Appears in *Concurrent Computations: Algorithms, Architectures and Technology*,
ed. S.K. Tewksbury, B.W. Dickinson, S.C. Schwartz, Plenum Press, New York 1988

Timeless Truths about Sequential Circuits

Geraint Jones
Oxford University Computing Laboratory
11 Keble Road
Oxford OX1 3QD, UK

Mary Sheeran
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, UK

30 September 1987

Abstract

We suggest the use of a declarative programming language to design and describe circuits, concentrating on the use of higher-order functions to structure and simplify designs. In order to describe sequential circuits, we use a language, μFP, which abstracts from temporal iteration. The practicalities of VLSI design make regularity attractive, and we describe the use of familiar higher order functions to capture spatial iteration.

By reasoning about circuits rather than signals (programs rather than data) one abstracts from the sequential nature of a circuit. By reasoning about forms of circuit (higher order functions) one can devise implementation strategies for whole classes of algorithms. Reasoning about μFP is formally quite similar to reasoning about FP.

In this paper we identify the semantic content of the formal similarity between FP and μFP. This makes it possible to carry over from conventional functional programming those intuitions we have about algorithm design. It also makes it possible to conduct parts of a design in the simpler world of static calculations, with confidence in the correctness of the corresponding sequential circuit.

Concurrent systems

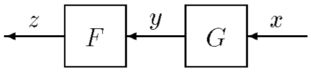
We begin from the premiss that designing a logic circuit is essentially the same activity as writing a program, but that the constraints on the programmer are in this case rather unusual. The efficient use of VLSI resources requires the designer to write a highly concurrent program, consisting of a very large number of simple but simultaneously operating processes which co-operate to achieve the desired effect. Economic considerations dictate that many of these processes have to be identical, or of a very few types. The interaction between these processes has, moreover, to be achievable within a limited network of communication links that can be laid out with short wires in a few layers on a flat surface.

Although the behaviour of the whole circuit is certainly *caused* by the actions of the individual components, it is difficult to understand an explanation of an array of processes which begins from a description of the possible actions of the components. Bottom-up design of a circuit in a state-based language which gives the programmer complete freedom to organise synchronization is correspondingly difficult.

The natural description of the function of an array circuit is in terms of the data objects which it manipulates, although these may be spread out in space over the array, and in time over the duration of the calculation[9]. We advocate a design method which elaborates a high-level specification of the data manipulation to be achieved by the whole circuit, into an implementation which is constrained by a predetermined framework of synchronization and communication routes[7, 6].

Combinational circuits

We represent a circuit in the first instance by its behaviour — that function which its output signal is of the signal at its input. The behaviour of a compound circuit can be induced from the behaviour of its components, and the way that they are connected. Thus if the input of an F circuit is driven by the output of a G circuit, they behave together as the composition $F \circ G$



$$z = Fy = F(Gx) = (F \circ G)x$$

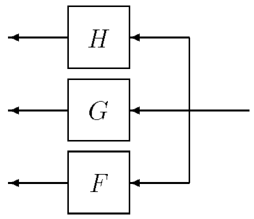
There is a hierarchy: zeroth order objects, signals like x, y, z; first order functions, circuit behaviours like F and G; and higher order functions (functions of functions), like composition.

Since we will be interested in laying out our circuits, we note that there is an alternative interpretation of this functional programming langauge. The base objects are wires, first order functions are circuits, and higher-order functions are circuit structures. This interpretation gives a formula for constructing a circuit which (if the interpretation is *sound*) will implement the behavioural interpretation. If a function is applied to a named argument, then the circuit has its input connected to the corresponding wire, and so on.

If wires can be named, and those names mentioned at will in a circuit description, then there is no guarantee that the circuit has a simple layout. There is certainly no simple and systematic way of laying out such circuits. Notice however, that we can describe the composition $F \circ G$ without needing to name wires, and that we can, given layouts of F and G, combine them systematically to lay out the composition.

To preserve this systematic laying out, we use Backus' FP[1] as a circuit description language, and we do not name signals and wires. Our reasoning will all be about the application of structures (higher order functions) to circuits (first order functions). We will need a repertoire of higher order functions adequate to describe interesting circuit layouts, but we will then confine ourselves to just those circuits which can be *constructed* by these structures.

Buses and collections of signals are produced by a constructed circuit, in which each component circuit generates the corresponding component of the output signal.



$[F, G, H]$

Since tuples may contain tuples, collections of signals can be given any required structure. A circuit with several input wires will be modelled by a function of a tuple, and a circuit with several output wires by a function returning a tuple.

