UNIVERSITÄT AUGSBURG

Deductive Hardware Design:
A Functional Approach

Bernhard Möller
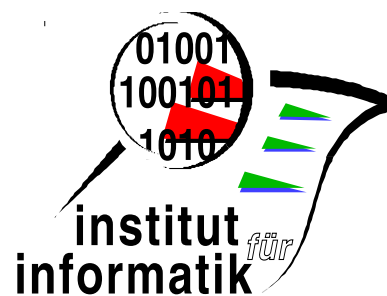
institut *für*
informatik

Deductive Hardware Design: A Functional Approach[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. The associated algebraic laws are then employed in deductive hardware design of basic combinational and sequential circuits as well as a brief account of pipelining. With this we tackle several of the IFIP WG 10.5 benchmark verification problems. Special emphasis is laid on parameterisation and re-usability aspects.

Part I: Introduction

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.

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.

2. Overview

We show 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.

We give derivations for basic combinational and sequential circuits as well as a brief account of pipelining. This tackles several of the IFIP WG 10.5 benchmark verification problems (see IFIP 94/97).

Special emphasis is laid on parameterisation and re-usability aspects.

3. The Framework

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).

3.1 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 .

3.2 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

$$| \text{ otherwise } = E_{n+1} .$$

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 .

3.3 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, ...] .

A useful operation on non-empty lists is the *folding* of their elements using a binary operator:
foldr1 f $[x_1,...,x_n]$ = f x_1 (f x_2 ... (f x_{n-1} x_n)...) .
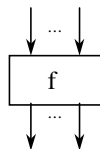
E.g. foldr1 (+) s computes the sum of all elements of s .

Part II: Combinational Circuits

4. A Model of Combinational Circuits

4.1 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

This function reflects the behaviour at one clock tick. 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*

$$f = g \quad \text{iff} \quad f\,x = g\,x \quad \text{for all } x .$$

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)

4.2 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.

Contrary to other approaches, 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.

In drawing diagrams we shall be liberal and use views in between rubber and rigid wiring.
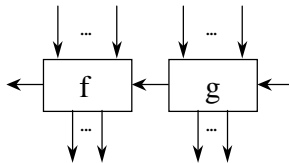In particular, we shall use various directions for the input and output arrows.
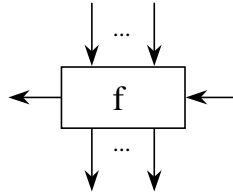
Example: Splicing along one wire is defined by

splice m f g (xs++[c]) = f (take m xs ++ [u]) ++ us
where (u:us) = g (drop m xs ++ [c])
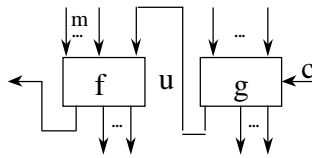
We straighten the lines to obtain the following form:

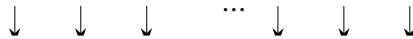**Lemma:** Splicing is associative in the following sense:

splice (m+k) (splice m f g) h = splice m f (splice k g h) .

Moreover, the identity id on singleton lists is its left and right neutral element.

Often we need to deal with wire bundles. In the case of circuits for binary arithmetic operators the wire bundles for the two operands will be interleaved:

9

$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \cdots \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$

To extract the corresponding sublists we use

- evns xs = [xs!!i | i <-[0.. length xs – 1], even i]

$$\downarrow \quad\quad \downarrow \quad\quad \downarrow \quad\quad \cdots \quad \downarrow \quad\quad \downarrow \quad\quad \downarrow$$

- odds xs = [xs!!i | i <- [0.. length xs – 1], odd i]

$$\downarrow \quad\quad \downarrow \quad\quad \downarrow \quad\quad \cdots \quad\quad \downarrow \quad\quad \downarrow \quad\quad \downarrow$$

The converse is shuf k which shuffles two lists of length k (represented as one list of length 2*k) and is specified by

        (shuf k xs) !! (2*i)   = x !! i
        (shuf k xs) !! (2*i+1) = x !! (k+i)
            for   length xs == 2*k and i <- [0..k-1] .

This is an implicit specification; its clauses will be used as algebraic laws in derivations. An explicit version is

        shuf k xs
          | length xs == 2*k   =
                    [x !! if even i then i`div`2 else i`div`2+k | i <- [0..2*k-1]]

5. Numbers and Their Representation

We head now for the derivation of some basic arithmetic circuits. We only treat natural numbers, but embedded into Int . As an auxiliary predicate we use

        below :: Int -> Int -> Bool
        n `below` m   =   0 <= n && n < m .

Then d is a base p digit iff d `below` p . Lists of base p digits are characterised by

        digits :: Int -> Int -> [Int] -> Bool
        digits p k xs =   length xs == k && all (`below`p) xs .

Now we define representation and abstraction functions between (the nonegative part of) Int and lists of base p digits. To cope with bounded word length, we parameterise then not only with p but also with the number of digits to be considered.
First we define the representation function

code :: Int -> Int -> Int -> [Int]

The result of code p k n is defined only for p > 1 and   n `below` p^k ; in this case it is
the base p representation of n in k digits precision (padded with leading zeros if
necessary):

        code p 0 0 = []
        code p (k+1) n = code p k (n `div` p) ++ [n `mod` p]

  Example: code 2 7 24 = [0, 0, 1, 1, 0, 0, 0]

For the corresponding abstraction function

    deco :: Int -> Int -> [Int] -> Int

the result of deco p k xs is the number represented by the list xs of k base p digits :

  deco p   0   []   =   0
  deco p (k+1) xs   =   (deco p k (init xs)) * p + last xs
These functions enjoy pleasant algebraic properties:

Lemma 5.1:
    The functions code and deco are inverses of each other:
                deco p k (code p k n) = n   if n `below` p^k
                code p k (deco p k xs) = xs   if   digits p k xs .
    Moreover, we have the decomposition/distributivity properties
                code p (j+k) (m * p^k + n) = code p j m ++ code p k n
                        if m `below` p^j && n `below` p^k
                deco p (j+k) (xs ++ ys) = (deco p j xs) * p^k + deco p k ys
                        if digits p j xs && digits p k ys .

