Deductive Hardware Design: A Functional Approach*

Bernhard Möller

Institut für Informatik, D-86135 Augsburg, Germany

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 *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 parametrization and re-usability aspects.

1 Introduction

1.1 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.
- In the first stage the main emphasis lies on *correctness*;
- in further stages transformations can be used to *increase efficiency*.
- 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.

* To appear in: B. Möller, J.V. Tucker (eds.): Prospects of Hardware Foundations. Springer LNCS (in preparation). This research was partially sponsored by Esprit Working Group 8533 NADA - New Hardware Design Methods.

Note that we do not view deductive design as alternative to, but complementary to verification. In fact, usually transformational derivations will be interleaved with verification of lemmas needed on the way. Conversely, verification may benefit from incorporation of standard reduction strategies based on algebraic laws.

There is a variety of approaches to deductive design, eg.

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

We shall follow the latter (see e.g. [4, 19]) and use mainly

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

1.2 Overview

We exemplify deductive hardware design in the particular area of

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

Directionality means that input and output are clearly distinguished. As for synchronicity we assume that our systems are clocked, in particular, that the clock period is long enough that all submodules stabilize their output within it.

The approach generalizes with varying degrees of complexity to adirectional systems, asynchrony, non-determinacy or continuous time. Adirectionality, as found eg. in buses, may be modeled better in a relational than in a functional setting, handshake communication may more adequately be treated by systems such as CCS, CSP or ACP.

references, Josephs

We show deductive design of basic combinational and sequential circuits and give a brief transformational account of pipelining. In particular we derive systolic circuits and tackle several of the IFIP WG 10.5 benchmark verification problems [11]. Special emphasis is laid on parameterization and re-usability aspects.

1.3 The Framework

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

- Functional languages supports various views of streams directly, eg. lazy lists or functions from time to data as first-class objects.
- Polymorphism allows generic formulations and hence supports re-use.
- Since all specifications are executable, direct prototyping is possible.
- Functional languages are being considered for their suitability as bases of modern hardware description languages; an example is the (unfortunately abandoned) language MHDL [20].
- Many other approaches to hardware specification and verification also use higher-order concepts to good advantage (see e.g. [8]).
- A transformation system ULTRA for the *Gofer* sublanguage of *Haskell* is being constructed at the University of Ulm under the direction of H. Partsch. It is an adaptation of the system CIP-S [3]. Several of our paper and pencil derivations have been replayed on a prototype version of ULTRA to check their correctness by machine. The set of transformation rules given here can be re-used for further derivations directly on the system.

Readers not familiar with *Haskell* will find a brief review of its essential constructs in the Appendix.

some more text

Part I: Combinational Circuits

2 A Model of Combinational Circuits

text A brief review of the essential concepts of *Haskell* that are used in this paper can be found in the Appendix.

2.1 Functions as Modules

A combinational module will be modeled as a function taking a list of inputs to a list of outputs. This function reflects the behaviour at one clock tick. This reflects the underlying assumption of synchrony: the complete tuple of inputs must be available before the output can be computed.

tie this in with laziness

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, since *Haskell* does not allow heterogenous lists.

So we assume some basic type `a` that is the direct sum of all data types involved, such as integers, booleans, bytes etc. Then a function `f` describing a module with `m` inputs and `n` outputs will have the type `f :: [a] -> [a]` but be defined only for input lists of length `m` and always produce output lists of length `n` . Assuming

```
[o1,...,on] = f [i1,...,im]
```

we represent such a module diagrammatically as

We now discuss briefly the role of functions as modules of a system. In a higher-order language such as *Haskell* 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 advantages in some situations, it can become quite tedious in others. So we prefer to have the possibility to switch.

The basis for showing equality of two expressions that yield functions as their values is the extensionality rule

$$\texttt{f} = \texttt{g} \text{ iff } \texttt{fx} = \texttt{gx} \text{ for all } \texttt{x} .$$

Many algebraic laws we use are equalities between functions, interpreted as extensional equalities.

Example 21 Function composition is defined in *Haskell* 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)
```

□

2.2 Modeling 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 (e.g. [10, 12]), 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.

This avoids the introduction of wiring combinators at a too early stage and carrying them through all the derivation in an often tedious manner.

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. So input arrows may not only enter at the top but also from the right or from the left; an analogous remark holds for the output arrows.

Example 22 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])
Assume now
\begin{verbatim}
xs = [x1,...,xm,x(m+1),...,xn]
[u1,u2,...,ul] = g [x(m+1),...,xn,c]
[v1,v2,...,vk] = f [x1,...,xm,u1]
```

Then we may depict `splice m f g xs` as

We straighten the wires to obtain the following form:

This form will be used for passing carries from one module to the next. □

Lemma 23 *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.*

This is essential in that it shows that the functional model adequately describes the graphical and layout views of hardware: there are no "parentheses" in circuits, and hence the mathematical model should not depend on parenthesization.