Tuples can be analysed and synthesised by left- and right-handed constructors, app_L and app_R and their inverses:

$$\begin{aligned}
\mathsf{app}_L \circ [p, [q, r, \ldots]] &= [p, q, r, \ldots] \\
\mathsf{app}_R \circ [[\ldots, p, q], r] &= [\ldots, p, q, r] \\
\mathsf{app}_L^{-1} \circ [p, q, r, \ldots] &= [p, [q, r, \ldots]] \\
\mathsf{app}_R^{-1} \circ [\ldots, p, q, r] &= [[\ldots, p, q], r]
\end{aligned}$$

and because pairs are so important we define the following higher order abbreviations

$$\begin{aligned}
(F \parallel G) \circ [a, b] &\stackrel{\text{def}}{=} [F \circ a, G \circ b] \\
\mathsf{fst}\ F &\stackrel{\text{def}}{=} (F \parallel \mathsf{id}) \\
\mathsf{snd}\ F &\stackrel{\text{def}}{=} (\mathsf{id} \parallel F)
\end{aligned}$$

where id is the identity function, $\mathsf{id} \circ a = a$.

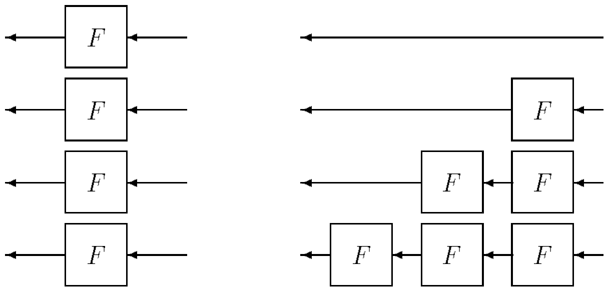
We will have loops in the language which allow us to duplicate a circuit over a tuple, either by *mapping*

$$\begin{aligned}
\propto F \circ [\,] &\stackrel{\text{def}}{=} [\,] \\
\propto F \circ \mathsf{app}_L &\stackrel{\text{def}}{=} \mathsf{app}_L \circ (F \parallel \propto F)
\end{aligned}$$

or by making a *right triangle*

$$\begin{aligned}
\triangle_R F \circ [\,] &\stackrel{\text{def}}{=} [\,] \\
\triangle_R F \circ \mathsf{app}_L &\stackrel{\text{def}}{=} \mathsf{app}_L \circ \mathsf{snd}\ (\propto F \circ \triangle_R F)
\end{aligned}$$

or, of course, the corresponding left handed construct.



$\propto F$ $\triangle_R F$

Getting those signals which are to interact to meet is a crucial part of the design of a circuit. We introduce left- and right-handed plumbing functions to *distribute* a signal across a bus

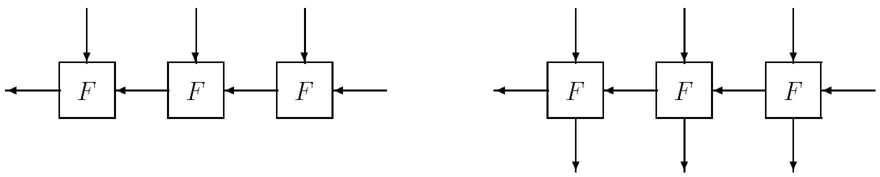
$$\mathsf{dist}_L \circ [p, [q, r, \ldots]] \stackrel{\text{def}}{=} [[p, q], [p, r], \ldots]$$
$$\mathsf{dist}_R \circ [[\ldots, p, q], r] \stackrel{\text{def}}{=} [\ldots, [p, r], [q, r]]$$

so that, for example, $\propto add \circ \mathsf{dist}_L \circ [a, [b, c, d]]$ adds the output of a independently to the outputs of each of b, c, d. We will also find the need of a generic interleaving circuit, zip,

$$\mathsf{zip} \circ [[a_0, b_0, c_0, \ldots], [a_1, b_1, c_1, \ldots], [a_2, b_2, c_2, \ldots], \ldots] \stackrel{\text{def}}{=} [[a_0, a_1, a_2, \ldots], [b_0, b_1, b_2, \ldots], [c_0, c_1, c_2, \ldots], \ldots]$$

Interleaving arises naturally in the process of elaborating a word-level design to bit level. Since a large interleaving is prohibitively expensive, occupying an area quadratic in the number of signals through it, it will be in the designer's interests to eliminate large interleavings from his circuit.

Ideally, for minimum communciation cost, most of the interactions in a circuit will be between adjacent circuit elements. Horizontal and vertical forms of *reduction* implement continued sums, and products, and similar forms of loop. For example $/_H\, add \circ [[x_0, x_1, x_2, \ldots], 0] = \sum x_i$, where 0 is the function that produces a constant zero output. Horizontal and vertical *arrays* are generalisations of reduce that yield partial results as outputs.