6. Development of an Adder

As our first case study we derive a simple adder

        add :: Int -> Int -> [Int] -> [Int] .

The first parameter is the base, the second the number of digits we treat. For the
specification we assume that the list zs is the interleaving of the two summands, i.e., that
digits p (2*k) zs holds. Then

        add p k zs =   code p (k+1) (deco p k (evns zs) + deco p k (odds zs)) .

The length k+1 for the result list serves to accommodate a possible overflow digit.

6.1 The Unfold/Fold Strategy

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

- *Unfold* the definitions of deco and code .
- *Simplify* and *rearrange.*
- *Fold* with the definition of add to get recursive calls.

The derivation is driven by the case structure of   deco and code .

> **Case k=0.** We calculate:
>   add p 0 []
> =   code p 1 (deco p 0 [] + deco p 0 [])
> =   code p 1 0
> =   code p 0 (0 `div` p) ++ [0 `mod` p]
> =   [] ++ [0]
> =   [0]

This is the termination case; here the overflow digit is 0 .

> **Case k > 0**. We calculate, assuming xs = evns zs and ys = odds zs :
>   add p (k+1) (zs ++ [x,y])
> = code p (k+2) (deco p (k+1) (xs ++ [x]) + deco p (k+1) (ys ++ [y]))
> =   code p (k+2) ((deco p k xs)*p + x + (deco p k ys)*p + y)
> =   code p (k+2) ((deco p k xs + deco p k ys)*p + x + y)
> =   code p (k+1) (deco p k xs + deco p k ys + (x + y) `div` p) ++ [(x + y) `mod` p]

This expression is a*lmost* foldable, but because of the additional summand (x + y) `div` p we are stuck!

6.2 Generalisation

A strategy which helps frequently in such cases is *generalisation.* It works in two stages.

- First one introduces additional parameters, which may be completely new ones or abstractions of constants in the original specification. These constants may even be "invisible" neutral elements which need to be made explicit first.
- Then one uses the additional degrees of freedom to make the derivation go through.

The original problem is then solved by instantiating the solution for the generalised problem. This strategy is well-known from inductive proofs: there one frequently needs to generalise the induction hypothesis to make the proof go through.

In the case of our adder we introduce a parameter for the extra summand that prevented the folding. The generalised specification reads

cadd p k (xs ++ [c]) =
code p (k+1) (deco p k (evns xs) + deco p k (odds xs) + c)

If one wishes to interpret this, then the new parameter c is the carry. But note that it has been introduced purely formally, "without thinking", as part of the generalisation strategy! The original problem is retrieved via the *embedding*

add p k xs = cadd p k (xs ++ [0])

Now we can *replay the derivation* for cadd . This results in

cadd p 0 [c] = [c]
cadd p (k+1) (xs ++ [x,y,c]) =
    cadd p k (xs ++

It turns out that we need an additional assertion about c , namely c `below` 2 , to ensure that the expression (x+y+c) `mod` p always yields a proper digit. Fortunately this assertion is preserved as an invariant of the recursion, i.e., if it holds for c it also holds for the new carry (x+y+c) `div` p .

6.3 Modularization

The resulting expression for the recursive case is very complex. We structure it by packing the two expressions for last digit and new carry in cadd into a function

fa p [x,y,c] = [(x+y+c) `div` p, (x+y+c) `mod` p] .

Now we may use splicing to obtain

cadd p (k+1) =   splice (2*k) (cadd p k) (fa p) .
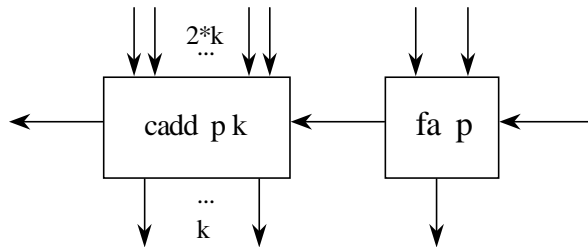
Of course, fa is the full adder function. But note again that this is introduced purely formally!

For fixed n we may now unwind the recursion to obtain the well-known regular design of the carry ripple adder:
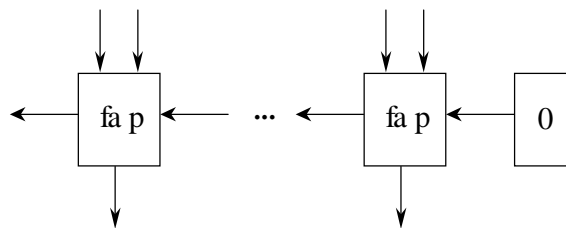
The associativity of splicing is essential here; it allows this "parenthesis-free" graphical layout.

Based on the decomposition properties for code and deco we can also show a decomposition property for cadd :

Lemma 6.1: cadd p (k+m) = splice (2*k) (cadd p k) (cadd p m)
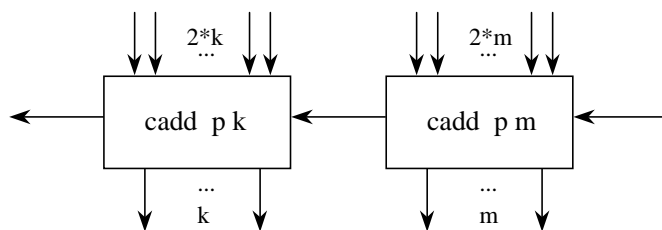
Proof:

Consider a list zs ++ zs' ++ c with length zs = 2*k and length zs' = 2*m and set xs = evns zs, ys = odds zs , xs' = evns zs', ys' = odds zs' . Then we calculate, using Lemma 5.1:

14

```
       cadd p (k+m) (zs++zs'++ [c])
   = code p (k+m+1) ( deco p (k+m) (xs++xs') + deco p (k+m) (ys++ys') + c)
   = code p (k+m+1) ((deco p k xs)*p^m + deco p m xs' +
                      (deco p k ys)*p^m + deco p m ys' + c)
   = code p (k+m+1) ((deco p k xs + deco p k ys + d)*p^m + r)
           where (d,r)  =  (z `div`p^m, z `mod` p^m)
                      z  =  deco p m xs' + deco p m ys' + c
   = code p (k+1) (deco p k xs + deco p k ys + d) ++  code p m r
           where (d,r)  =  (z `div`p^m, z `mod` p^m)
                      z  =  deco p m xs' + deco p m ys' + c
   = code p (k+1)(deco p k xs + deco p k ys + d) ++ us
           where (d:us)  =  code p (m+1) (deco p m xs' + deco p m ys' + c)
   = cadd p k xs ys d ++ us
           where (d:us)  =  cadd p m xs' ys' c
   = splice (2*k) (cadd p k) (cadd p m) (zs++zs' ++ [c])
```

Note that this proof has been performed at the specification level and hence holds for all correct implementations, not just the carry ripple adder!
This allows modular decomposition of large adders into smaller ones, say 4-bit modules. Again the associativity of splicing is essential here.