2.3 Wire Bundles

Often we need to deal with wire bundles. In the case of circuits for binary arithmetic operators it is usually assumed that the wires for the single bits of the two operands are interleaved (or *shuffled*) in the following fashion:

So the bits for one operand occur at even positions in the overall list of inputs, those for the other one at odd positions. To extract the corresponding sublists we use

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


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

Recursive versions of these functions are

```
evns [] = []
evns (x:xs) = x : odds xs
odds [] = []
odds (x : xs)  = evns xs
```

The converse is `shuf k` which shuffles two lists of length `k`, say `ys` and `zs`, into one list of length `2*k`.

Following our general principle that every module takes *one* list of inputs, we have to concatenate `ys` and `zs` into one list `xs`. Then `shuf` is specified by

```
(shuf n xs) !! (2*i)    =  xs !! i
(shuf n xs) !! (2*i+1)  =  xs !! (n+i)
```

for `length xs == 2*n` and `i <- [0..n-1]`. This is an implicit specification; the patterns on the left hand side are not legal *Haskell* patterns. However, the clauses of this specification will be used as algebraic laws in derivations. An explicit version is

```
shuf n xs
   | length xs == 2*n   =  ileave (take n xs) (drop n xs)

ileave [] [] = []
ileave (y : ys) (z : zs) = y : (z : ileave ys zs)
```

3 Numbers and Their Representation

We leave now briefly the field of circuits. As a preparation for the derivation of some basic arithmetic circuits we need some definitions concerning the representation of natural numbers w.r.t. a base `p`. To simplify matters we use the nonnegative part of the *Haskell* type `Int` for treating natural numbers; a different possibility would be the definition of a recursive data type

```
data Nat = Zero | Succ Nat .
```

We avoid this, since it would necessitate a lengthy redefinition of all arithmetic operators.

To characterize `p`-adic digits we use the auxiliary predicate `below :: Int -> Int -> Bool`

```
n `below` m  =  0 <= n && n < m .
```

Then `d` is a `p` digit iff `d `below` p`. Lists of length `k` consisting only of `p`-adic digits are characterized 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 nonnegative part of) `Int` and lists of `p`-adic digits. To cope with bounded word length, we parameterize them 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]
```

Its first argument `p` is the base of the number system; for `n > 0` the result of `code p k n` is defined only for `p > 1`. The second argument `k` is the number of digits we want to consider. With `k` we can represent only numbers `n` with `n` `` `below` p^k `` exactly. Hence for other numbers `n` the result of `code p k n` is undefined. Otherwise it is the `p`-adic representation of `n` in `k` digits precision, padded with leading zeros if necessary) The definition reads

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

Example 31 `code 2  5  24  =  [1, 1, 0, 0, 0]`

```
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 `p`-adic digits. Again `k` is the number of digits expected; the result is undefined if `xs` has a length different from `n`. The defintion reads

```
deco p    0   []  =  0
deco p (k+1)  xs  =  (deco p k (init xs)) * p + last xs
```

These particular abstraction and representation functions were introduced in [6]. They are useful in that they admit induction/recursion over the parameter `k`. They enjoy pleasant algebraic properties:

Lemma 32 *The functions* `code` *and* `deco` *are inverses of each other:*

```
deco p k (code p k n)  =  n    <==   n `below` p^k
code p k (deco p k xs) = xs    <==   digits p k xs .
```

We have the decomposition/distributivity properties

```
code p (j+k) (m * p^k + n)  =  code p j m ++ code p k n
  <==   m `below` p^j  &&  n `below` p^k
deco p (j+k) (xs ++ ys)  =  (deco p j xs) * p^k + deco p k ys
  <==   digits p j xs  &&  digits p k ys .
```

By the sign `<==` we mean logical implication from right to left; it is not part of official *Haskell*.

These properties are verified by structural induction over the lists involved using the following properties of `div` and `mod` (see [6]):

Lemma 33
```
(x+y) `mod` z = y `mod` z   <==   (x `mod` z) = 0
(x+y) `div` z = x `div` z + y `div` z   <==   (x `mod` z) = 0

(x `div` p^m) `div` p^n  =  x `div` p^(m+n)
(x `mod` p^m) `mod` p^n  =  x `mod` p^(min m n)
(x `mod` p^m) `div` p^n  =  (x `div` p^n) `mod` p^(max 0 (m-n))
(x `div` p^m) `mod` p^n  =  (x `mod` p^(m+n)) `div` p^m
```
For the latter four properties we have to assume `p>0`.