$/_H F$ $\backslash_H F$

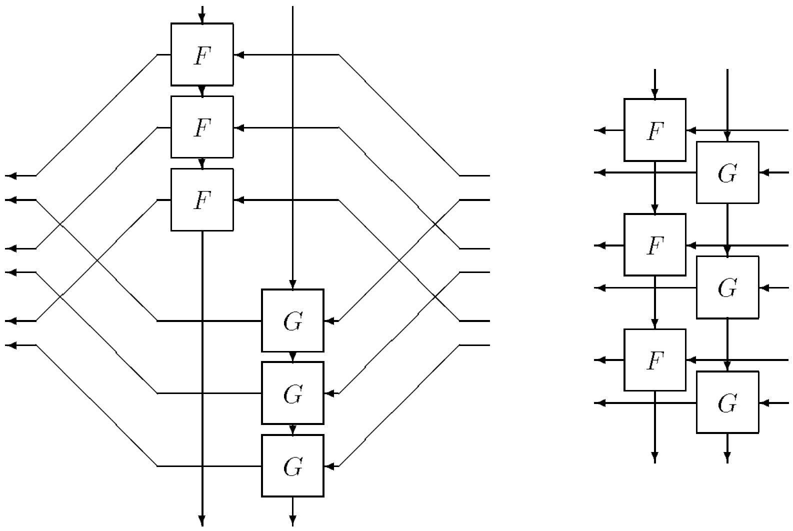
Equational reasoning

The equations defining various constructs, and indeed any other equations relating our FP expressions, are statements about the behavioural semantics of the language. That F is equal to G means that given the same input signal each will produce the same output. In other ways, F and G may be different; they may, for example, have radically different interpretations under the layout semantics. Consider $\mathsf{zip} \circ \mathsf{zip} = \mathsf{id}$ which states that the effect of two consecutive interleaving circuits can be achieved by straight wiring; the former has a greater area and complexity than the latter. F may be a natural specification of a desired signal transformation, but its natural layout may be prohibitively expensive; it may be harder to see immediately what transformation is implemented by G, whereas it has a simpler and smaller layout. In that case, their equality guarantees that G can safely be used efficiently to implement the specification F.

Consider the development of a circuit which performs some arithmetic on integers. Its specification will, initially, be given in terms of integer signals. In the course of implementation the integers will have to be represented by a number of bits, and the circuits that process integers elaborated into circuits that process those arrays of bits. Addition of integers, for example, can be implemented by an array of full-adders, *fadd*, taking as input the interleaving of the representations of the integers,

$$\begin{aligned}[sum, c_{out}] &= \backslash_V fadd \circ \mathsf{snd\ zip} \circ [c_{in}, [x, y]] \\ &\quad \text{where } fadd \circ [c_{in}, [x, y]] = [s, c_{out}]\end{aligned}$$

Circuits in which there are several adders would naturally be described in terms of arrays of full-adders and expensive interleavings; the equality

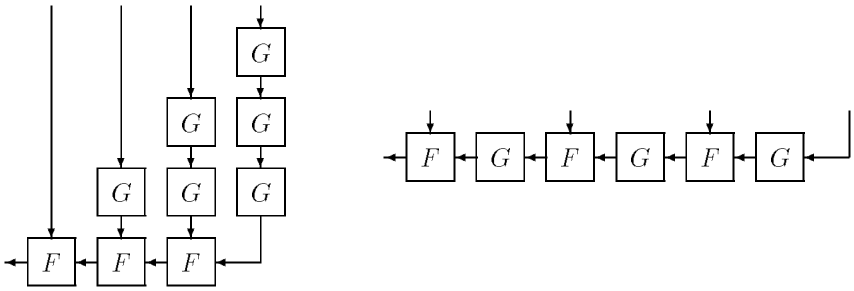


$$\mathsf{fst\ zip} \circ (\backslash_V F \parallel \backslash_V G) \circ \mathsf{snd\ zip} \quad = \quad \backslash_V (\mathsf{zip} \circ (F \parallel G) \circ \mathsf{zip})$$

permits the transformation of such a description into the familiar, if less obvious, interleaved data-path structure which occupies considerably less area, has shorter wires, and fewer wire-crossings. Notice that this equation is a statement about any circuits F and G, and can be used to improve the layout of any circuit of the given form involving large interleavings and arrays, irrespective of the function which it implements, and of the widths of the arrays.

There are also useful observations which can be made about circuit structures, conditional on properties of the components, for example

If	$F \circ \propto G$	$=$	$G \circ F$
then	$(/_H F) \circ \mathsf{app}_R^{-1} \circ \triangle_R G$	$=$	$/_H (F \circ \mathsf{snd}\ G) \circ \mathsf{app}_R^{-1}$



Again, this theorem can be used to reduce the size and component count of a circuit by eliminating triangles, provided that the components satisfy its hypothesis. This theorem can be proved, like the first, by induction on the sizes of the loops. Given a sufficiently rich collection of such theorems, a circuit development can proceed by successive use of the equations as rewriting rules.

