
```
lift1 :: (a -> b) -> [Stream a] -> [Stream b]
lift1 f [d] = [\ t -> f (d t)] = [f . d]
```

Similarly, we have for binary operations

```
lift2 :: (a -> a -> b) -> [Stream a] -> [Stream b]
lift2 g [d,e] = [\t -> g (d t) (e t)]
```

Another useful building block is a module that emits a constant output stream. For convenience we endow it with a (useless) input stream. So this module actually is a combination of a sink and a source. We define

```
cnst :: a -> [Stream b] -> [Stream a]
cnst x  =  lift1 (const x)
```

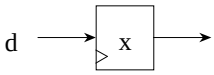


Here const is a predefined *Gofer* function that produces a constant unary function from a value.

5. Initialised Unit Delay

To model memory of the simplest kind we use a unit delay module. Other delays such as inertial delay or transport delay can be modeled similarly. For a value x the stream transformer (x &) shifts its input stram by one time unit; at time 0 it emits the initial value x :

```
(&) :: a -> [Stream a] -> [Stream a]
(x & [d])  =  [nonneg e]  where  e t   | t == 0  =  x
                                       | t  >  0  =  d (t-1)
```

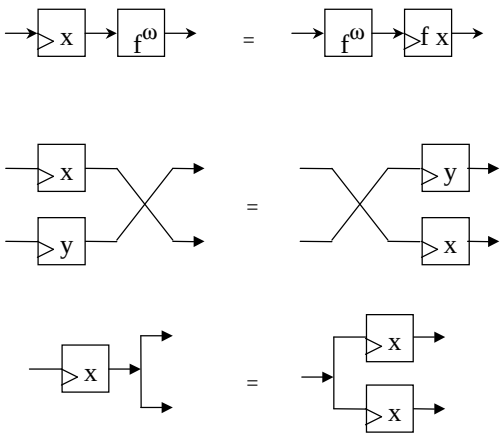


To push delays through larger networks we have the following

Lemma (Delay Propagation Rules):

- (x&) |> lift1 f = lift1 f |> ((f x) &)
  provided f is strict, i.e., is undefined whenever its argument is
- ((x&) ||| (y&)) |> lift2 g = lift2 g |> ((g x y)&)
  provided g is doubly strict, i.e., is undefined whenever *both* its argument are
- (x&) |> cnst y = cnst y |> (y&)
- ((x&) ||| (y&)) |> swap = swap |> ((y&) ||| (x&))
- (x&) |> fork = fork |> ((x&) ||| (x&))

These rules can be given in pictorial form as

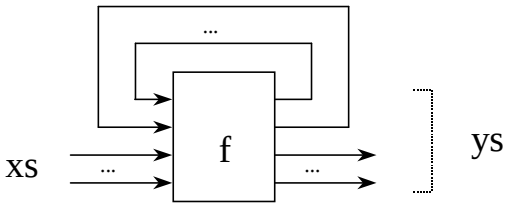


For propagation through |> and ||| we may use associativity of |> and the abiding law. These simple laws are quite effective as will be seen in later examples.

6. Feedback

Another essential ingredient of systems with memory is *feedback* of some outputs to inputs. We use
feed :: Int -> ([a] -> [a]) -> ([a] -> [a])
where the first parameter indicates how many outputs are fed back. The definition reads

```
feed k f xs  =  codrop k ys
  where  ys =  f (xs ++ cotake k ys)
         cotake n xs =  drop (length xs – n) xs
         codrop n xs  = take (length xs – n) xs
```

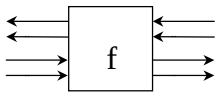


Note the recursive definition of ys which reflects the flowing back of information. This recursion is well-defined by the lazy semantics of *Gofer*.

The feedback operation enjoys a number of algebraic laws which show that it models the rubber wire abstraction correctly. For a systematic exposition see again ⓪tef→nescu 94.