4 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 for the number representation, the second the number of digits we treat. For the specification we assume that the list `zs` is the shuffle of the digit lists for the two summands, ie. that digits p (2*k) zs holds. Then we specify

```
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. We illustrate this by a diagram:

This specification does not yet provide a particular adding algorithm which could be directly taken as the description of an abstract layout. It clearly separates "what" from "how" and allows quite different implementations, such as carry-ripple and carry-lookahead adders. This will be exploited in later stages of our derivation.

4.1 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.

The derivation is driven by the recursion structure of the abstraction and representation functions. It follows a general strategy that can be partly automated, eg. by transformation tactics in ULTRA and, in the present case, does not require great amounts of intuition. This classical *unfold/fold strategy* (see e.g. [19]) consists pf the following steps:

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

The derivation follows the case analysis of `deco` and `code`.

Case `k=0`. We calculate:

```
  add p 0 []
=   {[ unfold add ]}
  code p 1 (deco p 0 [] + deco p 0 [])
=   {[ unfold deco, neutrality of 0 ]}
  code p 1 0
=   {[ unfold code ]}
  code p 0 (0 `div` p) ++ [0 `mod` p]
=   {[ arithmetic and unfold code ]}
  [] ++ [0]
=   {[ neutrality of [] ]}
  [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])
=   {[ unfold add ]}
  code p (k+2) (deco p (k+1) (xs ++ [x]) +
  deco p (k+1) (ys ++ [y]) )
=   {[ unfold deco ]}
  code p (k+2) ((deco p k xs)*p + x + (deco p k ys)*p + y )
=   {[ arithmetic ]}
  code p (k+2) ((deco p k xs + deco p k ys)*p + x + y)
=   {[ unfold code ]}
  code p (k+1) ((deco p k xs + deco p k ys)*p + x + y) `div` p)
  ++ [(deco p k xs + deco p k ys)*p + x + y)*p + x + y) `mod` p]
=   {[ by Lemma 33 ]}
  code p (k+1) (deco p k xs + deco p k ys + (x + y) `div` p)
  ++ [(x + y) `mod` p] .
```

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

4.2 Generalization

A strategy that frequently helps when direct folding is not possible is *generalization*. 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 generalized problem; this is also known as *embedding* the original problem into the generalized one. This strategy is well-known from inductive proofs: there one frequently needs to generalize 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 generalized 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 generalization strategy! In fact, this strategy can again be partly automated.

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 ++ [(x+y+c) `div` p]) ++ [(x+y+c) `mod` p]
```

We need to ensure that the expression `(x+y+c) `div` p` always yields a proper digit. The maximal values for `x` and `y` are `p-1`. In this case we have gfx+y+c = p + (p+c-2) so that the quotient by `p` is at least `1`. Since `1` is the only digit that exists in all number systems, notably the binary system, we have to guarantee that the quotient does not exceed `1`. Hence we need the additional assertion `c `below` 2`. Fortunately, this assertion is preserved as an invariant of the recursion, ie. if it holds for `c` it also holds for the new carry `(x+y+c) `div` p`. We forego a formal treatment of assertions here and refer to [15] instead.

4.3 Modularization

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

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

Of course, `fa` is the full adder function. But note again that this is introduced purely formally!

Now we may use splicing (cf. Section 2.2) to obtain

```
cadd p (k+1)   =   splice (2*k)  (cadd p k)  (fa p) .
```

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 41 `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

```
    cadd p (k+m) (zs ++ zs' ++ [c])
=     {[ unfold cadd ]}
    code p (k+m+1) (deco p (k+m) (xs ++ xs') +
                    deco p (k+m) (ys ++ ys') + c)
```

= {⟦ by Lemma 32 ⟧}

```
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)
```

= {⟦ arithmetic ⟧}

```
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
```

= {⟦ by Lemma 32 ⟧}

```
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
```

= {⟦ fold `code` ⟧}

```
code p (k+1)(deco p k xs + deco p k ys + d) ++ us
where (d:us) = codep (m+1)
                     (deco p m xs' + deco p m ys' + c)
```

= {⟦ fold `cadd` ⟧}

```
cadd p k (xs ++ ys ++ [d] ++ us)
where (d:us) = cadd p m (xs' ++ ys' ++ [c])
```