Sequential circuits

In what has gone before, we have been talking of combinational circuits and signals which have been constant in time. We will now develop the framework for discussing *synchronous* circuits. In such a circuit, each signal takes on a sequence of values, and there is some mechanism to ensure that component circuits which share a common signal agree about the times at which the signal is valid.

A synchronous circuit can be thought of as having a single global clock, which runs slowly enough to allow the circuit to settle on each clock cycle. The sequence of values representing a signal is the sequence of stable values of the signal at the ends of consecutive clock cycles. Notice, however, that we have abstracted from the mechanism that achieves synchronization, so the implementation could equally well be self-timed, or partly clocked and partly synchronized by handshaking.

We will use another functional programming language, μFP, superficially similar to FP, to describe synchronous circuits. (The name derives from a μ constructor introduced in [5] for constructing state-machines from state-transition functions.) In this language, the base objects are infinite sequences of signal values; the first order functions represent clocked circuits; and the higher order functions will again be circuit structures.

We will explain the meaning of μFP by giving a translation into FP. Where confusion might arise in what follows, μFP appears between double roman brackets, and the text outside the brackets is FP.

For a function F that represents a combinational circuit in FP, we will use the same name in μFP to stand for the effect of providing the (stateless) circuit with a sequence of values.

$$[\![F]\!] \stackrel{\text{def}}{=} \propto F$$

F and $[\![F]\!]$ are corresponding functions. We would draw the same circuit diagram for each of them. F maps a value to a value whereas $[\![F]\!]$ maps an infinite sequence of values to an infinite sequence of values. The definition says that $[\![F]\!]$ applies the function F to each element of the input sequence. If *not* is the combinational circuit which negates its input, then $[\![not]\!]$ is the sequential circuit which negates each successive element of its input sequence. If *add* is the combinational circuit which adds two integer signals, then $[\![add]\!]$ is the sequential circuit which adds each successive pair of integers from an infinite sequence of pairs of integers.

The meaning of higher order functions can be given by a systematic translation, which will be made precise later. The important thing is that the meaning of a μFP function should be given in terms of the corresponding FP function. Consider, for example, the $\triangle_R$ constructor. The input sequence of tuples is separated (using zip) into the tuple of sequences which appear on the wires which are the inputs to the triangle. On the output of the triangle, the tuple of sequences is recombined (using zip^{-1} into a sequence of tuples.

$$\begin{aligned} [\![\triangle_R F]\!] &\stackrel{\text{def}}{=} \mathsf{zip}^{-1} \circ \triangle_R [\![F]\!] \circ \mathsf{zip} \\ &= \mathsf{zip} \circ \triangle_R [\![F]\!] \circ \mathsf{zip} \end{aligned}$$

Proceeding in the same way, we can define constructs in μFP which correspond to each of the constructs of FP. This language describes the way that combinational circuits can be used to process sequences of inputs. In order to describe non-combinational circuits, we will need a non-combinational primitive, $\mathcal{D}$, a *latch* which delays its input by a single time step. The fact that we use doubly infinite sequences allows us freely to manipulate *anti-delays*, $\mathcal{D}^{-1}$, that is, circuits which predict their inputs. The output of an anti-delay at any time is the value which will appear at its input at the subsequent time. An anti-latch shifts each value on a time-sequence forward by one time step. Latches and anti-latches can be cancelled, because $\mathcal{D} \circ \mathcal{D}^{-1} = \mathcal{D}^{-1} \circ \mathcal{D} = \mathsf{id}$. Latches and anti-latches apply to all types of signal, so

$$\mathcal{D} \circ \mathsf{app}_L = \mathsf{app}_L \circ (\mathcal{D} \parallel \propto\mathcal{D}) = \propto\mathcal{D} \circ \mathsf{app}_L$$

Of course, whilst one can implement latches, anti-latches cannot be given a sound physical realisation. They can be used to record requirements on the times at which data are made available to a circuit, and they can be manipulated in the same way as latches in the course of a development, provided that they are eliminated before the implementation is complete. Triangles of latches and anti-latches appear naturally in describing data-skew.

