Synchronous Digital Circuits as Functional Programs

PETER GAMMIE
The Australian National University and National ICT Australia, Sydney

Functional programming techniques have been used to describe synchronous digital circuits since the early 1980s and have proven successful at describing certain types of designs. Here we survey the systems and formal underpinnings that constitute this tradition. We situate these techniques with respect to other formal methods for hardware design and discuss the work yet to be done.

Hardware designs traverse a series of abstraction layers: what might begin as a high-level behavioural model that addresses architecture issues will, when mature, typically be manually translated into a register-transfer level (RTL) description that captures how the high-level computations are performed by finite-state means using logic gates and memories. This is typically validated against the original model using simulation and testing, or more formally with model checking techniques or a proof assistant. The resulting net lists (circuit schematics represented as graphs) are semi-automatically mapped to an implementation technology and laid out for realisation in silicon.

The original motivation for developing domain-specific languages (DSLs) [Mernik et al. 2005] for the upper reaches of this process was to harness the huge increases in transistor densities on silicon chips forecast by Moore’s law [Mead and Conway 1980]. It was hoped that productivity would rise with the abstraction level, allowing designs...
to be more reusable, scalable and correct. Traditional imperative programming languages were a poor fit as their implicit sequentiality conflicts with the intrinsic parallelism of hardware, and a global store is in tension with the ideal of placing computations physically near the relevant state [Nikhil 2011]. For these reasons simulation languages – specifically Verilog (based on C syntax) and VHDL (Ada) – were pressed into service as general-purpose hardware description languages (HDLs).

Despite their widespread use in industry, neither of these languages has been completely adequate. Their semantics are complex and have resisted useful formalisation [Boulton et al. 1992; Gordon 1995]. Only subsets of these languages can be synthesised to hardware, and these subsets need not be treated coherently by different tools. Moreover they lack modern semantically well-founded abstractions such as algebraic data types, higher-order functions (HOFs), overloading, subtyping and so forth. We contend that this leads to unnecessarily obfuscated descriptions, and greatly reduces the benefits of formal verification as it must be postponed until semantically-clear objects have been produced, which are typically low-level net-lists. This decreases the effectiveness and increases the cost of such techniques, as the cost of rectifying flaws is a function of when they are found [Brooks Jr. 1995]. In addition the high-level structure and intuitions must somehow be rediscovered in these lower-level artifacts.

In the face of these deficiencies, many people have investigated how circuits may be described as functional programs, with most treating the common special case of synchronous digital circuits. Such models abstract the propagation delays of the combinational logic but not the transitions between states; our simulations are cycle accurate with respect to their realisation in hardware, and we have a global clock. In contrast an asynchronous model allows different components in a system to proceed independently [Jantsch and Sander 2005].

The majority of the methods we examine are structural techniques for combining system elements. These elements have behavioural descriptions and may represent subsystems at any level of abstraction; we do not require that they be synthesisable, though in our examples we will take them to be familiar logic gates. We will not go below the gate level as our synchrony assumption breaks down and resistive and capacitive effects begin to intrude [Winskel 1986; Kloos 1987; Hanna 2000; Axelsson et al. 2005]. The advantages of a compositional semantics and reusable descriptions we take to be self-evident.

As implementers we would like to minimise the effort involved in providing the ever-increasing set of abstractions that users might like. One approach is to embed a DSL (to create an EDSL) into a suitably expressive meta language [Landin 1966; Hudak 1996], which allows the reuse of parsers, type checkers, optimisers, and some analysis tools while avoiding at least some of the myriad pitfalls of language design. We adopt Haskell as an exemplar of the modern functional programming languages [Hughes 1989; Peyton Jones 2003] that have been shown to be attractive hosts.

Here we focus on the successful tradition of rendering synchronous digital circuits and similar systems as more-or-less pure first-order functional programs. The key features of this approach are the non-standard evaluation order and the use of higher-order functions to structure the descriptions, which we discuss at length in later sections. We concentrate in particular on the simulation semantics given
data Signal α = α := Signal α

head :: Signal α → α
head (x := zs) = x

tail :: Signal α → Signal α
tail (x := zs) = zs

repeat :: α → Signal α
repeat x = x := repeat x

map :: (α → β) → Signal α → Signal β
map f zs = f (head zs) := map f (tail zs)

zip :: (α → β → δ) → Signal α → Signal β → Signal δ
zip f xs ys = f (head xs) (head ys) := zip f (tail xs) (tail ys)

false, true :: Signal Bool
false = repeat False
true = repeat True

neg :: Signal Bool → Signal Bool
neg sig = map not sig

and2 :: Signal Bool → Signal Bool
and2 sig _ = zip (&&) sig sig

delay :: α → Signal α → Signal α
delay x sig = x := sig


Fig. 1. A simple embedded DSL for describing synchronous circuits.

to these circuits, and touch on other interpretations such as circuit layout, energy consumption, hazard detection, worst-case timing analysis and technology mapping; Sheeran [2005] explores these topics in more depth along one of the lines of research reviewed here. The pragmatics of these description mechanisms are just as important as the clarity of the semantics: there is little point in algebraic simplicity if the descriptions are too inconvenient to write and maintain.

We begin our survey by discussing a folklore rendition of synchronous digital circuits in a non-strict functional programming language before examining the hardware description projects that have used these techniques. Afterwards we consider some closely related subjects and topics of future research.