= {⟦ fold `splice` ⟧}

```
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, which may even be heterogeneous. 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 parametrization 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.

4.4 Abstraction

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

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

The idea is that, given a function `f :: Int -> Int -> [Int] -> [Int]`, the module `digrep f p k (zs ++ [c])` computes a `p`-adic representation of the value of `f` on the shuffled 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 base `p` and number `k` of digits and uses a list consisting of two "proper" arguments and a "carry". Assuming that `digits p (2*k + 1) (zs ++ [c])` holds, we specify

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

For the base case `k=0` we calculate

```
  digrep f p 0 [c]
=   {[ unfold digrep ]}
  f p 0 [deco p 0 [], deco p 0 [], c]
=   {[ unfold deco ]}
  f p 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 4.3 shows, it is more advantageous to head for a decomposition property of `digrep`. By analyzing the proof of Lemma 41 we can find a sufficient condition on `f` that makes the proof go through in general. Following [9] we call `f` factorizable if

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

holds for all natural numbers `j,k,m,n,p,q,r`. Now Lemma reflm:spliceadd generalizes to

Theorem 42 (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])
```
= {[unfold `digrep`]}
```
f p (k+m) [deco p (k+m) (xs++xs'),
deco p (k+m) (ys++ys'), c]
```
= {[unfold `deco`]}
```
f p (k+m) [(deco p k xs)*p^m + deco p m xs',
(deco p k ys)*p^m + deco p m ys', c]
```
= {[factorizability]}
```
splice 2 (f p k) (f p m) [deco p k xs, deco p k ys,
                          deco p m xs', deco p m ys', c]
```
= {[unfold `splice`]}
```
f p k [deco p k xs, deco p k ys, d] ++ us
where (d:us) = f p m [deco p m xs', deco p m ys', c]
```
= {[fold `digrep` twice]}
```
digrep f p k (zs ++ [d]) ++ us
where (d:us) = digrep f p m (zs'++[c])
```
= {[fold `splice`]}
```
splice (2*k) (digrep f p k) (digrep f p m)
(zs ++ zs' ++ [c]) .
```
□

This is in fact Hanna's Factorization Theorem (see again [9]), which gives a general scheme for correct implementations of iterative arithmetic circuits. The proof of Lemma 41 contains a section which uses Lemma 32 to show that `(*)` above defines a factorizable `f`; the remainder is isomorphic to the proof of Theorem 42.

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

Corollary 43 *For* `k > 0` *we have*

```
digrep f p k  =  foldr1 (splice 2) (copy k (f p 1)) .
```

Here we use the standard *Haskell* function `copy`. The expression `copy k x` produces a list consisting of `k` copies of `x`.

Another instance of `digrep` is a comparator circuit, described by

```
digrep f p k  where  f p k [m,n,c] =  [eq m n  /\  c] .     (**)
```

Here,

```
eq m n =  if m == n then 1 else 0  and  b /\ c =  b*c ,
```

so that we have numerical representations of the usual Boolean operations. It is straightforward to show that the function `f` in `(**)` is indeed factorizable. To obtain a comparator circuit, we have to instantiate `c` appropriately, viz. by the neutral element `1` of `/\`, and to unwind the specification using the Factorization Theorem. This results in

5 Successor (Counting)

Next we want to derive a counter circuit, ie. 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) (deco 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
=     {[ unfold succ ]}
    code p (k+1) (deco p k xs + 1)
=     {[ neutrality of 0 ]}
    code p (k+1) (deco p k xs + 0 + 1)
=     {[ fold deco ]}
    code p (k+1) (deco p k xs + deco p k (copy k 0) + 1)
=     {[ fold cadd ]}
    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

```
ha [x,c]  =  fa [x,0,c]  =  [(x+c) `div` p , (x+c) `mod` p]
```

Of course, `ha` is the half adder function. But again it has been introduced purely formally.

The simplified design looks as follows:

6 Specialization: Base 2

For `p=2` we obtain the usual representations

```
ha 2 [x,c]  =  [x /\ c,  x >< c]

fa 2 [x,y,c]  =  [ d \/ e,  z]
   where [d,u]  =  ha 2 [x,y]
         [e,z]  =  ha 2 [u,c]
```

Here, `/\, \/` and `><` are again the arithmetic representations of the Boolean operations on base 2 digits with `><` denoting exclusive or.

7 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). DA From the carry ripple design we read off the recurrence equation

```
c (i+1)     =  (p i /\ c i) \/ g i  where
(g i, p i)  =  (x i /\ y i  , x i >< y i ) .
```

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

```
c (i+1)  =
  foldr1 (\/)[ ( foldr1 (/\) [ p k | k <- [j+1..i] ) /\ g j  |
               j <- [-1..i] ]
  where g (-1)  =  c 0 .
```

Here `foldr1` is a predefined *Haskell* 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 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.

8 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.

8.1 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
-- undefined otherwise
```

The identity `id` is predefined polymorphically by `id y = y` and hence does not need to be extended 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 [] = []
```

8.2 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 first-class citizens.