Duality

The choice of the same notation for μFP as for FP was not made deliberately to cause confusion. The definitions of μFP functions are so chosen that equalities such as

$$\mathsf{fst\ zip} \circ (\backslash_V F \parallel \backslash_V G) \circ \mathsf{snd\ zip} = \backslash_V (\mathsf{zip} \circ (F \parallel G) \circ \mathsf{zip})$$

are true not only in FP but also in μFP. Moreover, although $\mathcal{D}$ had no equivalent in FP, these equalities continue to be true when applied to sequential circuits, for example

$$F \circ \propto\mathcal{D} = \mathcal{D} \circ F \;\Rightarrow\; (/_H F) \circ \mathsf{app}_R^{-1} \circ \triangle_R \mathcal{D} = /_H (F \circ \mathsf{snd}\ \mathcal{D}) \circ \mathsf{app}_R^{-1}$$

This last (an instance of what [4] calls a 'retiming' lemma) is particularly useful, giving a way of transforming a combinational ripple-through circuit, $/_H F$, and an expensive data-skewing circuit, $\triangle_R \mathcal{D}$, into a pipelined circuit $/_H(F \circ \mathsf{snd}\ \mathcal{D})$.

It is, of course, very much simpler to prove these equalities to be true in FP than it is to check the corresponding theorems in μFP. Were it possible to say that any equality true in FP applied also in μFP, performing the proofs by induction in the simple combinational world of FP would be an easy way of checking powerful theorems about sequential circuits. Unfortunately, this is not the case.

Consider the conditional (multiplexer) function, cond, defined in FP by

$$\mathsf{cond} \circ [p, [f, g]] = \begin{cases} f & \text{if } p \text{ returns true} \\ g & \text{if } p \text{ returns false} \end{cases}$$

Since it is stateless, the natural definition of cond in μFP would seem to be $[\![\mathsf{cond}]\!] = \propto\mathsf{cond}$. It is readily shown that in FP

$$h \circ \mathsf{cond} \circ [p, [f, g]] = \mathsf{cond} \circ [p, [h \circ f, h \circ g]]$$

by an argument by cases on the value of the output of p. This theorem is not true in μFP for all h: indeed, it is only necessarily true in case h is stateless, in particular

$$\mathcal{D} \circ \mathsf{cond} \circ [p, [f, g]] \neq \mathsf{cond} \circ [p, [\mathcal{D} \circ f, \mathcal{D} \circ g]]$$

To rescue the situation, it will be necessary to make precise the 'systematic translation' of μFP into FP. We will divide μFP constants into two classes, such that a theorem true in FP will necessarily be true in μFP provided that only *timeless* constants appear in its statement. It will transpire that cond is not timeless.

Types

In order to be precise about the translation of μFP into FP, we will need to assign types to FP expressions. We will assume that there exists some set of base types (integers, Booleans, etc.) The exact nature of base types is unimportant, since we are principally concerned with the size and structure of tuples. We will return, later, to the question of what a type is; for the present we will simply record that $x : T$ if x is of type T.

What is interesting about the type of an object is a record of the the way the object is constructed. For any types $T, U, V, \ldots$ there is a product type $[T, U, V, \ldots]$ for which

$$x : T, y : U, z : V, \ldots \;\Rightarrow\; [x, y, z, \ldots] : [T, U, V, \ldots]$$

and there is a type which is the type of all (finite and infinite) homogeneous tuples with components of a given type

$$x_i : T \;\Rightarrow\; [x_0, x_1, x_2, \ldots] : T^\star$$

These type constructors are related, because

$$x : [T, T, T] \;\Rightarrow\; x : T^\star$$

All zeroth order objects in FP can be given types constructed in these ways.

The type of a function which takes inputs of type A and delivers a result of type B will be denoted $A \rightarrow B$. Thus, function application can be typed by the rule

$$f : A \rightarrow B, x : A \;\Rightarrow\; fx : B$$

Type variables bound by the universal quantifier will be used to indicate polymorphic functions in the usual way. For example,

$$\mathsf{id} : \forall\alpha.\alpha \rightarrow \alpha$$

$$\mathsf{dist}_L : \forall\alpha, \beta.[\alpha, \beta^\star] \rightarrow [\alpha, \beta]^\star$$

A polymorphic function can also have any substitution instance of its type. In general, an object has many types, for example

$$\mathsf{id} : \forall\alpha.\alpha \rightarrow \alpha$$

$$\mathsf{id} : \forall\beta.(\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)$$

$$\mathsf{id} : \forall\alpha, \beta.[\alpha, \beta]^\star \rightarrow [\alpha, \beta]^\star.$$

Note that an application may also return a polymorphic function. For example, id applied to id is again id, the polymorphic identity function.

An assignment of types to an expression is made by choosing instances of types of the constants and variables, and ascribing types to each of its sub-expressions, so that each application is exactly well typed. An assignment of types to an equation between two expressions consists of assignments of the same types to each expression.

Timelessness

Just as the same symbols and names are used to stand for corresponding objects in FP and μFP, so we will use the same names for their types. Thus, if $f : A \to B$ in FP has a corresponding meaning in μFP we will write also $[\![f : A \to B]\!]$.