7. Interconnection (Mutual Feedback)

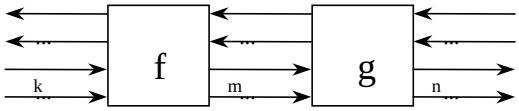
In more complex designs it may be convenient to picture a module f with inputs and outputs distributed to both sides:



We want to compose two such functions to model interconnection of the respective modules. To this end we introduce

```
connect :: Int -> Int -> Int -> [Stream a ] -> [Stream a ]
```

The three Int-parameters in connect k m n f g are used similarly as for splicing: they indicate that k inputs are supposed to come from the left neighbour of f , that m wires lead from f to g , and that n outputs go to the right neighbour of g .



We define therefore

```
connect k m n f g xs  =  take n zs ++ drop m ys
          where    ys  =  f (take k xs ++ drop n zs)
                   zs  =  g (take m ys ++ drop k xs)
```

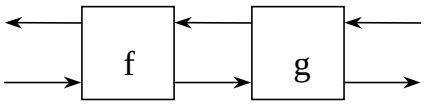
This involves a mutually recursive definition of ys and zs which again is well-defined by the lazy *Gofer* semantics.

Lemma: Interconnection is associative:

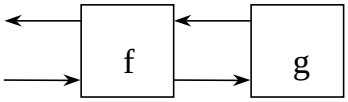
connect m n p (connect k m n f g) h = connect k m n f (connect m n p g h)

Also, connect has the identity id as its neutral element. Two interesting special cases are

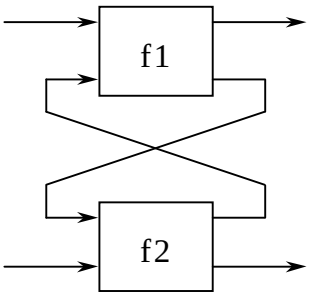
- f =||= g = connect 1 1 1 f g



- f = | g = connect 1 1 0 f g

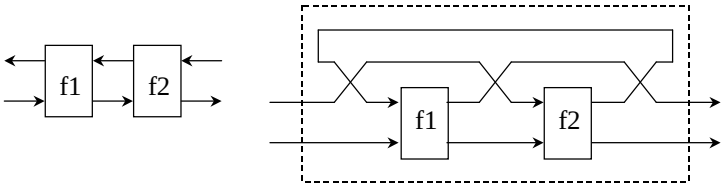


The operator =||= is also known as *mutual feedback* $\otimes$. The corresponding network can be depicted as



Using a suitable torsion of the network we can relate interconnection to feedback:

```
f1 =||= f2        =     feed 1 ( (id ||| swap)  |>  (f1 ||| id)  |>  (id ||| swap)  |>
                                 (f2 ||| id)  |>  (id ||| swap) )
```



8. A Convolver

We tackle our first example now. A *non-programmable convolver* of degree n uses n fixed weights to compute at each time point t >= n the convolution of its previous n inputs by these weights. For convenience we collect the weights also into a stream w .

8.1 Specification

The convolver is specified by

```
conv :: Stream Int -> Int -> [Stream Int] -> [Stream Int]
conv w n d  = [e]
```

```
where e      = \t ->  if  t < n  then bot
                                 else  sum [ w (n-i) * d (t-i) | i <- [1..n] ] ]
                  bot  =  bot    -- undefined element
```

It should be clear that the problem generalises to arbitrary compositions of fold and apply-to-all operations.

8.2 Derivation of a Convolver Circuit

Our first goal is now to derive an inductive (recursive) version of conv which does no longer refer to sum and uses only operations on single data elements. To achieve this we use the classical *unfold/fold strategy* (see e.g. Partsch 90).