1. CIRCUIT SEMANTICS

One may expect the semantics of gate-level descriptions of synchronous digital circuits to be straightforward, and indeed the prevailing attitude amongst existing hardware description languages seems to be that lifting standard propositional logic to a temporal domain suffices for simulation [Johnson 1983; Camilleri et al. 1986; O’Donnell 1987; Erkök 2002]. For concreteness we capture the essence of this approach in the set of combinators shown in Figure 1, expressed in Haskell.

Here the non-strictness of our host language is crucial; the Signal α datatype models an infinite stream of values of type α. A proper value of this type has the form x₀ := ... := xᵢ := ... for values xᵢ of type α, where the subscript indexes progression on an unbounded discrete timescale. That Signal α is also non-strict in the values it carries (of type α) is less essential, but we will demonstrate its utility later in this section.

Circuits are first-order stream transformers of type Signal α → Signal β, mapping streams of inputs to streams of outputs. State in sequential circuits is provided by a finite collection of initialised delay elements (clocked D-type flip flops) that provide access to values from the previous instant, and the instantaneous value of any wire is a function of the inputs and the values of the delay elements for that instant.
we will see this approach supports many useful equational laws that are often easier to apply than those for reasoning about arbitrary mutable state.

Our implementation of combinational logic is a pointwise lifting of the instantaneous operations to the temporal domain. As we use Haskell’s recursion to model feedback, “cons” (our $>$ operator) should not evaluate its arguments [Friedman and Wise 1976]. In other words evaluation is driven by data dependencies only.

Clearly we can derive other operations such as $\text{xor}$, and as we will explore later in more detail, write succinct circuit generators in Haskell.

By way of an example, consider the twisted ring counter of Stavridou [1993, §3.3.2] shown in Figure 2. This circuit cycles through the sequence $000 \rightarrow 100 \rightarrow 110 \rightarrow 111 \rightarrow 011 \rightarrow 001$, which intuitively involves complementing the rightmost bit and moving it to the leftmost position, shuffling the others to the right. It self stabilises: whatever the state of the delay elements, the circuit will return to this sequence in a finite number of steps.

Each binding defines a wire, and the meaning of the whole network is the least fixed point of this set of equations. As such it is a Kahn network [1974]; Claessen [2001, Chapter 5] presents many examples written in this style.

Note that each syntactic use of a gate in the description is intended to correspond to an actual gate in the hardware realisation. We will see that this expectation is in tension with the semantics of the host language in the same way that assuming that each procedure definition in a program is represented in the compiled object code is sometimes erroneous.

This encoding is termed a shallow embedding as there is no syntactic representation of circuits that can be manipulated from within Haskell. Its strength is that we can easily add new types of circuit elements, and freely reuse Haskell as a metalanguage. Its weakness is that we cannot manipulate descriptions from within the language, or reason about them inductively. In contrast a deep embedding would explicitly represent syntax, which can be challenging to define and use in a typed setting. Later we will see that Haskell’s type classes provide a third way.

Our naïve semantics has an infelicity, however. Consider the following circuit:
Synchronous Digital Circuits as Functional Programs

<table>
<thead>
<tr>
<th>(&amp;&amp;)</th>
<th>⊥</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>⊥</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>and</th>
<th>⊥</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>F</td>
<td>⊥</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>⊥</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pand</th>
<th>⊥</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>F</td>
<td>⊥</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Fig. 3. The truth tables of short-circuit and (&&) standard to most programming languages, bi-strict and and parallel and. Values for the first argument are on the left, and for the second on the top. The value ⊥ denotes a diverging argument.

\[
f(x) = \text{out} \\
\text{where} \\
\quad \text{out} = \neg (\text{and2 out} x)
\]

We can see that \( f(x) \) diverges for all \( x \) by considering the definition of \( \text{and2} \) in Figure 1 and the semantics of (&&) shown in Figure 3. In contrast the symmetric variant \( f'(x) = \text{out where out} = \neg (\text{and2} x \text{ out}) \) converges to true on the argument false. This behaviour is termed short-circuit evaluation in strict languages such as ML and C.

As \( f \) and \( f' \) have isomorphic circuit diagrams, we expect them to have the same semantics, and therefore \( \text{and2} \) should make symmetric use of its inputs. One option is to make \( \text{and2} \) strict in the head of its second argument, causing both \( f \) and \( f' \) to always diverge. This yields the traditional model where every well-defined loop is required to contain a delay, and as we will see, this must be the semantics intended by the champions of the approach sketched above. Here we explore the less trodden path of making \( \text{and2} \) non-strict in both its arguments.

To motivate this choice, consider the classic example due to Malik [1993] shown in Figure 4. For any circuits \( f \) and \( g \), this circuit generator is intended to dynamically choose between \( f \circ g \) and \( g \circ f \) using only single copies of \( f \) and \( g \) and three multiplexers. (A multiplexer chooses between one of its inputs on the basis of an

\[
\begin{align*}
\text{c} & \quad \text{x} \\
\hline
\text{1} & \quad \text{f} \\
\text{0} & \quad \text{g} \\
\hline
\text{c} & \quad \text{x} \\
\end{align*}
\]

Fig. 4. A useful cyclic circuit schema that, for arbitrary \( f \) and \( g \), computes either \( (f \circ g) \cdot x \) or \( (g \circ f) \cdot x \) depending on the input \( c \). Without cycles two copies of \( f \) and \( g \) would be required.

auxiliary input.) What makes this design work is that the apparent combinational cycles in the schematic cannot be realised dynamically, i.e., every assignment to the inputs yields an acyclic path through the circuit, assuming that the multiplexers are symmetrically non-strict in the inputs they choose between. If we construct such multiplexers from the basic gates and neg, then must be lazy in at least one argument for this to obtain. We discuss this example further in §2.3.