For example, the FP function *first*, which selects the first of a pair of inputs, has type $\forall\alpha,\beta.[\alpha,\beta] \to \alpha$. We will write the type of the μFP function $[\![\mathit{first}]\!]$, which takes a *stream* of pairs to the *stream* of the first elements of those pairs, in exactly the same way: $[\![\mathit{first} : \forall\alpha,\beta.[\alpha,\beta] \to \alpha]\!]$. The streams are implicit in the type. We call *first* timeless, because we can express the meaning (i.e. the translation into FP) of the μFP function in terms of the FP function itself.

$$[\![\mathit{first}]\!] = \mathit{first} \circ \mathsf{zip}$$

We simply zip the stream of pairs together, to get a pair of streams. The first stream contains the stream of first elements, so we select it using the FP function *first*. Similarly, we can express other μFP plumbing functions and combining forms in terms of their FP equivalents. This allows us to develop laws about μFP by appealing to the corresponding FP laws. This was the strategy used in [5]. Here, we generalise the approach by defining an injection that relates the meaning of an FP expression to the meaning of the corresponding μFP expression. An object or function will be injected according to its type.

We define a family of functions, π, indexed by our range of monotypes, such that for any timeless object x

$$x : T \;\Rightarrow\; [\![x]\!] = \pi_T x$$

Indeed, this will be the definition of timelessness: an object x of type T (strictly speaking, a pair of objects, one in FP, one in μFP) is timeless precisely when the two meanings are related by the injection π_T. We will say that a polymorphic object is timeless if each of its instances is timeless. Thus, a polymorphic function f is timeless iff

$$f : \forall\alpha.T(\alpha) \;\Rightarrow\; \forall\alpha.([\![f]\!] = \pi_{T(\alpha)} f)$$

In fact, because of our choice of polymorphic primitive functions, it is always the case that $\pi_{T(\alpha)} f$ is independent of α. So, $\pi_{T(\alpha)}$ can be used for $\pi_{\forall\alpha.T(\alpha)}$.

π, being an injection, has an inverse, such that $\pi_T^{-1} \circ \pi_T$ is the identity function on the domain of π_T and $\pi_T \circ \pi_T^{-1}$ is the identity function on the range of π_T. At this point we can be more precise about what a type is: we identify a type with its injection function and call types equal if they have the same injections, distinct if they have distinct injections.

To complete the definition of π, we must define it for each of our type constructors. The injection for a pair type is constructed from the injections for the components. First, the components of the pair of streams are injected, and then the resulting pair is transposed, to produce a stream of pairs.

$$\pi_{[A,B]} = \mathsf{zip} \circ (\pi_A \parallel \pi_B)$$

and similarly for larger tuples. In particular, the injection for a homogeneous tuple is

$$\pi_{A^\star} = \mathsf{zip} \circ \propto\pi_A$$

Because π is an injection, we can conclude that

$$\pi^{-1}_{[A,B]} = (\pi^{-1}_A \parallel \pi^{-1}_B) \circ \mathsf{zip}$$

$$\pi^{-1}_{A^\star} = \propto\pi^{-1}_A \circ \mathsf{zip}$$

Thus we define the injection for zeroth order objects.

The injection for the type of a function is chosen to make the following diagram commute for timeless f:

FP

$$\begin{tikzcd} \cdot \arrow[r, "f : A \rightarrow B"] \arrow[d, "\pi_A"'] & \cdot \arrow[d, "\pi_B"] \\ \cdot \arrow[r, "{[\![f : A \rightarrow B]\!]}"'] & \cdot \end{tikzcd} \qquad \pi_{A \rightarrow B} f = \pi_B \circ f \circ \pi^{-1}_A$$

meaning of μFP in FP

The force of the commutativity of this diagram is that rather than computing the result $[\![f]\!][\![x]\!]$ of applying the μFP function f to x, applying the corresponding FP function to corresponding arguments yields corresponding results.

$$\begin{aligned} [\![(fx) : B]\!] &= \pi_B(fx) \\ &= (\pi_B \circ f)((\pi^{-1}_A \circ \pi_A)x) \\ &= (\pi_B \circ f \circ \pi^{-1}_A)(\pi_A x) \\ &= [\![f : A \rightarrow B]\!][\![x : A]\!] \end{aligned}$$

If an equation true in FP is given in which all constants are timeless, and types ascribed to its constants and variables such that all applications are properly typed, then that equation will remain true if interpreted in μFP. We can construct a μFP equation from an FP one as follows. Replace each constant in the FP equation by its projection, and each variable by a projection of a corresponding variable. Because the initial equation is well typed, all of the 'internal' projections cancel as shown in the example above. The only effect is to project each side by the same projection, so the equation is still true of all values of the new variables, provided only that they are of the right type. The result is a μFP equation in which the only requirement on the free variables is that they have the appropriate μFP types. Thus, the laws constructed in this way apply even to circuits which contain the sequential primitive $\mathcal{D}$. So, we can create new μFP laws simply by *promoting* FP laws which contain only timeless constants.