For t >= 0 and [e] = conv w 0 d we calculate

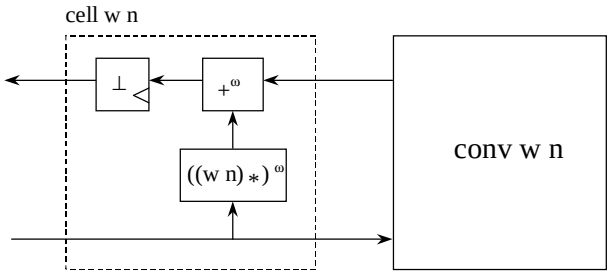
```
e t
=        sum [ w (0-i) * d (t-i) | i <- [1..0] ]
=        sum [ w (0-i) * d (t-i) | i <- [] ]
=        sum []
=        0
Hence   conv 0  =  cnst 0 .
```

For t >= n+1 and [e] = conv w (n+1) d we obtain

```
          e t
=         sum [ w (n+1-i) * d (t-i) | i <- [1..n+1] ]
=         w n * d (t-1) + sum [ w (n+1-i) * d (t-i) | i <- [2..n+1] ]
=         w n * d (t-1) + sum [ w (n+1-(j+1)) * d (t-(j+1)) | j <- [1..n] ]
=         w n * d (t-1) + sum [ w (n-j) * d (t-1-j) | j <- [1..n] ]
=         w n * d (t-1) + c (t-1)
where  [c] = conv w n d .
```

Now combinator abstraction yields

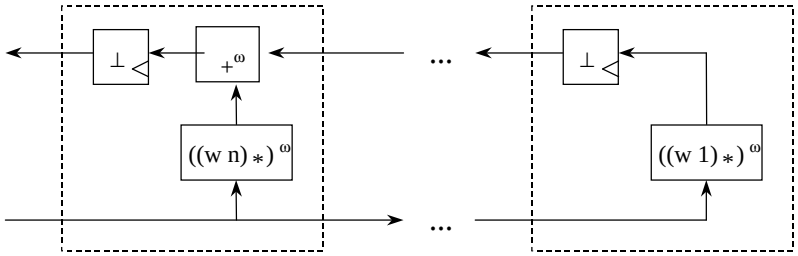
```
conv w (n+1)   =   (cell w n)  =|  (conv w n)
cell w n [li,ri]  =    (bot  &  lift2 (+)  (lift1 (w n *) [li], [ri]),  [li])
```



Unwinding the recursion for fixed n we obtain a regular design:

conv w n = (foldr1 (=||=) [cell w k | k <- [1..n]) =| cnst 0

After simplification of the rightmost cell this yields



However, we have a long combinational broadcasting path (fanout n) at the bottom

8. 3 Towards a Systolic Version

A circuit is *combinational* if it uses only lifted operations and sequential or parallel composition. In clocked systems, the clock period is determined by the longest combinational path.
A circuit is *systolic* if it is built - using sequential and parallel composition and feedback - out of small combinational modules which are separated by delay elements. A systolic circuit has the advantage that the clock period can be kept relatively short.
We want to obtain a systolic version of our convolver. Hence we have to introduce additional delay elements.

9. Slowdown

The technique to achieve this is *slowdown* (see e.g. Leiserson, Saxe 83, Jones, Sheeran 90). The k-fold slowed down version of a circuit works on k interleaved streams. So each of these is processed at rate k slower than in the original circuit.

9.1 Interleaved Streams

To talk about the component streams of such a "multistream" we introduce

```
split k j d t  =  d (k*t + j) .
```

So split k j d is the j-th of the k component streams where numbering starts again with 0 E.g. split 2 0 d and split 2 1 d consist of the values in d at even and odd time points, respectively. Then d can be considered as an alternating interleaving of these.