Another example is the hardware bus arbiter of R. de Simone that is naturally rendered as a combinatorially-cyclic circuit [Potop-Butucaru et al. 2007, §2.3]. Fairness is enforced by circulating a token around a ring of arbiter cells, and the token holder can delegate permission to proceed to the succeeding cells in the ring.

These cycles also arise naturally when we abstract from the gate to the functional block level, as observed by Burch et al. [1993]. Their example of a carry-lookahead adder requires the adders and carry-lookahead generator to instantaneously interact across an abstraction boundary.

Semantically we can treat cyclic combinational logic in the same way as other recursive definitions, by using a domain [Winskel 1993]; in this instance we introduce a third value to our Bool type and impose a (partial) information ordering on these values:

\[
\begin{array}{c}
F \\ \\
\downarrow \\
T \\
\end{array}
\]

This is to say that the undefined value is less defined than either of T and F, and that these two proper values are distinct. Intuitively we take to mean that the wire does not settle to a valid value, with and T representing the standard stable Boolean values. We emphasise that is not so much an unknown value as an invalid one in this semantics.

Using this domain we can give a symmetrically non-strict semantics to our and primitive using the pand function shown in Figure 3, which also shows two of its stricter cousins for comparison. With unfortunate consequences for our simple embedded DSL of Figure 1, Plotkin [1977] showed that pand is not sequentially computable; see Gunter [1992, §6.1] and Brookes [1993] for further background on this point. As most functional languages are intended to have such a deterministic sequential semantics, we should use the stricter and if we rely on the host language’s recursion. If we wish to support combinational cycles then we need to adopt an alternative semantics for recursion, such as explicit iteration of some reified representation, which implies that we can no longer write our circuits as simple recursion equations in the host language. Alternatively one could introduce support for parallel or non-deterministic operations into the host language, but doing so can severely complicate its implementation and semantic properties [Hughes 1983; Moran 1998].

We note that the sequential behaviour of circuits is unaffected by this change to the combinational semantics; we continue to use non-strict sequences. However this may not be true if we wish to accommodate non-determinism, or ways of observing the circuit without changing its interface, such as for debugging purposes. How invasive this is could be taken as a measure of how flexible the methods of the
following sections are.

Constructive circuits always assign all wires non-$\bot$ values when always fed non-$\bot$ inputs; these can be unfolded into semantically-equivalent acyclic circuits, which can then be passed to tools that do not directly support combinational loops. These circuits are termed “constructive” due to their relationship with intuitionistic propositional logic. Such circuits have been used to give a semantics to an imperative synchronous language (see §3.1).

Combinational cycles trade time for space, and convergence may require time exponential in circuit size [Shiple et al. 1996] in the presence of nested loops. Neiroukh et al. [2008] found references to these types of circuits stretching back to switching theory in the 1960s. Shiple et al. [1996] have grounded this parallel semantics in the physical models of Brzozowski and Seger [1995]. The connection with constructive logic continues to be explored by Mendler et al. [2012], and Riedel and Bruck [2003] show that cycles can yield significant space reductions in practice.

The reader should not be seduced into believing this semantics completely reflects the physical behaviour of cyclic circuits. Consider the classic set-reset latch:

$$srLatch \bar{s} \bar{r} = (q, \bar{q})$$

where

$$q = \neg (\text{and} 2 \bar{s} \bar{q})$$
$$\bar{q} = \neg (\text{and} 2 \bar{r} q)$$

While the structural description on the right is accurate, the semantics we have ascribed to the primitives does not yield the desired latching behaviour as observed in practice. This is because the retention of the latch’s value across cycles depends crucially on the propagation delays that our assumption of synchrony has already abstracted from, and the semantics presented here does not retain the values of wires between cycles. Similarly tri-state busses may not be properly treated by this semantics either.

Descriptions in this style are quite pleasant as the connection with the circuit’s net list is quite clear, and there is no extraneous sequentiality; these recursion equations encode data dependency amongst the components and nothing more. Moreover we can easily incorporate subsystems described at more abstract levels than primitive gates for the purposes of high-level design validation. However giving these descriptions alternative semantics, such as an explicit representation of a circuit’s net list, is difficult in a pure host language. We discuss this issue in §2.3 and later sections.

2. CIRCUITS AND FUNCTIONAL PROGRAMMING

Having sketched the semantics we might expect of an HDL for synchronous digital circuits, we now review systems that represent circuits using functional programming languages. We begin with the the combinatory approach of $\mu$FP, and the
contemporaneous use of recursion equations by Johnson [1983]. Hydra bridges the two traditions and points the way to the Haskell-hosted Lava systems that continue to be developed. We discuss the Hawk project that applied these techniques to microarchitectures, the Jazz system, and the Cryptol® language for describing implementations of cryptographic primitives. We conclude with some higher-level behavioural techniques.