We must decide which of our FP constants are timeless. To do this, we must check that the meanings of the FP and corresponding μFP constants are related by the appropriate injection. For example, since the function *first* is stateless, we know that

$$[\![\mathit{first}]\!] = \propto \mathit{first}$$

We must check that *first* is timeless, i.e. that

$$[\![\mathit{first}]\!] = \pi_{[\alpha,\beta]\to\alpha}\mathit{first}$$

By the definition of π,

$$\begin{aligned}
\pi_{[\alpha,\beta]\to\alpha}\mathit{first} &= \pi_\alpha \circ \mathit{first} \circ \pi^{-1}_{[\alpha,\beta]} \\
&= \pi_\alpha \circ \mathit{first} \circ (\pi_\alpha^{-1} \parallel \pi_\beta^{-1}) \circ \mathsf{zip} \\
&= \pi_\alpha \circ \pi_\alpha^{-1} \circ \mathit{first} \circ \mathsf{zip} \\
&= \mathit{first} \circ \mathsf{zip} \\
&= \propto \mathit{first}
\end{aligned}$$

So, *first* is timeless. Note that here we have used the function *first* at two different instances. The *first* in the last line takes a pair of values to a value, whereas the *first* in the line before takes a pair of streams of values to a stream of values. This use of two instances always happens when we inject a function, and it shows that only polymorphic functions can be timeless. Similarly, the other plumbing functions, such as dist_L, zip and app_R^{-1} can be shown to be timeless.

The structural combining forms such as $\propto$ and $\triangle_R$ are also timeless. The injection gives us a systematic way of giving meaning to the μFP combining forms. Let us take $\triangle_R$ as an example.

$$\triangle_R : \forall\alpha.(\alpha \to \alpha) \to (\alpha^\star \to \alpha^\star)$$

For $\triangle_R$ to be timeless,

$$[\![\triangle_R]\!] = \pi_{(\alpha\to\alpha)\to(\alpha^\star\to\alpha^\star)}\triangle_R$$

$$\begin{aligned}
(\pi_{(\alpha\to\alpha)\to(\alpha^\star\to\alpha^\star)}\triangle_R)f &= (\pi_{\alpha^\star\to\alpha^\star} \circ \triangle_R \circ \pi^{-1}_{\alpha\to\alpha})f \\
&= \pi_{\alpha^\star\to\alpha^\star} \circ \triangle_R \circ (\pi_\alpha^{-1} \circ f \circ \pi_\alpha) \\
&= \pi_{\alpha^\star} \circ \triangle_R(\pi_\alpha^{-1} \circ f \circ \pi_\alpha) \circ \pi^{-1}_{\alpha^\star} \\
&= \mathsf{zip} \circ \propto\pi_\alpha \circ \triangle_R(\pi_\alpha^{-1} \circ f \circ \pi_\alpha) \circ \propto\pi_\alpha^{-1} \circ \mathsf{zip} \\
&= \mathsf{zip} \circ \propto\pi_\alpha \circ \propto\pi_\alpha^{-1} \circ \triangle_R f \circ \propto\pi_\alpha \circ \propto\pi_\alpha^{-1} \circ \mathsf{zip} \\
&= \mathsf{zip} \circ \triangle_R f \circ \mathsf{zip}
\end{aligned}$$

So,

$$\begin{aligned}
[\![\triangle_R f]\!] &= [\![\triangle_R]\!][\![f]\!] \\
&= \mathsf{zip} \circ \triangle_R[\![f]\!] \circ \mathsf{zip}
\end{aligned}$$

as we had before.

Most of the definitions of the combining forms given in [5] can be calculated in the same way. The only exception is the conditional combining form. We can see why conditional is not timeless by considering the multiplexing circuit cond. cond is not polymorphic; it demands that its first argument be a single boolean. However, when we project cond, we find that we are trying to use it with a stream as its first argument, so that the resulting equation cannot be correctly typed. So, $[\![\mathsf{cond}]\!] = \propto\mathsf{cond}$ is not equal the projection of cond, and cond cannot be timeless. The conditional combining form can be defined in terms of cond, and so is not timeless. In fact, only polymorphic functions can be timeless.

With the exceptions of $\mathsf{cond}, \mathcal{D}$, and $\mathcal{D}^{-1}$, all generic functions mentioned in this paper are timeless. It follows that the theorems quoted in the earlier discussion of FP are true of μFP.