Since decomposition holds for all implementations, we may even use combinations of various adders, e.g. a (carry ripple) splicing of 4-bit carry lookahead adders (see below).

Here we have a typical combination of *parameterisation* and *modularization*.

It should also be noted that we have
```
        fa p [x,y,c]  =   cadd p 1 [x,y,c]
```
so that the carry ripple design can also be seen as the result of an iterated application of Lemma 6.1

6.4 Abstraction

We now review the derivation to find the algebraic laws that went into it. We abstract from the particular case of addition and define a general function

```
        digrep :: (Int -> [Int] -> [Int]) -> Int -> Int -> [Int] -> [Int]  .
```

The idea is that digrep f p k (zs ++ [c]) works on the interleaved digit representation zs of two natural numbers and a "carry"   c . Again, p is the base and k the number of digits we treat. The function f takes into account the number k of digits and a list of two "proper" arguments and a "carry". If digits p (2*k + 1) (zs ++ [c]) holds, we specify

digrep f p k (zs ++ [c]) = f k [deco p k (evns zs), deco p k (odds zs), c] .

To retrieve the adder function, we have to set, for m,n `below`p^k ,

    f k [m,n,c] = code (k+1) (m+n+c)   (*) .

For the base case k=0 we calculate

    digrep f p 0 [c]
  = f 0 (deco p 0 []) (deco p 0 []) c
  = f 0 0 0 c .

For the inductive case we could now also replay the derivation of cadd for digrep .
However, as the remark at the end of Section 6.3 shows, it is more advantageous to head
for a decomposition property of digrep . By analysing the proof of Lemma 6.1, we can
find a sufficient condition on f that makes the proof go through in general. Following
Hanna et al. 90 we call f *factorizable* if

    f (j+k) [m*p^k+q, n*p^k+r, c] = splice 2 (f j) (f k) [m,n,q,r,c]

holds for all natural numbers j,k,m,n,p,q,r . Now Lemma 6.1 generalises to

Theorem 6.2 (Factorization Theorem):
    Let f be factorizable. Then
            digrep f p (k+m) = splice (2*k) (digrep f p k) (digrep f p m) .
Proof:
    digrep f p (k+m) (zs++zs'++ [c])
  = f (k+m) [deco p (k+m) (xs++xs'), deco p (k+m) (ys++ys'), c]
  = f (k+m) ((deco p k xs)*p^m + deco p m xs')
            (deco p k ys)*p^m + deco p m ys') c
  = splice 2 (f k) (f m) [deco p k xs, deco p k ys, deco p m xs', deco p m ys', c]
  = f k [deco p k xs, deco p k ys, d] ++ us
            where (d:us) = f m [deco p m xs', deco p m ys', c]
  = digrep f p k (zs ++ [d]) ++ us
            where (d:us) = digrep f p m (zs'++[c])
  = splice (2*k) (digrep f p k) (digrep f p m) (zs++zs' ++ [c])

This is in fact F. K. Hanna's Factorization Theorem (see again Hanna et al. 90), which
gives a general scheme for correct implementations of iterative arithmetic circuits. The
proof of Lemma 6.1 contains a section which uses Lemma 5.1 to show that (*) above
defines a factorizable f ; the remainder is isomorphic to the proof of Theorem 6.2.

Using this theorem and the fact that digrep f p 1 = f p 1 we can unwind digrep f k into a
regular layout:

Corollary 6.3: For $k > 0$ we have

$$\text{digrep } f \; p \; k \; = \; \text{foldr1 (splice 2) (copy } k \; (f \; p \; 1)) \; .$$

Another instance of this is a comparator circuit, described by

$$\text{digrep } f \; p \; k \quad \text{where} \quad f \; p \; [m,n,c] = [\text{eq } m \; n \; \wedge \; c] \qquad\qquad (\text{**}) \; .$$

Here, eq $m \; n = $ if $m == n$ then 1 else 0 and $b \wedge c = $ b*c , so that we have numerical representations of the usual Boolean operations. It is straightforward to show that also (**) defines a factorizable f . To obtain a comparator circuit, we have to instantiate c appropriately, viz. by the neutral element 1 of $\wedge$, and unwind the specification using the Factorization Theorem. This results in
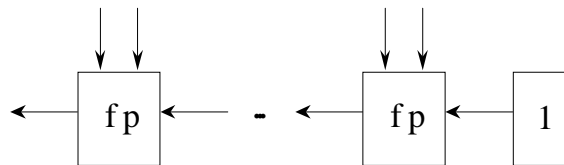
7. Successor (Counting)

Next we want to derive a counter circuit, i.e., an implementation of the successor function on digit representations. The specification reads

   succ :: Int -> Int -> [Int] -> [Int]
    succ p k xs = code p (k+1) (decode p k xs + 1)

This is quite similar to the adder specification. We therefore try to *re-use* the adder design. Formally we need to reduce succ to add; this is done by making the hidden neutral element 0 of addition visible so that we have a second operand for addition. We calculate:

      succ p k xs
    = code p (k+1) (decode p k xs + 1)
    = code p (k+1) (decode p k xs + 0 + 1)
    = code p (k+1) (decode p k xs + decode p k (copy k 0) + 1)
    = cadd p k (shuf k (xs ++ copy k 0) ++ [1])

Although this is a first correct implementation, it is too inefficient. The fact that in the unwound version we have calls of the form fa [x,0,c] may be used to simplify the design. Define an auxiliary function

$$\text{ha } [x,c] \; = \; \text{fa } [x,0,c] \; = \; [(x+c) \text{ `div` } p \, , \, (x+c) \text{ `mod` } p]$$