### 2.1 \(\mu\)FP

Sheeran [1984] based her \(\mu\)FP system on the FP language of Backus [1978], who championed a combinatory style of programming now termed point-free. In essence, function composition is emphasised over application, and algebraic laws are prized [Bird 1987; Meijer et al. 1991].

\(\mu\)FP extends FP by lifting instantaneous operations to streams with the \(\alpha\) combinator, better known as map, and a delay operator \(\mu\):

\[
\begin{align*}
\mu &:: (\text{Signal }\alpha, \text{Signal }\delta) \sim (\text{Signal }\beta, \text{Signal }\delta) \\
&\rightarrow \text{Signal }\alpha \sim \text{Signal }\beta \\
\mu f &= \lambda x. \text{let } (y, z) = f(x, \text{delay }? z) \text{ in } y
\end{align*}
\]

The diagram on the left depicts \(\mu f\) for an arbitrary circuit \(f\), and on the right is a simulation semantics for \(\mu\) in Haskell. The latter should not be taken too literally as both FP and \(\mu\)FP are untyped, and the only constraints on the implementations of combinators is that they satisfy the associated laws. We again informally identify the type of stream with the Signal domain. A strength of the combinatory approach is that the type of circuits \(\alpha \sim \beta\) which map inputs of type \(\alpha\) to outputs of type \(\beta\) can be separated from the function space of the meta language \(\alpha \rightarrow \beta\). Note that the register introduced by \(\mu\) is initialised by the “don’t care” constant ? value.

Circuits are described structurally and given two semantics: simulation, by translation into the sequence type of FP along the lines of what we sketched in §1, and layout using the DSL for functional geometry of Henderson [1982]. This early example of reinterpretation was realised as a custom processor rather than an embedding in a host language.

Higher-order functions (HOFs) capture the regularity of data-oriented circuits in an elegant manner. For example, the row combinator\(^1\) expresses a common pattern used, for instance, in a simple ripple-carry adder:

\[
\begin{align*}
\text{row} &:: (\alpha, \beta) \rightarrow (\alpha, \delta) \\
&\rightarrow (\alpha, [\beta]) \rightarrow (\alpha, [\delta]) \\
\text{row }f (x, []) &= (x, []) \\
\text{row }f (x, y:ys) &= \\
&= \text{let } (x', z) = f(x, y) \\
&\quad (x'', zs) = \text{row }f(x', ys) \\
&\quad \text{in } (x'', z : zs)
\end{align*}
\]

We note that such structural definitions are much more intuitive and less verbose than a typical generic definition in VHDL, where the use of array indices introduce

\(^1\)The row function is called \texttt{mapAccumL} in the standard Haskell \texttt{Data.List} module.
the spurious possibilities of off-by-one errors and so forth.

μFP emphasises composition and not the primitive circuits; the latter are not further specified by Sheeran [1984]. Instead a fixed set of higher-order combining forms that have good geometric and algebraic properties are studied. Sheeran observes that almost all the laws of FP apply to μFP, with the notable exception of a conditional distribution law. The FP version is as follows:

\[
\begin{align*}
  h \circ (i \rightarrow t ; e) &= (i \rightarrow h \circ t; h \circ e)
\end{align*}
\]

where

\[
\begin{align*}
  (- \rightarrow - ; -) &:: (\alpha \rightarrow \text{Bool}) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \\
  (i \rightarrow t; e) &= \lambda \alpha. \text{if } i \alpha \text{ then } t \alpha \text{ else } e \alpha
\end{align*}
\]

Lifting (- \rightarrow - ; -) to μFP is an exercise in tuple spaghetti:

\[
\begin{align*}
  (i \rightarrow t; e)_{\muFP} &= \text{map}(\pi_1^3 \rightarrow \pi_2^3; \pi_3^3) \circ \text{zip3} \circ [i, t, e]
\end{align*}
\]

where \([i, t, e]\) is informal notation for the fanout \(\lambda x. (i x, t x, e x)\) and \(\pi_i^n\) projects the \(i\)th component of an \(n\)-tuple. In μFP, \(h\) must be combinational for the putative equation to hold, for there is always a stateful \(h\) that can distinguish \(t\) from \(e\) if they are different.

Sheeran also proposed a fixed-point fusion rule for her μ construct:

\[
\begin{align*}
  \mu[f, g] \circ \mu[h, j] &= \mu[f \circ [h \circ \pi_1^2 \circ \pi_2^2, \pi_2^2 \circ \pi_2^2], \pi_1^2 \circ \pi_2^2], \\
  &\quad \quad [g \circ [h \circ [\pi_1^2 \circ \pi_2^2, \pi_2^2 \circ \pi_2^2], \pi_1^2 \circ \pi_2^2], j \circ [\pi_1^2, \pi_2^2 \circ \pi_2^2]]
\end{align*}
\]

This law is intended to be used as a fission law, in the right-to-left direction: it moves the independent parts of a state-holding element closer to the relevant computations. We note that this law does not hold in our Signalα domain due to the presence of partial lists; we discuss this issue further in §3.5.

A major source of discomfort in the purely combinatory style of programming is the need to explicitly route values from definition to use; in the applicative style we used in §1 the λ-calculus provides this service implicitly by allowing us to give names to wires in some scope. This plumbing problem is certainly why raw combinators are generally thought of as compilation targets and not source languages.