```
(f |> g) xs   =   g (f xs)
```

For parallel composition we need to supply the respective operator with the number of `k` inputs to be routed to the left module; the remaining ones are routed to the right module.

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

By the definition of `take` and `drop` this works also if `k > length xs`: in that case `f` gets the full list `xs` whereas `g` only gets the empty list `[]`. Here is the diagram for `par k f g`:

Conventional polymorphism is too weak to model parallel composition more adequately. To avoid the necessity of uniformly typing all list element one would

need an extension to "tuples as first-class citizens" with concatenation of tuple types and also of tuples as primitive operations. However, this might lead to problems with automatic type checking. Therefore we have chosen the simple approach above.

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

8.3 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 [21].

Associativity:

```
f  |> (g |> h)  =  (f |> g) |> h
par (m+k) (par m f g) h   =  par m f  (par k g h)
```

Abiding Law I:

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

where `n` is the output arity of `f` and the input arity of `g`.

Neutrality:

```
id |> f  =  f  =  f |> id
par m  f  ide  =  f  =  par 0  ide  f
```

Idempotence:

```
swap |> swap  =  id
```

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

8.4 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  =
      par i  (bsink i)  (parj (bid j)  (bsink (n-j-i)))
```

for `0 <= i <= n` and `0 <= j <= (n-i)`.

We have the following fusion rule:

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

8.5 Recursions for the Bundle Operations

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

Example 81 The bundle operation `bswap m n` swaps its first `m` inputs with the following `n` ones. It is defined by the equations

```
        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 m) (bswap k (k+n))
```

□

9 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. This is analogous to the process of λ-abstraction in combinatory logic (see eg. [1]). Therefore we term this operation *combinator abstraction*. For its definition, 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.

Suppose now an expression `E` all formal parameters of which occur in the repetition free list `[x0,...,xn-1]`. Then the combinator abstraction `CA E` of `E` is defined by induction over the structure of `E`.

```
CA [xi]  =  sel n i 1
CA f     =  \xs -> [ f  (xs!!0) ... (xs!!(k-1))]
      if  f :: t0 -> ... tk-1 -> t
CA (f E1 ... Em)  =
   (CA (E1 ++  ...  ++ Em)) |>  CA f
CA (E1 ++ ... ++ Em)  =
   bfork m n |> (CA E1 |||  ...  |||  CA Em)
```

Example 91 `CA ([x /\ y] ++ [y >< x])  =`

```
bfork 2 2 |> (bfork 2 2 |> ((sel 0 1 ||| sel 1 1) |> CA /\) |||
              bfork 2 2 |> ((sel 1 1 ||| sel 0 1) |> CA ><))    ) .
```

This can, of course be simplified to

```
bfork 2 2 |> ((bid 2 |> CA /\) ||| (swap |> CA ><)) .
```

□

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`, ie. if the sublists of formal parameters are disjoint and in order.

The situation can often be improved using swaps .

Example 92 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
```

□

10 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 [10, 18].

Part II: Sequential Hardware

11 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 *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 the pseudo-value `undefined` after the "proper" finite part.

We will use partial definitions 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 (ie. enforces the assertion `t >= 0`) and on nonnegative time points agrees with `f` .

12 Networks

Again we model bundles of inputs and 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.

Our diagrams will now be drawn sideways:

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

12.1 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. We define

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

Alternatively, since streams are functions themselves, the lifting may also be expressed using function composition:

```
lift1 f [d]  =  [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)]
```

The inscriptions of the boxes follow notationally the view of infinite streams of functions used in [16].

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 *Haskell* function that produces a constant unary function from a value.

12.2 Initialized 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 stream by one time unit; at time `0` it emits `x` as the initial value:

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

We now state laws for pushing delays through larger networks. They allow for each circuit constructor to shift delay elements from the input side to the output side or vice versa under suitable change of the initialization value. These laws are used centrally in our treatment of systolic circuits.

Lemma 121 (Delay Propagation Rules) *If* `f` *is strict, ie. is undefined whenever its argument is, then*

```
(x#) |> lift1 f    =  lift1 f  |> ((f x) #)
```

If `g` *is doubly strict, ie. is undefined whenever both its argument are, then*

```
((x#) ||| (y#)) |> lift2 g  =  lift2 g  |> ((g x y)#)
```

Moreover,

```
           (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.