Of course, ha is the half adder function. But again it has been introduced purely formally.
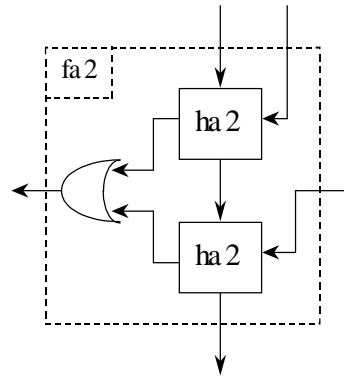The simplified design looks as follows:
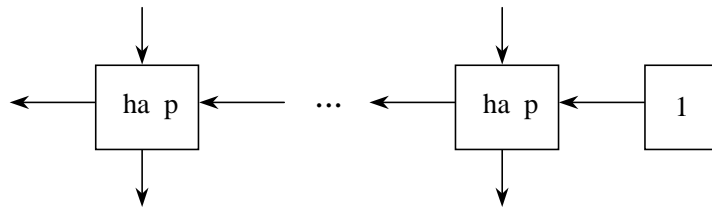
8. Specialization: Base 2

For p=2 we obtain the usual
representations

$$\text{ha 2 } [x,c] \; = \; [x \wedge c, \; x >< c]$$

$$\text{fa 2 } [x,y,c] \; = \; [\, d \vee e, \; z]$$
$$\text{where } [d,u] \; = \text{ha 2 } [x,y]$$
$$[e,z] \;\; = \text{ha 2 } [u,c]$$

Here, $\wedge, \vee$ and $><$ are the arithmetic representations of the Boolean operations on base
2 digits, e.g.

$$x \; \wedge \; y \; = \; x*y \, .$$

9. The Carry Lookahead Adder

It is well known that the carry ripple adder is time-inefficient, since the length of the longest path through the design (along which the carries ripple) is proportional to the number of digits processed. So there have been various proposals to speed up the carry computation. One idea is to compute the carries in parallel with the sums; this leads to the carry lookahead adder which we want to derive formally now.

Let the modules in the carry ripple adder be numbered from the right starting with 0 and let x_i, y_i and c_i be the i-th input digits and carries (where c_0 is some given value) . From the carry ripple design we read off the recurrence equation

$$c (i+1) = (p_i \text{ `andN` } c_i) \text{ `orN` } g_i \text{ where}$$
$$(g_i , p_i) = (x_i \text{ `andN` } y_i , x_i \text{ `xorN` } y_i)$$

By usual techniques for solving recurrences we obtain a closed form for the carries:

$$c (i+1) = \text{foldr1} (\backslash/)[(\text{foldr1} (/\backslash) [p_k \mid k <\text{- } [j+1..i]) \text{ `andN` } g_j \mid j <\text{- } [-1..i]]$$
$$\text{where } g (-1) = c_0$$

Here foldr1 is a predefined *Gofer* function which takes a binary operator and a non-empty list and combines all list elements by that operator, associating them to the right.

For reasons of space we draw the picture of the carry lookahead computation only for for 3 digits:

Using this form of carry computation results in a circuit in which the path length is independent of the number of digits processed. This gain is bought at the expense of fan-in proportional to the number of digits. So for electrical reasons this design is meaningful only for small numbers of digits, say 4 or 8. But from our above decomposition property we know that we may connect several carry lookahead adders in a carry ripple fashion to obtain a correct adder which will then be faster by a factor 4 or 8 than the original pure carry ripple adder.
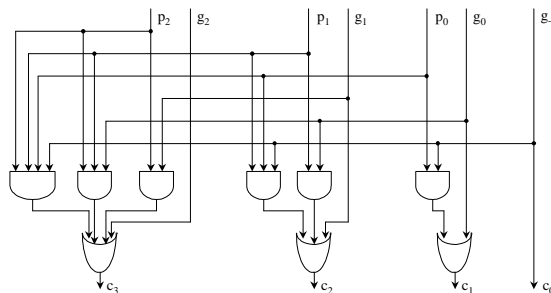
10. More About Wiring

So far we have mostly described connections using the rubber view of wires ("logical connection"). We now sketch how to step from the logical connection to a topology with rigid wires, crossings and fan-out.

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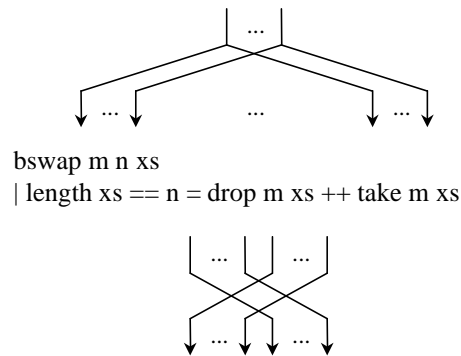

10.1 Basic Wiring Elements

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] = []
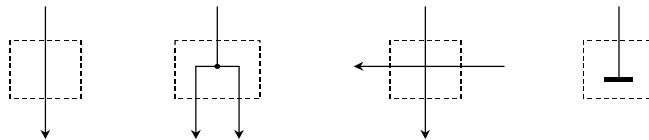
These operations are extended to wire bundles:

    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


The identity id is predefined polymorphically by id y = y and hence doesn't need to be extende to wire bundles. The sink can be handled by setting generally sink xs = [] . We will discuss other versions later.

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

$$ide [] = []$$

10.2 Sequential and Parallel Composition

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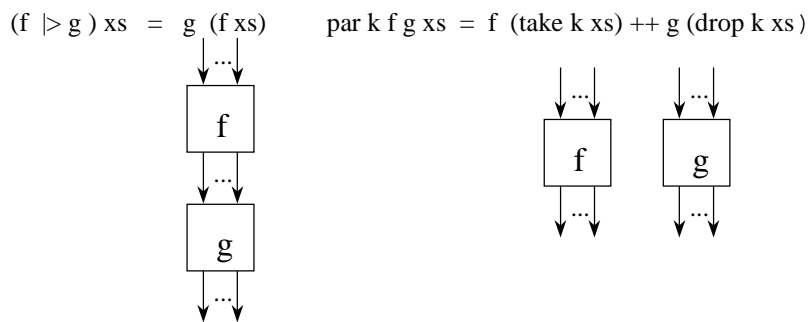
We abbreviate par 1 by the infix operator ||| .