μFP has been applied to the design of circuits with regular structure such as adders, multipliers and a correlator, and more generally to systolic arrays, where critical path lengths are reduced by pipelining in a hazard-free non-recursive way.
The process begins with purely combinational circuit designs which are transformed into sequential pipelines by retiming transformations. Through a disciplined use of the state introduced in this final step, the original and retimed circuits can be very simply related. All examples are data-oriented, and control-oriented circuits do not tend to have the geometric regularities that these combinators capture.

Sheeran [2005] reviews this line of research as well as her work on some of the descendants of this system that we discuss later in this article.

2.2 Hardware synthesis from first-order recursion equations

Johnson and his collaborators have made an extended investigation into the practical use of derivational reasoning in digital design [Johnson 1983; Johnson and Bose 1997; Johnson 2001]. Their goal is to provide tools to explore the space of implementations of a high-level behavioural specification. Here synchronous digital circuits are represented as first-order recursion equations over streams as we discussed in §1.

The first major application of these techniques [Johnson 1983, Chapter 5] was to the refinement of an interpreter for a higher-order language into a stack-based virtual machine using the approach developed by Wand [1982]. This process relied on general results about flowchart schemata [Manna 1974; Greibach 1975], such as the fact that all tail-recursive functions can be implemented in constant space, and arbitrary functions can be evaluated using a stack. Given that these schemata can be captured by higher-order functions, we can see this as a control-oriented complement to the $\mu$FP agenda, but where the original specifications are behavioural and more abstract.

This process uses the program transformation framework of Burstall and Darlington [1977], with the preservation of total correctness left to the discretion of the designer [Johnson 1983, §2.4.5]. Circuits represented as recursive streams are optimised using equations similar to those in the previous section [Johnson 1983, Chapter 6]. Suitably oriented, these equations can transform circuits that operate on their arguments in parallel into sequential ones.

An untyped lazy functional programming system by the name of Daisy was the vehicle for this research, and circuit descriptions were manipulated by hand. A strength of this approach is that all refinement artifacts are executable, i.e., can be experimented with programmatically.

Building on this work, Johnson and Bose [1997] developed the DDD tool. Here the refinement process begins with a first-order specification expressed as a pure iterative (tail recursive) function in the strict untyped functional language Scheme, extended with a mechanism for recursively defining streams. These are structurally decomposed into putative hardware blocks, again using the Burstall and Darlington [1977] rules. From these DDD mechanically generates architectural descriptions consisting of control and datapath circuitry, which is further optimised using laws about recursive stream transformers like those we have seen before. Finally representations of abstract types such as numbers are chosen and shown sufficient using data refinement.

Some of these steps have side conditions, such as showing that a fixed-width binary representation of a number is adequate. Paul Miner [Johnson 2001, §3.3] experimented with using the PVS proof assistant to demonstrate these conditions and the soundness of “ingenious” circuit optimisations but was stymied by the lack
of support for infinite streams in proof assistants at the time.

Johnson and his colleagues have used this approach to derive implementations of the FM8501 and FM9001 processors due to Hunt [Bose and Johnson 1993], a PCI bus interface and a Java byte code generation core. They claim that this is a useful technique for circuits with high algorithmic complexity. Similarly to µFP, it does not address the specification of interfaces, power supplies or clock trees.

2.3 Hydra

Hydra [O'Donnell 1987; 1992; 1995; 2003] is a long-running experiment in representing circuits in various pure functional programming languages following Johnson's tradition of circuit transformation. It holds fast to the idea of directly expressing circuits in the host programming language and reasoning equationally in that language. This has the advantage over µFP of allowing new combining forms to be introduced by the end user of the system.

The central problem with this approach is of identifying shared subcircuits. Consider this rendition of the \( \text{fgORgf} \) circuit in Figure 4 in the style of Hydra (and Lava 2000 which we meet later in this article):

\[
\text{fgORgf} :: (\text{Signal } \alpha \leadsto \text{Signal } \alpha) \rightarrow (\text{Signal } \alpha \leadsto \text{Signal } \alpha) \\
\rightarrow (\text{Signal Bool, Signal } \alpha) \leadsto \text{Signal } \alpha \\
\text{fgORgf } f \ g \ (c, x) = \text{out}
\]

where

\[
\begin{align*}
\text{fOut} & = f (\text{mux} (c, x, \text{gOut})) \\
\text{gOut} & = g (\text{mux} (c, \text{fOut}, x)) \\
\text{out} & = \text{mux} (c, \text{gOut}, \text{fOut})
\end{align*}
\]

where the \( \text{mux} \) combinator is defined as:

\[
\text{mux} :: (\text{Signal Bool, Signal } \alpha, \text{Signal } \alpha) \leadsto \text{Signal } \alpha \\
\text{mux} (c :> cs, x :> xs, y :> ys) = (\text{if } c \text{ then } x \text{ else } y) :> \text{mux } cs \text{ } xs \text{ } ys
\]

If we think of \( \text{fgORgf} \) as a standard Haskell definition then we can apply the unrestricted \( \beta \)-rule to unfold the definition of \( \text{gOut} \) in the definition of \( \text{fOut} \):

\[
\text{fOut} = f (\text{mux} (c, x, (g (\text{mux} c \text{ fOut } x))))
\]

This new circuit is extensionally equal to the previous one, and so these should not be distinguished by any Haskell context. However they are clearly structurally distinct, as the new version uses two copies of \( f \). In other words, \( \beta \)-reduction invalidates our identification of function definitions with hardware gates. Therefore we seek a way to make these circuits observably different while retaining enough of the host language's semantics to support the kind of equational reasoning that circuit transformations depend upon.