The following properties of split are useful for proving the slowdown propagation rules below:

Lemma 9.1: (x&) |> split k 0 = (split k (k-1)) |> (x&)
(x&) |> split k j = split k (j-1) (0 < j < n)

To interleave k streams from a list we use

```
ileave k ss t  =  (ss !! (t `mod` k))(t `div` k)
```

We have, provided length ss >= k ,

```
split k j (ileave k ss)  =  ss!!j .
```

A special case is the interleaving of k copies of the same stream:

```
rep k d =  ileave k (copy k d) .
```

The above property yields

```
split k j (rep k d)  =  d .
```

9.2 The Slowdown Function

Now the slowdown function is specified implicitly by

```
split k j (slow k f s) = f (split k j s) .
```

Here f is an arbitrary function on streams, not just a lifted unary operation. In particular, f may look at all the history of a stream. By this definition, slow k f s may be considered

as splitting s into k substreams, processing these individually with f and interleaving the result streams back into one stream. From the specification the following proof principle is evident:

Lemma 9.2: If for a function h and all j in [1..k] we have
h |> split k j = (split k j) |> f
then h = slow k f .

For easier manipulation we want to obtain an explicit version of slow .

Since by definition of split
split k j (slow k f s) t' = slow k f s (k*t' + j)
we have conversely
slow k f s t
= slow k f s (k*(t`div`k) + t`mod`k)
= split k (t`mod`k) (slow k f s) (t`div`k)
= f (split k (t`mod`k) s) (t`div`k) .

In sum:
slow k f s t = f (split k (t `mod` k) s) (t `div` k) .

9.3 Propagation Laws for Slowdown

The function slow distributes nicely through our circuit building operators:

- slow k (x &) = foldr (|>) id (copy k (x &))
- slow k (cnst x) = cnst x
- slow k (f |> g) = slow k f |> slow k g
- slow k (f ||| g) = slow k f ||| slow k g
- slow k (feed m f) = feed m (slow k f)
- slow k (f =||= g) = slow k f =||= slow k g
- slow k (f =| g) = slow k f =| slow k g

This means that the k-fold slowed down version of a circuit results by replacing each delay element by k ones. A further useful propagation law for slow is given by

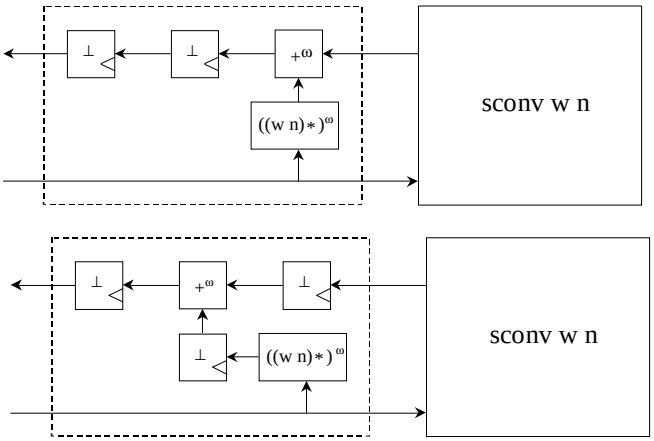
Lemma 9.3: Suppose that (x&) |> f = f |> (y&) . Then also
(x&) |> slow k f = (slow k f) |> (y&) .

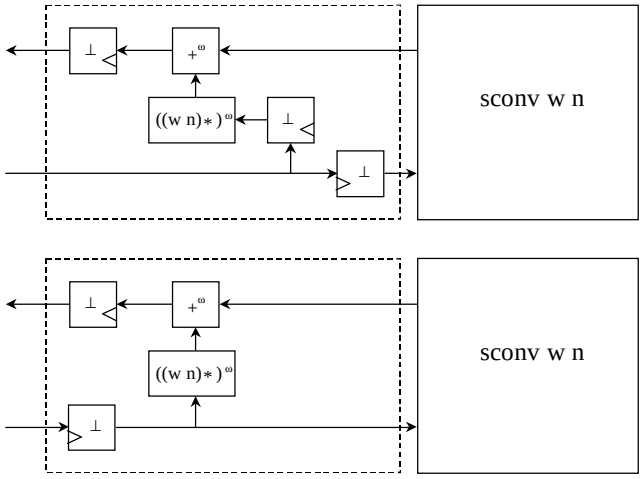
10. A Systolic Convolver: The 2-Slow Convolver

Using k-fold slowdown we can interleave k computations or pad streams with dummy elements by merging the stream proper with a constant stream of dummies. The latter approach isusually taken in verification approaches to the systolic convolver: only the stream values at odd time points are of interest; at even time points the value 0 is used.
We want to *derive* a systolic convolver. We leave the decision whether to use proper interleaving or padding open; both can be achieved by suitable embeddings of the original conv function into the slowed down one defined by

sconv n = slow 2 (conv n) .