13 Example: The Single Pulser

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

13.1 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 characterize 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 ] , ie.
pulser [d]  =  [ startPulse d ]
```

13.2 Derivation of a Pulser Circuit

For `t = 0` we calculate

```
    startPulse d 0
=     {[ unfold startPulse ]}
    d 0 && ( 0==0 || not (d (0-1) )
=     {[ Boolean algebra ]}
    d 0 .
```

For `t > 0` we have

```
    startPulse d t
=     {[ unfold startPulse ]}
    d t && ( t==0 || not (d (t-1) )
=     {[ t > 0 and Boolean algebra ]}
    d t && not (d (t-1))
=     {[ fold # ]}
    d t && not ((x # d) t) .
```

for arbitrary `x`. Now we try to choose the initialization value `x` such that

```
startPulse d t  =  d t  &&  not ((x # d) 0)
```

holds also for `t=0`, ie.

```
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 (&&) .
```

14 Feedback

For describing systems with memory we need another essential ingredient. It is the very general concept of feeding back some outputs of a module as inputs again. This allows, in particular, the preservation of a value for an arbitrary period, ie. storing of values.

14.1 The Feedback Operation

Given a module `f :: [a] -> [a]` the module `feed k f` results from `f` by feeding back the last `k` outputs to the last `k` inputs:

```
feed ::  Int  -> ([a] -> [a]) -> ([a] -> [a])

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 *Haskell*.

14.2 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 [21].

Stretching wires:

```
f  |>  feed k g  |>  h   =   feed k ((f ||| id) |>  g  |>
                             (h ||| id) )
```

Abiding law II:

```
feed k f  |||  g   =   feed k (f ||| g)
```

Shifting a module:

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

15 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 *Haskell* semantics.

Lemma 151 *Interconnection is associative in the following sense:*

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

Moreover,, `connect` *has the identity* `id` *as its neutral element.*

Two interesting special cases are

```
f =||= g  =  connect 1 1 1 f g
```

and

```
f =| g  =  connect 1 1 0 f g
```

The symbols have been chosen such that they indicate the places of the external wires of the resulting circuits: whereas `f =||= g` has external wires on the left and on the right, `f =| g` has them only on the left. The operator `=||=` is also known as mutual feedback $\otimes$ (see eg. [2]). 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)    )
```

With the help of this connection, the proof of the above lemma can be given using purely the laws of network algebra. Hence the lemma is valid for all models of network algebra, not just our particular one.

16 A Convolver

We want to tackle a somewhat 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. Mathematically, the convolution is defined as

$$\sum_{i=1}^{n} w_{n-i} * d(t-i) \ ,$$

where d is the input stream and the w_j are the weights. Convolution is used in digital filters.

16.1 Specification

Using list comprehension, the above mathematical definition can be directly transcribed into a *Haskell* specification. For convenience we collect the weights also into a stream `w`. Then the convolver is specified by

```
conv :: Stream Int -> Int -> [Stream Int] -> [Stream Int]
conv w n d  =
   [ \t -> if t < n then undefined
                    else sum [ w (n-i) * d (t-i) | i <- [1..n] ] ]
```

It should be clear that the problem generalizes 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.

16.2 About Error Handling

We have not used `nonneg` here but rather modeled non-initialization by the pseudo-value `undefined`. However, the only essential assumption about `undefined` is the strictness property `x + undefined = undefined`. This could also be achieved by introducing an additional error element using *Haskell*'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
```

Similarly definitions would be given for the other arithmetic operations. Since this is somewhat cumbersome, though, we have chosen the above method.

16.3 Derivation of a Convolver Circuit

We now want to derive from the formal specification a regular layout described, as in the case of the adder, by a recursion. The obvious parameter to drive the recursion is the number n of terms in the sum, since the summation function is defined recursively itself and we can carry over its recursion structure to the convolver circuit.

The base case is `n = 0`. For `t >= 0` and `[e] = conv w 0 d` we calculate

```
   e t
=    {[ specification of e ]}
   sum [ w (0-i) * d (t-i) | i <- [1..0] ]
=    {[ definition of intervals ]}
   sum [ w (0-i) * d (t-i) | i <- [] ]
=    {[ definition of list comprehension ]}
   sum []
=    {[ definition of sum ]}
   0
```

Hence `conv 0 = cnst 0` with `cnst` defined as in Section 12.1.

We now perform the induction step. For `t >= n+1` and `[e] = conv w (n+1) d` we obtain

```
   e t
=    {[ specification of e ]}
   sum [ w (n+1-i) * d (t-i) | i <- [1..n+1] ]
=    {[ spliting the interval, definition of sum ]}
   w n * d (t-1) + sum [ w (n+1-i) * d (t-i) | i <- [2..n+1] ]
=    {[ index transformation ]}
   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] ]
=    {[ fold conv ]}
```

```
  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]  =
   [undefined # lift2(+) (lift1 ((w k) *) [li], [ri]),  li]
```

Here the expression `((w k) *)` is a so-called *operator section* of *Haskell*. It denotes the *residual function* that remains when the operator `*` is supplied with a fixed first argument while the second one still kept variable. Using λ-notation we have the equivalence

```
((w k) *)  =  \y -> (w k) * y
```

Although the parentheses in the right hand side expression are not required by the *Haskell* syntax we have added them for clarity.