O'Donnell has proposed several ways of resolving this reification problem (see also Claessen [2001, Chapter 3]):

— In more pragmatic times, O'Donnell [1987] suggested the use of pointer equality to reify the expression graph of the circuit. This is a non-conservative extension to a pure language, rendering the foundational \( \beta \)-rule potentially unsound everywhere, thereby destroying equational reasoning.
— O’Donnell [1992] asked the circuit designer to do what the language processor could not; a combinator is added so that labels can be manually attached to components. This approach is inconvenient, non-compositional and impedes the use of higher-order combinators such as \texttt{row}.

— Most recently, O’Donnell [2003] advocated the manipulation of the circuits as Haskell abstract syntax using Template Haskell [Sheard and Peyton Jones 2002]. This is at best a partial solution as the syntax for circuits and generators are not clearly separated here; intuitively we expect to run a circuit generator, perhaps using higher-order combinators as canvassed in \S2.1, that yields the abstract syntax of a particular circuit. As the generators are arbitrary definitions in a Turing-complete language, it is difficult to see how this approach is any easier than writing a traditional standalone language processor. In any case manipulating the abstract syntax of the host language is fraught with semantic issues and runs the risk of destroying many of the reasoning principles valued by functional programmers. We discuss this approach further in \S4.

Hydra supports a variety of circuit semantics [O’Donnell 1995], though as we observed earlier, below the synchronous gate level lurk many subtle issues. O’Donnell and Rünger [2004] designed a carry lookahead adder using Hydra as a notation for reasoning in the Squiggol style popularised by Bird [1987].

2.4 Lava

The original Lava system [Bjesse et al. 1998] was an attempt to embed a flexible hardware description language into pure Haskell in such a way that circuit descriptions could be both generated and manipulated within the host language. 

\textit{Type classes} [Kaes 1988; Wadler and Blott 1989] were used to give a signature for the circuit primitives. By parametrising these with a \textit{monad} [Wadler 1997], each interpretation of a circuit in Lava could encapsulate the effects it requires. For instance, a net list interpretation may use a state monad to assign a number to each wire and map each basic component into a graph node. Effects such as non-determinism or probing encapsulated signals can be easily modelled using appropriate monads. This is the middle path between shallow and deep embeddings mentioned in \S1, and is now termed a \textit{finally tagless} representation [Carette et al. 2009].

The provided loop combinator supports cycles in sequential logic:

$$\text{loop} :: \text{CircuitMonad}\ m \Rightarrow (\alpha \rightarrow m\ \alpha) \rightarrow m\ \alpha$$

where the \texttt{CircuitMonad} class is the signature of this and the other basic circuit combinators. Intuitively such a recursion operator should perform the effects of its argument computation only once while providing the computation access to the value it finally yields. This invalidates an unfolding semantics, and therefore the application of the $\beta$-rule that duplicated circuitry in \S2.3, while preserving this law in the purely functional parts of the language. Erkők [2002] later gave an axiomatic treatment of these operators, and developed a syntax to reduce the syntactic burden when defining several values by simultaneous monadic recursion. Here is our \texttt{fgORgf} example is this style\footnote{The \texttt{do rec} syntactic form has displaced the keyword \texttt{mdo} introduced by [Erkők 2002].}:

\begin{verbatim}
    do rec
\end{verbatim}
\[
\text{fgORgf} :: \text{CircuitMonad} m \\
\Rightarrow (\alpha \to m \alpha) \to (\alpha \to m \alpha) \to (\text{Bool}, \alpha) \to m \alpha
\]
\[
\text{fgORgf}\ f\ g\ (c,\ x) = \\
\begin{array}{l}
\text{do rec}\\
f\text{Out} \leftarrow \text{mux}(c,\ x,\ g\text{Out}) \gg= f\\
g\text{Out} \leftarrow \text{mux}(c,\ f\text{Out},\ x) \gg= g\\
\text{mux}(c,\ g\text{Out},\ f\text{Out})
\end{array}
\]

where the bind operator (\gg=) is a monadic equivalent to (reverse) function application, and \text{mux} now has type \text{CircuitMonad} m \Rightarrow (\text{Bool}, \alpha, \alpha) \to m \alpha; our type of circuits \(\alpha \rightsquigarrow \beta\) is concretely \(\alpha \to m \beta\).

We contend that this description is almost as syntactically appealing as those in Hydra (§2.3) and Lava 2000 (§2.5). However the monadic structure makes visible the order in which the components of the circuit are defined [Claessen 2001, §1.8]; in other words, a circuit can be given two semantically distinguishable descriptions in this notation simply by permuting the monadic commands. We might attempt to repair this infelicity by requiring that our monad be commutative, i.e., that it is insensitive to such permutations, but clearly any interpretation that assigns unique names to the gates will fail to have this property. This lack of full abstraction also complicates formally reasoning about circuit equivalences.