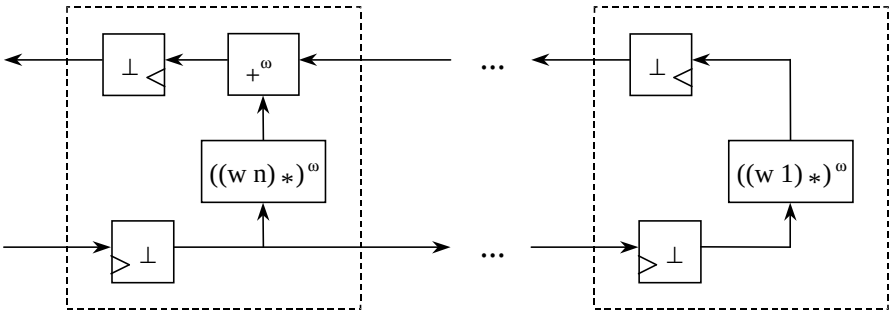
Now we push the second delay introduced by the slowdown through the various modules. We perform the derivation pictorially:





The step of pushing the delay through sconv w n is justified Lemma 9.3. Unwinding the recursion again we obtain a regular systolic design:

```
sconv w n  = (foldr1 (=||=) [scell w k  |  k <- [1..n]]) =| cnst 0
scell w k  [li,ri]  =  [bot & lift2 (+) (lift1 (w k *)[bli], [ri]), bli]
                       where bli  =  bot & li
```



11. A Systolic Recogniser for Regular Expressions

Our next example is a recogniser for regular expressions. The treatment was inspired by Backhouse, Vaccari 97.

11.1 Regular Expressions

We first give a grammar for regular expressions in the form of a recursive *Gofer* data type:

```
data RE a  =  Eps | Term a | RE a :+ RE a |
     RE a :. RE a | Star (RE a)
```

This is polymorphic in the type a of "terminal symbols". Strings over these symbols are modeled by lists of type [a] . Using this we next define the corresponding derivation relation

```
(|-) :: (Eq a) => (RE a) -> [a] -> Bool
```

The intention is that ex |- s yields True iff string s is in the language generated by the regular expression ex . The relation is defined by induction over the cases:

- Eps |- s = length s == 0
- Term x |- s = length s == 1 && head s == x
- (ex1 :+ ex2) |- s = (ex1 |- s) || (ex2 |- s)
- (ex1 :. ex2) |- s = or [(ex1 |- s1) && (ex2 |- s2) | (s1,s2) <- splits s]
- Star ex |- s = length s == 0 || (ex :. Star ex |- s)

The auxiliary function splits computes all possible splits of a list:

```
splits :: [a] -> [([a],[a])]
splits s = [(take n s, drop n s) | n <- [0..length s]]
```

For later use we recall *Arden's rule*: If ex does not generate the empty string, i.e., if we have not(ex |- []) , then from

```
X = Eps  :+  ex :. X
```

we may conclude

```
X = Star ex .
```

11.2 Sections (Observation Windows)

Many properties of streams are conveniently specified by assertions about contiguous stream parts. So we use

```
section :: Stream a -> Time -> Time -> [a]
section d t1 t2  =  [ d t | t <- [t1..t2] ]
```

Example: section (\t -> t^2) 2 5 = [4, 9, 16, 25]

We have the following

Laws:

- length section d t1 t2 = max (t2-t1+1) 0
- take k (section d t1 t2) = if k <= t2-t1+1 then section d t1 (t1+k-1)
- else section d t1 t2
- drop k (section d t1 t2) = if k <= t2-t1+1 then section (t1+k) t2 else []

12.3 Specification of the Recogniser

The recogniser for a regular expression E receives a stream of tokens and a stream of enable bits. It emits True every time it has seen a complete instance of E which started at some earlier time when the enable stream was True . This is formalised by

```
rec :: Eq a => RE a -> Stream a -> Stream Bool -> Stream Bool
rec exp d e t  = if t < 0 then bot
                         else or [ e j && exp |- section d j (t-1) | j <- [0..t]]
```

Although this specification, like all *Gofer* programs, is executable, it is very inefficient. Moreover, it is not in a form that can be directly implemented as a circuit.

12.3 Transformation

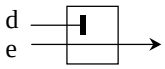
We can, for each variant of the grammar, transform the recogniser into more efficient form. As a sample we show the derivation for the case of a terminal symbol:

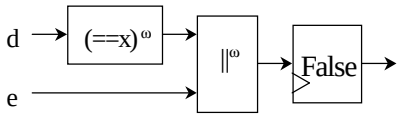
```
rec (Term x) d e t
=  or [ e j && Term x |- section d j (t-1) | j <- [0..t] ]
=  or [ e j && length s == 1 && head s == x
        where s = section d j (t-1)  |  j <- [0..t] ]
=  or [ e j &&  t-1-j+1 == 1 && head s == x
        where s = section d j (t-1)  |  j <- [0..t] ]
=  or [ e j &&  j == t-1 && head s == x
        where s = section d j (t-1)  |  j <- [0..t] ]
=   t > 0 && e (t-1) && head s == x
    where s = section d (t-1)(t-1)
=   t > 0 && e (t-1) && d (t-1) == x
=   ((lift1 (==x) ||| id)  |>  lift2 (&&)  |>  (False&))  d e t
```

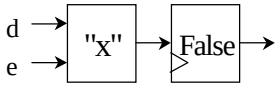
The result of these transformations is given below:

- rec Eps = sel 2 1 2 -- just the enable bit

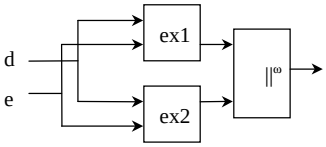
- rec (Term x) = (lift1 (==x) ||| id) |> lift2 (&&) |> (False&)
  As an abbreviation we write "x" = (lift1 (==x) ||| id) |> lift2 (&&) .



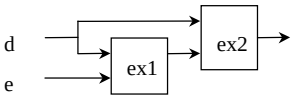




- rec (ex1:+ ex2) =
  bfork 2 |> (par2 (rec ex1) (rec ex 2)) |> lift2 (||)

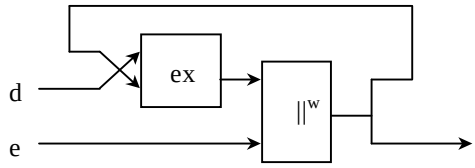


- rec (ex1 :. ex2) = (fork ||| id) |> (id ||| rec ex1) |> rec ex2



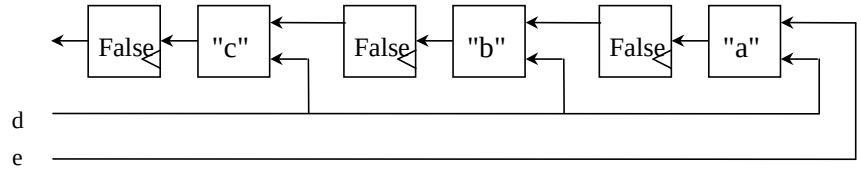
Provided that not (ex |- []) we obtain, using Arden's rule

rec (Star ex) = feed 1 ((par 2 swap id) |> (par 2 (rec ex)id)| |>
lift2 (||) |> fork)