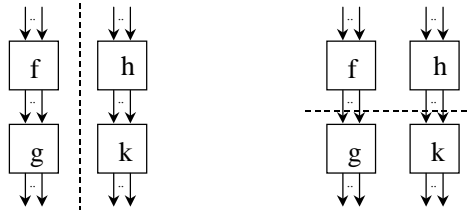## 10.3 Basic Laws (Network Algebra I)

All semantic models for graph-like networks should enjoy a number of natural properties which reflect the abstraction that lies in the graph view. A systematic account of these properties has been given in Stefanescu 94.

Associativity:

- $\quad$ f $\,|\!\!>\,$ (g $\,|\!\!>\,$ h) $\;=\;$ (f $\,|\!\!>\,$ g) $\,|\!\!>\,$ h
- $\quad$ par (m+k) (par m f g) h $\;=\;$ par m f (par k g h)

Abiding Law:

- $\quad$ par m (f $\,|\!\!>\,$ g) (h $\,|\!\!>\,$ k) $\quad=\;$ (par m f h) $\,|\!\!>\,$ (par n g k)

Neutrality:

- $\quad$ id $\,|\!\!>\,$ f $\;=\;$ f $\;=\;$ f $\,|\!\!>\,$ id
- $\quad$ par m f ide $\;=\;$ f $\;=\;$ par 0 ide f

Idempotence:
- $\quad$ swap $\,|\!\!>\,$ swap $\;=\;$ id

Whereas associativity and abiding just allow "parenthesis-free layouts", use of neutrality or idempotence means simplification/complexification of abstract layouts.

10.4 Selection

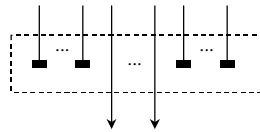Using parallel composition we can now give alternative definitions for block identity and sink:

- bid n = foldr1 (‖‖) (copy n id)
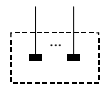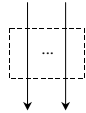
- bsink n = foldr1 (‖‖) (copy n sink)

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))))
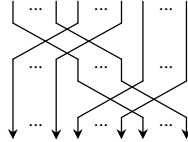

We have the following fusion rule:
bfork 2 |> par (j-i) (sel n i j) (sel n j k) = sel n i k

10.5 Recursions for the Bundle Operations

Using sequential and parallel composition we can reduce the bundle operations to the primitives.

Example:
- bswap m 0 = ide   bswap 0 n = id
- bswap 1 1 = swap
- bswap k (k+m+n) = par (k+m) (bswap k (k+m)) (bid n) |>
                              par m (bid n) (bswap k (k+n))

11. 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 = <u>f</u> where <u>f</u> = \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)

Example:
CA ([x /\ y] ++ [y >< x]) =
bfork 2 |> ((sel 2 0 ||| sel 2 1) |> /\) ||| ((sel 2 1 ||| sel 2 0) |> ><))
This can, of course be simplified to
bfork 2 |> ((bid 2 |> /\) ||| (swap |> ><))

The basic rules above lead to circuits involving very high fan-outs. More refined rules avoid this, e.g.

CA (E1 ++ ... ++ En) = CA E1 ||| ... ||| CA En
if ID [E1,...,En] = ID E1 ++ ... ++ ID En, i.e., if the sublists of formal
parameters are disjoint and in order.

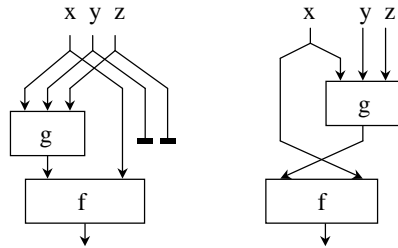The situation can often be improved using swaps .

Example:
- We have
CA f (g [y,z] ++ [x]) = bfork 2 |> (g ||| sel 3 0 1) |> f

- A simpler version is
  CA f (swap ([x] ++ g [x,y])) =
  (fork ||| id) |> (id ||| g) |> swap |> f

12. A Further Example: Shuffling

Recall the specification of the shuffle operation from Section 4.2:

    (shuf k xs) !! (2*i)   =   x !! i
    (shuf k xs) !! (2*i+1) = x !! (k+i)
        for   length xs == 2*k and i <- [0..k-1]

Some calculation yields the following inductive version:

    shuf 0 = id
    shuf 1 = id
    shuf (k+1) = (par 1 id (par k (cshiftl k) id)) |> (par 2 id (shuf k))

    cshiftl k = foldr1 (splice 2) (copy k swap)
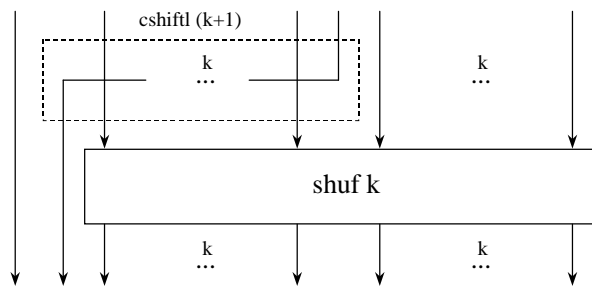
For further details on wiring we refer to Hotz et al. 86 and Molitor 91.

Part III: Sequential Hardware

13. A Model of Streams

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 .

14. Networks

Again we model bundles of in/outputs by lists, this time of streams. By polymorphism we can re-use all our connection primitives, such as |>, par, fork , swap and splice and their laws for stream transformers as well.
Our diagrams will now be drawn sideways:
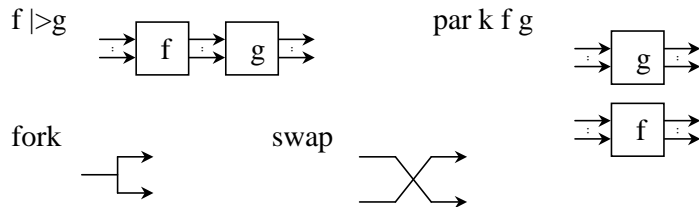
The input/output streams are numbered from bottom to top in the respective lists.