The original Lava system suffered somewhat from the limitations of using single-parameter type classes for reinterpretation, and successor systems such as Hawk (§2.7) experimented with generalisations.

2.5 Lava 2000

Lava was later refined by Claessen [2001] into the Lava 2000 system, which is an embedded DSL for parametrised circuits whose instances are fed into myriad tools for analysis: simulation and realisation in hardware via the industry-standard VHDL, model checking of various kinds [Halbwachs et al. 1993; Clarke et al. 1999], testing with QuickCheck [Claessen 2001, Chapter 4] and so forth. This design-and-verify approach contrasts sharply with the transformational correct-by-construction approaches championed by Sheeran (§2.1), Johnson (§2.2) and O’Donnell (§2.3), all of which rely on equational reasoning in the host language.

In Lava 2000 circuit generators are standard Haskell expressions as we saw in §2.3. When run, these expressions generate a description of a concrete circuit which is reified into a data structure by disciplined pointer-equality testing. This is termed observable sharing. In contrast to the earlier systems Lava 2000 has no need of a precise semantics for its host language as it is merely the language of circuit generators, which are only executed and not analysed.

Claessen [2001, §3.3.4] notes that observable sharing makes visible the difference between call-by-need (laziness) and call-by-name (non-strictness): circuits without parameters are shared whereas those with parameters are duplicated, acting like templates. This loss of the \(\beta\)-rule of the \(\lambda\)-calculus is hardly surprising – we are trying to identify sharing, which is precisely the distinction between these semantics. At the source level this problem is ameliorated by the adoption of a particular style of description that is less likely to trap the unwary. It also relies on defeating compiler optimisations such as common-subexpression elimination and the full laziness transformation [Peyton Jones 1987] that introduce sharing.
Lava 2000 additionally marked a departure from using the underlying lazy functional programming language to give a direct semantics for circuits: instead, the circuit generator builds a monomorphic graph describing the final circuit, which is then interpreted by traversal. Extra types of circuit elements such as non-deterministic choice can be modelled as distinct kinds of graph nodes. Circuits are therefore a subset of Haskell expressions that are treated as abstract syntax, similarly to O’Donnell [2003] but within a single metalanguage.

This approach allows Claessen [2003] to handle circuits with combinational cycles by computing explicit (reified) fixed points, but precludes the possibility of polymorphic signals: circuits in Lava talk about bits and integers only. Moreover it limits the possibility of transmitting some of the structure of the circuit generator to the backends without extensive surgery to Lava 2000 itself. For instance, it may be more efficient for a tool consuming these descriptions to generate a single instance of a circuit and copy that as required instead of receiving the entire description of a subsystem at each point of use. Also by allowing arbitrary HOFs as combining forms, circuits in Lava 2000 do not always have reasonable layouts.

Lava 2000 and a variant designed by Satnam Singh at Xilinx (§2.6.1) were applied to the design and realisation of a sorter core based on Batcher’s butterfly techniques [Claessen et al. 2003]. They have also been used to analyze many other combinational circuits such as adders and multipliers [Axelsson 2003], and as a host for a sequential language much simpler than what we discuss in §3.1 [Claessen 2001, Chapter 6]. More recently Sheeran [2005; 2011] has developed techniques for context-sensitive circuit generators and optimisers using this system.

2.6 Other Lavas

“Lava” has come to denote the structural description of hardware in Haskell. We briefly review three of these systems.

2.6.1 Xilinx Lava. As mentioned earlier, Singh developed a variant of Lava while at Xilinx, Inc. as an experimental vehicle for mapping circuits to the company’s Virtex line of Field Programmable Gate Arrays (FPGAs, a type of reconfigurable hardware). In contrast to other Lavas, this system included explicit layout combinatorics similar to those in µFP (§2.1) [Singh and James-Roxby 2001]. Singh [2011] shows that user-specified layouts remain useful in some cases.

Circuit descriptions are similar to those in Lava 2000. Primitive gates are specified in terms of the look-up tables that FPGAs provide. Sharing is accounted for using a monad internally, which creates a monomorphic graph that is then translated into VHDL (etc.) for consumption by external tools. There is no support for cycles of any kind.

In addition to the sorter network mentioned above, Xilinx Lava was used to describe dynamic (runtime) reconfiguration and specialisation [Singh 2004]. Unusually for a Lava, clock signals are explicitly mentioned in descriptions, allowing the suspension of a stateful circuit through clock gating.

2.6.2 York Lava. Naylor and Runciman [2012] use York Lava to describe their Reduceron graph-reduction processor, which runs on an FPGA. This is a revival of the idea of programming-language specific processors that avoid the von Neumann bottleneck of a single global store. Such experiments are far easier to carry out now
as reconfigurable hardware is quite affordable, and more likely to be adopted as the sequential performance of standard processors flatlines. The source language is the kernel of a lazy language compatible with Haskell. The processor is described in a mix of recursion equations and an imperative behavioural language they call Recipe, which is given a semantics by translation into their Lava.

The semantics of York Lava is standard. The project investigated the use of explicit fork points to signal sharing [Naylor and Runciman 2009]: the overloaded fork combinator should be used to indicate that a wire has multiple sinks. This allows most useful circuits to be reified while retaining the purity of the host language in a manner ultimately quite similar to the explicit use of recursion combinators. However this approach was abandoned in favour of Lava 2000-style pointer comparisons.