12.4 A Systolic Version

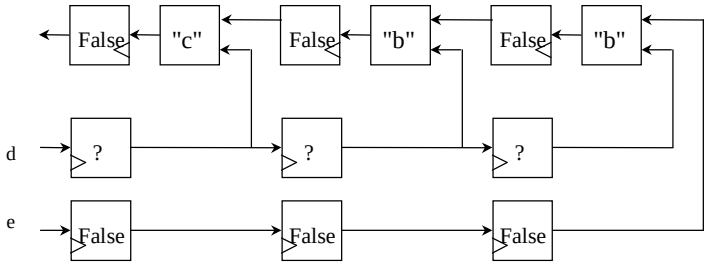
The resulting design again gives rise to long combinational paths with broadcast. E.g. for rec (Term a :. Term b :. Term c) we obtain



So let us look for a systolic version. The strategy is as before:

- apply slowdown (by 2);
- shift the additional delays through the basic modules.

The result for our previous example is the following, where "?" stands for an arbitrary value:



13. Summary

We have seen a number of essential ingredients of deductive hardware design:

- algebraic reasoning,
- parameterisation,
- modularization,
- re-use of designs and derivations,
- precise determination of initialisation values.

The algebraic approach has eased the derivations considerably. In particular, we did not need to introduce anti-delays as e.g. in Vaccari/Backhouse 1997.

Further elaboration of this approach will mainly concern design in the large, asynchronous systems and other notions of time.

Acknowledgement: Many helpful remarks on this paper were provided by G. ☎tef→nescu.

References

F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner,P. Pepper: The Munich project CIP. Volume II: The program transformation system CIP-S. LNCS 292. Springer 1987

F.L. Bauer, B. Möller, H. Partsch, P. Pepper: Formal program construction by transformations - Computer-aided, Intuition-guided Programming. IEEE Transactions on Software Engineering 15, 165-180 (1989)

M.J. Gordon: Why higher-order logic is a good formalism for specifying and verifying hardware. In: G.J. Milne, P.A. Subrahmanyam (eds.): Formal aspects of VLSI design. North-Holland 1986

G. Hotz, B. Becker, R. Kolla, P. Molitor: Ein logisch-topologischer Kalkül zur Konstruktion integrierter Schaltungen. Informatik - Forschung und Entwicklung 1, 28-47 and 72-82 (1986)

G. Jones, M. Sheeran: Circuit design in Ruby. In: J. Staunstrup (ed.): Formal methods for VLSI design. Elsevier 1990, 13—70

B. Möller: Assertions and recursions. In: G. Dowek, J. Heering, K. Meinke, B. Möller (eds.): Higher order algebra, logic and term rewriting. Second International Workshop, Paderborn, Sept. 21-22, 1995. LNCS 1074. Springer 1996, 163-184

B. Möller: Deductive hardware design: a functional approach. Institut für Informatik, Universität Augsburg, Report 1997-09, December 1997. To appear in: B. Möller, J.V. Tucker (eds.): Prospects of Hardware Foundations. Springer LNCS

P. Molitor: A survey on wiring. J. Inf. Process. Cybern. EIK 27, 3-19 (1991)

H.A. Partsch: Specification and transformation of programs - A formal approach to software development. Berlin: Springer 1990

D.L. Rhodes: Analog modeling using MHDL. In: J.-M. Bergé (ed.): Current issues in electronic modeling, Issue #2 “Modeling in analog design. Kluwer 1995

G. ⓪tef→nescu: Algebra of flownomials. Institut für Informatik, Technical Unicersity Munich, Report TUM-I9437, 1994

M. Vaccari, R. Backhouse: Deriving a systolic regular language recognizer. In: R. Bird, L. Meertens (eds.): Algorithmic languages and calculi. Proc. IFIP TC2/WG2.1 Working Conference, Le Bischenberg, Feb. 1997. Chapman&Hall 1997, 49-72