Our recursive formation law for the basic convolver can be depicted as follows:

16.4 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

16.5 Towards a Systolic Version

A circuit is combinational if it uses only lifted operations and sequential or parallel composition. In a clocked circuit, the clock period is determined by the stabilization time of the circuit which depends on its longest combinational path.

In systolic circuits one tries to minimize the clock period by making the combinational modules involved quite small. Then the clock period can be kept relatively short, namely it can be taken as the maximum of the stabilization times of the combinational submodules involved. Since there is, however, no general rule for calling a combinational module "small" the precise definition of systolism avoids sich a notion. Rather, a circuit is called *systolic* (cf. [13, 14, 7])

if there is at least one delay element along every connection wire between any of its combinational modules. A related but somwhat different notion of systolism is used in the field of massively parallel systems; however, there no explicit delay elements are employed.

We want to obtain a systolic version of our convolver. Hence we have to introduce additional delay elements.

17 Speedup by Slowdown

The technique to introduce delays formally is slowdown (see e.g. [13, 14, 12]). 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.

17.1 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 with `0` again. Eg. `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 171 *We have*

```
(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)
```

Provided that `length ss >= k`, we have

```
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 .
```

17.2 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 172 *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
=     {[ definition of `div` and `mod` ]}
    slow k f s (k*(t `div` k) + t `mod` k)
=     {[ by (*) ]}
    split k (t `mod` k) (slow k f s) (t `div` k)
=     {[ specification of slow ]}
    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) .
```

17.3 Slowdown Algebra

The function `slow` distributes nicely through our circuit building operators, as stated by the following

Lemma 173
```
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 (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)
```

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

Lemma 174 *Suppose that* `(x#) |> f = f |> (y#)`. *Then also*

```
(x#) |> slow k f  = (slow k f) |> (y#) .
```

18 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 is usually 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 by Lemma 174. 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]  =
   [undefined # lift2 (+) (lift1 ((w k) *)[bli], [ri]), bli]
  where bli  =  undefined # 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 [17].

comparison with verif approaches

19 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 is 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 -> s
pipe xs j q   =  run  xs  (exe j q)
```

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

```
run []  q  =  q
   --  pipeline exhausted
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 .
```

To derive the recursive case we need the central assumption for the correctness of the version of pipelining we treat here. We stipulate that execution of an instruction does not change the contents of the program memory. This means that the program has to be kept in a part of the memory that is administered in a read-only fashion. This assumption can be expressed formally as

```
fetch a (exe j q)  =  fetch a q          (*)
```