Statelessness

There is a second, less general, way of promoting laws from FP to μFP. Any FP law can be promoted to μFP, provided all variables are guaranteed to be stateless. Suppose $A = B$ in FP. If A and B are stateless, we know that $[\![A]\!] = \propto A$ and that $[\![B]\!] = \propto B$.

$$A = B \;\Rightarrow\; \propto A = \propto B \;\Rightarrow\; [\![A]\!] = [\![B]\!]$$

So, $A = B$ in μFP also.

Sometimes, it is appropriate to use a mixture of the two forms of promotion. We noted earlier that the following theorem is true in FP, but is only necessarily true in μFP if h is stateless.

$$h \circ \mathsf{cond} \circ [p, [f, g]] = \mathsf{cond} \circ [p, [h \circ f, h \circ g]]$$

We divide the law into two parts. The first part cannot be made timeless because of the presence of cond.

$$h \circ \mathsf{cond} = \mathsf{cond} \circ [id, [h, h]]$$

The only way in which we can promote this law to μFP is to require h to be stateless. The second part

$$a = b \;\Rightarrow\; a \circ [p, [f, g]] = b \circ [p, [f, g]]$$

contains only the timeless constants composition and construction, so it can be promoted to μFP without placing any requirements on the variables. The net result is that the original FP law is true in μFP, provided h is stateless.

Conclusion

We have introduced the notion of timelessness, and have shown that any FP law which contains only timeless constants can immediately be promoted to μFP. Timelessness is defined by means of an injection which maps the meaning of an FP expression to the meaning of the corresponding μFP expression. With the exception of cond, the multiplexing function, and the sequential constants $\mathcal{D}$ and $\mathcal{D}^{-1}$, all of the plumbing functions and combining forms that we use to describe circuits are timeless. This means that we can reason about complex sequential circuits as though we were still in the simpler FP world. We no longer have to prove each individual μFP law correct.

In those cases where this timeless promotion cannot be used, we have shown that a weaker form of promotion, which demands that all variables be stateless can be applied. Some useful laws can be formed by a hybrid timeless/stateless promotion, and we have given an example of this. The final group of μFP laws comprises those which cannot be proved by promotion. These are laws which apply only to μFP and they will typically refer to the constants $\mathcal{D}$ and $\mathcal{D}^{-1}$, which have no counterparts in FP. The laws in this small group must be proved individually.

We have recently been investigating the use of relations to represent circuits[8]. Essentially, the same development can be performed as is described here, but with relations standing as the first order objects. The relation records the signals at the terminals of a circuit, but abstracts from the causal relationship between inputs and outputs, simplifying the treatment of complex data-flows. In contrast to the approach of [2], by confining the structure of the design to the same second- and higher-order functions described here we preserve the simple equational reasoning about circuits.

Tools are being developed to lighten the clerical burden of applying rewriting rules[3], and to record the transformation process as a means of documenting design decisions.

Acknowlegements

We would like to thank Wayne Luk for comments on an earlier draft, and John Hughes for some stimulating late night discussions about types. Part of the work on this paper was done while the second author was a visiting scientist at the IBM Almaden Research Center, San Jose.

References

[1] J. Backus, ‘Can programming be liberated from the von Neumann style?’, *Commun. ACM*, vol. 21(8), pp. 613–641, 1978.

[2] M.J.C. Gordon, ‘Why higher order logic is a good formalism for specifying and verifying hardware’, Technical report No. 77, Cambridge University Computing Laboratory, 1985.

[3] G. Jones and W. Luk, ‘Exploring designs by circuit transformation’, in W. Moore, A. McCabe and R. Urquhart (eds.), ‘Systolic Arrays’, Bristol: Adam Hilger, 1987, pp. 91–98.

[4] C.E. Leiserson and J.B. Saxe, ‘Optimising synchronous systems’, *J VLSI & Comput. Syst.*, 1983, vol. 1(1), pp. 41–67.

[5] M. Sheeran, 'μFP, an Algebraic VLSI design language', D.Phil. thesis, University of Oxford, 1983.

[6] M. Sheeran, 'Designing regular array architectures using higher order functions', in *Proc. Int. Conference on Functional Programming Languages and Computer Architecture*, Spriger-Verlag LNCS 201, 1985, pp. 220-237.

[7] M. Sheeran, 'Design and verification of regular synchronous circuits', *IEE Proceedings*, vol. 133, Pt. E, No. 5, pp. 295–304 September 1986.

[8] M. Sheeran and G. Jones, 'Relations + Higher Order Functions = Hardware Descriptions', in *IEEE Proc. Comp Euro 1987*, pp. 303–306

[9] U.C. Weiser and A.L. Davis, 'A wavefront notation tool for VLSI array design', in H.T. Kung, R.F. Sproull and G.L. Steele (eds.), 'VLSI systems and computations', (Computer Science Press, 1981), pp. 226–234.