15. Lifting and Constant

To establish the connection with combinational circuits we need to iterate their behaviour in time. To this end 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)
```
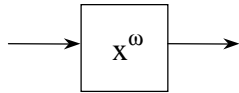

Here const is a predefined *Gofer* function that produces a constant unary function from a value.

16. 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 x as the initial value:

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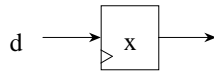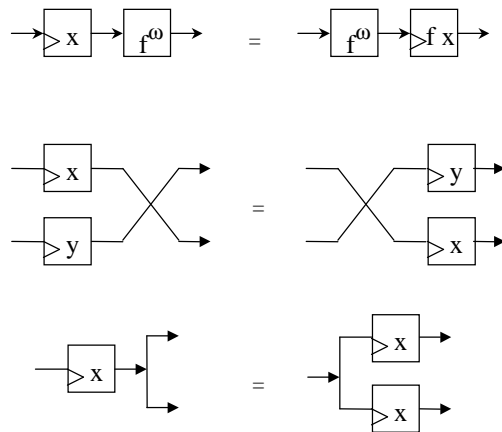

To push delays through larger networks we have the following

Lemma 16.1 (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

$\rightarrow \triangleright x \rightarrow f^{\omega} \rightarrow$ = $\rightarrow f^{\omega} \rightarrow \triangleright f\ x \rightarrow$

(diagram of crossing streams with boxes $\triangleright x$, $\triangleright y$ = $\triangleright y$, $\triangleright x$)

(diagram of box $\triangleright x$ split = split into $\triangleright x$, $\triangleright x$)

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.

17. Example: The Single Pulser

To show the model at work we will treat a *single pulser* as our first example. The informal specification requires it to emit a unit pulse whenever a pulse starts in its input stream.

17.1 Formal Specification

We model this by a transformer of streams of Booleans. A *pulse* is a maximal time interval on which a stream is constantly True. First we characterise those time points at which a pulse starts formally by

    startPulse :: Stream Bool -> Time -> Bool
    startPulse d t = d t && (t==0 || not(d (t-1))

Note that by Time -> Bool = Stream Bool we may view startPulse also as a stream transformer.
Now we can give the formal specification of the pulser:

    pulser [d] = [\t -> startPulse d t] , i.e.,

pulser [d] = [startPulse d]

17.2 Derivation of a Pulser Circuit

For t = 0 we calculate
        startPulse d 0
    = d 0 && (0==0 || not (d (0-1))
    = d 0
For t > 0 we have
        startPulse d t
    = d t && (t==0 || not (d (t-1))
    = d t && not (d (t-1))
    = d t && not ((x & d) t)
for arbitrary x . Now we try to choose the initialisation value x such that
        startPulse d t = d t && not ((x & t) 0)
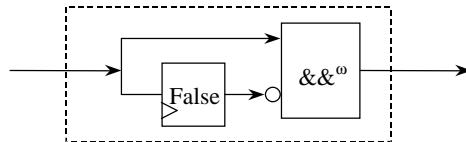holds also for t=0, i.e.,
        d 0 = d 0 && not x
This is satisfied for all values d 0   iff   x = False .

Now combinator abstraction yields
      pulser = fork |> (id ||| ((False &) |> lift1(not))) |> lift2 (&&)

18. Feedback

18.1 The Feedback Operation

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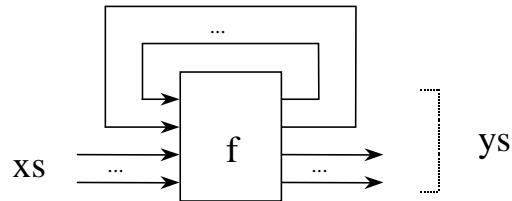inputs. We use
feed :: Int -> ([a] -> [a]) -> ([a] -> [a])
where the first parameter indicates how many outputs are fed back. The definition reads
        feed k f xs = codrop k ys
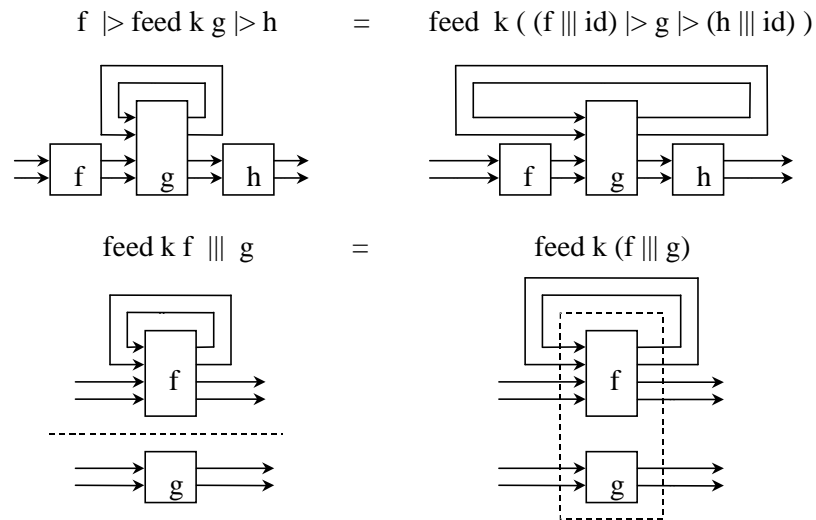   where ys = f (xs ++ cotake k ys)

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*.
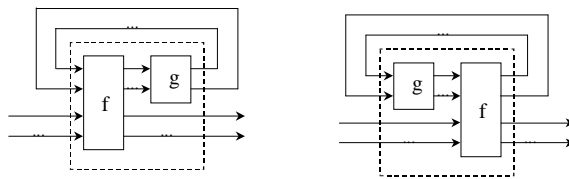
18.2 Properties of Feedback (Network Algebra II)

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 Stefanescu 94.
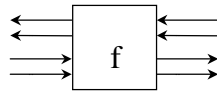
$$f \mathbin{|>} \text{feed } k \text{ } g \mathbin{|>} h \quad = \quad \text{feed } k \text{ } (\text{ } (f \mathbin{|||} \text{id}) \mathbin{|>} g \mathbin{|>} (h \mathbin{|||} \text{id}) \text{ })$$

$$\text{feed } k \text{ } f \mathbin{|||} \text{ } g \quad = \quad \text{feed } k \text{ } (f \mathbin{|||} g)$$

Shifting a block:

feed k (f |> (id ||| g))   =   feed m ((id ||| g) |> f)


19. 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 .
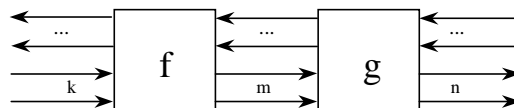
We define therefore

$$\text{connect k m n f g xs} = \text{take n zs ++ drop m ys}$$
$$\text{where} \quad \text{ys} = \text{f (take k xs ++ drop n zs)}$$
$$\text{zs} = \text{g (take m ys ++ drop k xs)}$$

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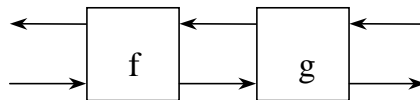
Lemma: Interconnection is associative:

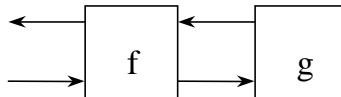$$\text{connect m n p (connect k m n f g) h} = \text{connect k m n f (connect m n p g h)}$$

The proof can be given using purely the laws of network algebra. Hence it is valid for all models of network algebra, not just our particular one. Also, connect has the identity id as its neutral element.
Two interesting special cases are

- $\quad$ f =‖= g = connect 1 1 1 f g
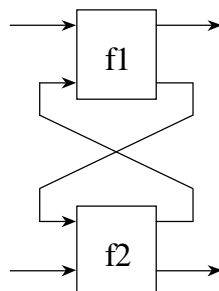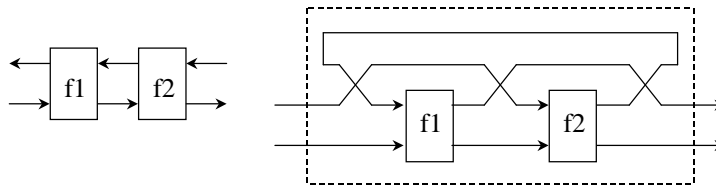

- $\quad$ f = | g = connect 1 1 0 f g

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:

$$f1 =\!\|\!= f2 \qquad = \qquad \text{feed } 1 \ (\ (\text{id} \ ||| \ \text{swap}) \ |> \ (f1 \ ||| \ \text{id}) \ |> \ (\text{id} \ ||| \ \text{swap}) \ |>$$
$$(f2 \ ||| \ \text{id}) \ |> \ (\text{id} \ ||| \ \text{swap}) \ )$$

20. A Convolver

We want to tackle a somehwat more involved example now. In particular, we want to prepare the way to systolic circuits.

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 .

20.1 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. Since we have taken such an abstraction step already in Section 6.4, we do not want to repeat this here.

20.2 About Error Handling

We have not used nonneg here but rather played everything back to the "totally undefined" element bot defined by a nonterminating recursion. However, the only essential assumption about bot is the strictness property $x + \text{bot} = \text{bot}$ .This could also be achieved by introducing an additional error element using *Gofer*'s facilities for defining variant record types and adapting addition accordingly:

```
data Error a  =  Proper a | Err
instance Num a => Num (Error a) where
Proper x + Proper y  =  Proper (x+y)
_ + _  =  Err                        --   etc.
```

Since this is somewhat cumbersome, though, we have chosen the above method.

20.2 Derivation of a Convolver Circuit
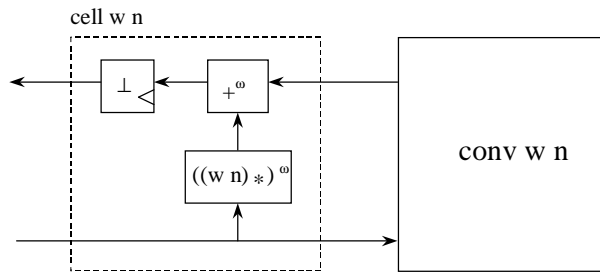
For t >= 0 and [e] = conv w 0 d we calculate

    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 k [li,ri] =   [bot & lift2 (+) (lift1 (w k *) [li], [ri]), li]

cell w n

Unwinding the recursion. For fixed n > 0 we obtain again a regular design:

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

After simplification of the rightmost cell this yields
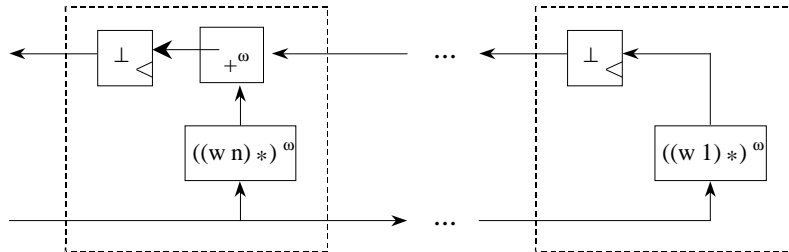
However, we have a long broadcasting path (fanout n) at the bottom.

20. 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.

21. 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.

21.1 Interleaved Streams

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

$$\text{split } k \text{ } j \text{ } d \text{ } t \text{ } = \text{ } d \text{ } (k*t + j) \text{ .}$$

So split k j d is the j-th of the k component streams where numbering starts with 0 again. 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 21.1: (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 .

21.2 The Slowdown Function

Now the slowdown function is specified implicitly by
        (slow k f) |> split k j   = (split k j) |> f .

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 21.2: 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) .

21.3   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 21.3: Suppose that   (x&) |> f = f |> (y&) . Then also
   (x&) |> slow k f = (slow k f) |> (y&) .

22. 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, employing the delay propagation rules, we push the second delay introduced by the slowdown through the various modules. We perform the derivation pictorially:
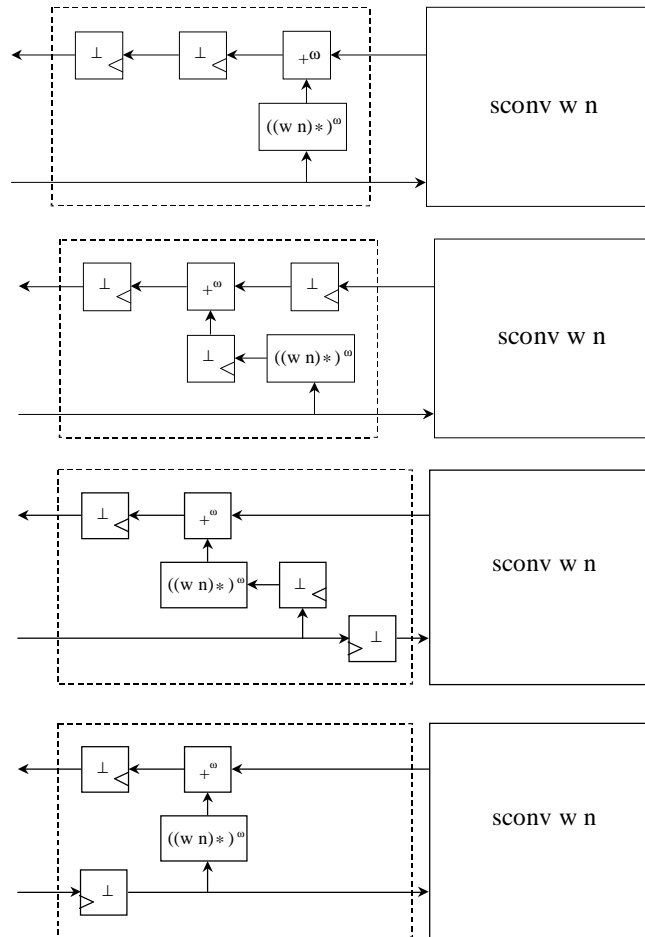
The step of pushing the delay through sconv w n is justified Lemma 21.3. Unwinding the recursion again we obtain a regular systolic design:

```
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
```

This simplifies into
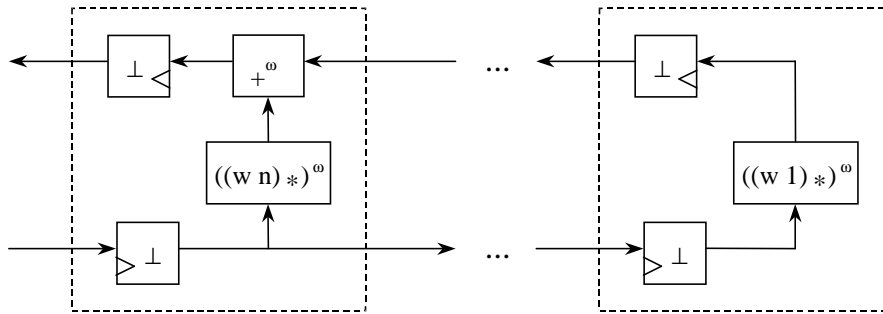
Of course, the techniques we have developed do not only apply to the convolver, but are of general interest for the derivation of systolic implementations of circuits. As a further case study, a systolic recognizer for regular expressions is developed in Möller 98.

23. Pipelining

As a final example we want to leave the level of circuits and step up to questions about microprocessor architectures. To exemplify our approach there we give a brief account of the essence of pipelining.
Let a be a set of instruction addresses, i a set of instructions and s a set of machine states. Assume, moreover, a function
        fetch :: a -> s -> i
that obtains the instruction stored under an address in the current state and a function
        exe :: i -> s -> s
for executing an instruction in a state to yield a new state. Then the fetch/execute-cycle of a machine can be defined by the function

        run :: [a] -> s -> s
        run [] q = q
        run (x : xs) q = run xs (exe (fetch x q) q)

We now want to uncouple the fetch and execute phases so that they can be done in parallel. This done by a suitable embedding into a function which has as parameters an instruction to be performed currently and a list of addresses of further instructions:

        pipe :: [a] -> i -> s -> state
        pipe xs j q = run xs (exe j q)

The original function `run` is reduced to `pipe` by the equations

```
run [] q = q                              -- done
run (x : xs) = pipe xs (fetch x q) q      -- put 1st instruction into
                                          -- pipeline and run that
```

The goal is now again a version of pipe that is independent of run . As the termination case we obtain

pipe [] j q = exe j q .

Next we calculate

$$pipe\ (x : xs)\ j\ q$$
$$=\ run\ (x : xs)\ (exe\ j\ q)$$
$$=\ run\ xs\ (exe\ (fetch\ x\ q')\ q')$$
where q' = exe j q
$$=\ \text{-- assume now that execution does not change the contents of}$$
-- the program memory, i.e., assume fetch a q' = fetch a q
run xs (exe (fetch x q) q') where q' = exe j q
$$=\ pipe\ xs\ (fetch\ x\ q)\ (exe\ j\ q)$$

This means that fetching the next instruction can be done in parallel with executing the current one.

Note that the derivation is completely polymorphic; no assumptions are made about the types a, s, and i . The only assumption is the property

fetch x (exe j q) = fetch x q .

In particular, the transformation can be iterated to obtain pipelines with several stages if exe can be decomposed into further subfunctions.

24. 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.

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

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)

C. Delgado Kloos: Semantics of digital circuits. LNCS **285**. Springer 1987

C. Delgado Kloos, W. Dosch, B. Möller: Design and proof of multipliers by correctness-preserving transformation. In P. Dewilde, J. Vandewalle (eds.): Proc. IEEE International Conference on Computer Systems and Software Engineering CompEuro 92. IEEE Computer Society Press 1992, 238-243

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

K. Hanna, N. Daeche, M. Longley: Specification and verification using dependent types. IEEE Trans. Softw. Eng. **16:9**, 949-964 (1990)

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)

IFIP 94/97: IFIP WG 10.5 Verification Benchmarks. Reachable via internet under http://goethe.ira.uka.de/hvg/benchmarks.html

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

C.E. Leiserson, J.B. Saxe: Optimizing synchronous systems. J. VLSI and Computer Systems **1**, 41-68 (1983)

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: An algebraic approach to systolic circuits. Institut für Informatik, Universität Augsburg, Report 1998-01, January 1998

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. Stefanescu: Algebra of flownomials. Institut für Informatik, Technical Unicersity Munich, Report TUM-I9437, 1994