for all `a,j,q`. With this assumption we calculate

```
  pipe (x : xs) j q
=   {[ unfold pipe ]}
  run (x : xs) (exe j q)
=   {[ unfold run ]}
  run xs (exe (fetch x q') q')
  where q' = exe j q
=   {[ by (*) ]}
  run xs (exe (fetch x q) q')
  where q' = exe j q
=   {[ fold pipe ]}
  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 validity of equation `(*)`. In particular, the transformation can be iterated to obtain pipelines with several stages if `exe` can be decomposed into further subfunctions.

reference to Börger

20 Summary

expand

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

- algebraic reasoning,
- parameterization,
- modularization,
- re-use of designs and derivations,
- precise determination of initialization values.

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 F.K. Hanna, J. Philipps, P. Scholz and G. Ştefănescu.

References

1. H.P. Barendregt: Functional programming and lambda calculus. In: J. van Leeuwen (ed.): Handbook of theoretical computer science. Vol. B: Formal models and semantics. Amsterdam: Elsevier 1990, 321–363

2. M. Broy, G. Ştefănescu: The algebra of stream processing functions. Institut für Informatik, TU München, Report TUM-I9620, 1996
3. 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. Lecture Notes in Computer Science **292**. Berlin: Springer 1987
4. 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)
5. C. Delgado Kloos: Semantics of digital circuits. Lecture Notes in Computer Science **285**. Berlin: Springer 1987
6. 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
7. S. Even, A. Litman: On the capabilities of systolic systems. Math. Sytems Theory 27, 3-28 (1994)
8. 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
9. K. Hanna, N. Daeche, M. Longley: Specification and verification using dependent types. IEEE Trans. Softw. Eng. 16:9, 949-964 (1990)
10. 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)
11. IFIP 94/97: IFIP WG 10.5 Verification Benchmarks. Reachable via internet under `http://goethe.ira.uka.de/hvg/benchmarks.html`
12. G. Jones, M. Sheeran: Circuit design in Ruby. In: J. Staunstrup (ed.): Formal methods for VLSI design. Elsevier 1990, 13-70
13. C.E. Leiserson, J.B. Saxe: Optimizing synchronous systems. J. VLSI and Computer Systems 1, 41-68 (1983)
14. C.E. Leiserson, J.B. Saxe: Retiming synchronous circuitry. Algorithmica 6, 5-35 (1991)
15. 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. Lecture Notes in Computer Science **1074**. Berlin: Springer 1996, 163-184
16. B. Möller: Ideal stream algebra. In: B. Möller, J.V. Tucker (eds.): Prospects for hardware foundations. Lecture Notes in Computer Science . Berlin: Springer (this volume)
17. B. Möller: An algebraic approach to systolic circuits. Institut für Informatik der Universität Augsburg, Report 1998-01, January 1998. Also in: B. Möller, M. Sheeran (eds.): Proceedings of the Workshop FTH '98 — Formal Techniques for Hardware and Hardware-Like Systems, Marstrand, June 19, 1998. Chalmers institute of Technology 1998, pages
18. P. Molitor: A survey on wiring. J. Inf. Process. Cybern. EIK 27, 3-19 (1991)
19. H.A. Partsch: Specification and transformation of programs - A formal approach to software development. Berlin: Springer 1990
20. 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

21. G. Ştefănescu: Algebra of flownomials. Institut für Informatik der TU München, Report TUM-I9437, 1994

Appendix: Essential Constructs of *Haskell*

20.1 Basic Types and Functions

For those not familiar with *Haskell*, we briefly repeat its essential elements. Basic types are `Int` for the integers and `Bool` for the Booleans with elements `True` and `False`. The operations of conjunction and disjunction are denoted by `&&` and `||`, resp. These are the semi-strict versions evaluating their arguments from left to right, ie. satisfying

```
x && y  =  if x then y    else False
x || y  =  if x then True else y
```

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 x1 x2 ... xn`. In this case `f` has the higher-order type `f :: t1 -> (t2 -> ... (tn -> t) ...)` or, abbreviated, `f :: t1 -> t2 -> ... tn -> 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 $\lambda x.E$ 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 .
```

20.2 Case Distinction and Assertions

Haskell 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 `Ei` for which the corresponding `Ci` 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 [15]).

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 201 By the equations

```
f 0 = 5
f 1 = 7
```

the function `f :: Int -> Int` is defined only for argument values `0` and `1`. □

20.3 Lists

The type of lists of elements of type `a` is denoted by `[a]` . The list consisting of elements `x1,...,xn` is written as `[x1,...,xn]` ; in particular, `[]` is the empty list. Concatenation is denoted by `++` . Prefixing an element to a list is denoted by the colon operator:

```
x:xs = [x] ++ xs .
```

The function `length` returns the length of a list. The `i`th element of list `xs` is selected by the expression `xs!!i` (where numbering starts with `0`).

A list may be split into two parts using the functions

```
take, drop :: [a] -> Int -> [a] .
```

For non-negative integer `k` the list `take k xs` consists of the first `k` elements of `xs` if `k <= length xs` and of all of `xs` if `k > length xs` . For negative `k` the expression `take k xs` is undefined. The list `drop k xs` results by removing `take k xs` from the front of `xs`. Hence one always has

```
take k xs ++ drop k xs = xs .
```

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 $\{fx|x \in S, px\}$. The meaning of the list comprehension expression `[ f x | x <- L, p x]` 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 `f :: :`

```
foldr1 f [x1,...,xn]  =  f x1 (f x2 ... (f  xn-1 xn)...) .
```

Eg. `foldr1 (+) s` computes the sum of all elements of `s` . The function `foldr1` itself has the type `(a -> a -> a) -> [a] -> a` .

A variant of `foldr1` that also copes with empty lists is `foldr` ; it uses an additional argument `e` that specifies the value for empty lists. The defining equations read

```
foldr f e []  =  e
foldr f e [x] ++ xs = f x (foldr f e xs)
```

Based on `foldr` one can define a universal quantifier over lists. For a predicate `p :: a -> Bool` one has

```
all p xs  =  foldr (&&) True [p x | x <- xs ] .
```

So `all p xs` is `True` iff `p x` is `True` for all `x` in `xs` .

type classes?

laziness, comparison with other f'nal lang's