Layout is performed by the FPGA toolset.

2.6.3 Kansas Lava. The Kansas Lava system is a vehicle for investigating circuit transformation and refinement. Gill and Farmer [2011] report on the “semi-formal” derivation of an error-correcting code using the worker/wrapper transformation [Gill and Hutton 2009; Gammie 2011], in concert with applicative functors [McBride and Paterson 2008] and type functions [Chakravarty et al. 2005]. In contrast to the structural use of lists we saw in §2.1, the dimensions of vectors and matrices are encoded in their types, which is both safer and more awkward as present Haskell systems do not have full support for type-level arithmetic. Layout is not prescribed.

Gill [2009] previously advocated another solution to the reification problem: instead of polluting the semantics of the pure core of Haskell by making the sharing of values observable at all types ala Lava 2000 (§2.5), scrutinising the structure of a circuit is confined to the IO monad, where anything goes. Once again a test for pointer equality is employed, and this extra discipline makes the approach both safer – one is less likely to accidentally exploit the observation of sharing – and more obscure, as the semantics of the IO monad is complex, fluid and yet to be formally specified. Moreover it suffers from exactly the same problem as Lava 2000: by allowing call-by-name and call-by-need semantics to be distinguished, the β-law of the λ-calculus fails, as we previously remarked. This may complicate relating fully-formal derivations and Kansas Lava circuits and generators.

This system uses the standard Kahn network semantics for circuits, and maintains both a shallow and deep embedding of the circuit to allow for direct simulation and VHDL export. As a result the simulation semantics of the circuits is not isolated from Haskell’s, which precludes a treatment of combinational cycles. Clock information is explicitly encoded into types in a manner similar to Lucid Syncrone (see §3.1).

Layout is performed by external tools.

2.7 Hawk

Hawk [Matthews et al. 1998; Launchbury et al. 1999] is a DSL embedded in Haskell for describing and reasoning about microarchitecture. Semantically it is very traditional, employing non-strict lists of values to model synchronous systems, though it does not require nor guarantee that these systems be finite-state.

The emphasis of this system is on algebraic abstractions of pipelined microprocessor designs using transactions, which record the relevant state of the system.
for each instruction as it proceeds through the pipeline. This requires more type structure than allowed by Lava 2000. Early versions of Hawk attempted to use the type classes and monads of the original Lava, but this approach was abandoned due to the difficulty of finding a suitable recursion combinator, and the lack of methods for resolving ambiguous uses of multi-parameter type classes that represent relations between types. Many of the issues they identified were soon addressed [Jones 2000; Erkők 2002; Chakravarty et al. 2005]. Later versions of Hawk provided only a simulation semantics along the lines of §1.

The proposed algebraic laws for manipulating microarchitectures were verified in Isabelle/HOL [Nipkow et al. 2002], for which the theory of converging equivalence relations was developed by Matthews [1999] to allow the definition of recursive functions in HOL over infinite sequences. Under mild conditions such functions have unique fixed points, and unlike the domain theoretic approach, uncomputable functions can be defined. We discuss formal models further in §3.5.

The Hawk group built models of the then state-of-the-art Intel Pentium Pro in addition to the DLX, a standard example of a pipelined processor. Matthews [2000] reviews the project and discusses how Hawk relates to other HDLs.

2.8 Cryptol

Cryptol® is a proprietary DSL and toolset developed by Galois, Inc. for compiling descriptions of cryptographic algorithms into hardware or software [Browning and Weaver 2010]. The language provides only bits as a primitive type, with sized sequences and tuple constructors used to aggregate values. Its type system is very flexible, allowing the definition of size- and type-polymorphic functions, and constraints allow sizes to be underspecified. Cryptol descriptions can be checked for equivalence using external tools such as SAT and SMT solvers.

Combinational circuits are described applicatively, as in §1, but as instantaneous functions. These can be lifted to streams pointwise, as before, or as transition functions for state machines in the coiterative style using an unfold combinator. The language restricts the use of higher-order functions to those that can be unfolded at compile time, which is often sufficient for the sort of circuit combinators discussed in §2.1. Partial application is not supported, and functions are uncurried.

A construct similar to Haskell’s list comprehensions is used to define streams recursively, which is realised as delayed feedback in the generated circuit. It is also used to traverse finite sequences, and the language goes beyond purely structural descriptions by providing par, seq and reg combinators that specify how the comprehension should be scheduled in time and space. Browning and Weaver [2010, §3.4] show that, by default, mapping a function f across a finite sequence s yields hardware with as many fs as the width of s, whereas the seq annotation generates only a single f and the requisite synchronous scheduling logic to process s sequentially. The reg combinator pipelines a circuit in a standard way.

Layout is once more performed by external tools.

2.9 Jazz

The Jazz system was developed by A. Frey, with contributions from F. Bourdoncle, G. Berry, P. Bertin and J. Vuillemin, contemporaneously with the original Lava system [Claessen 2001, §1.11]. It had a Java-inspired syntax but was in fact a higher-
order, lazy, purely-functional language that supported the combination of subtyping and parametric polymorphism proposed by Bourdoncle and Merz [1997]. Novelty it provided native support for the arithmetic over 2-adic integers (streams of booleans) due to Vuillemin [1994]. The elaboration of circuit descriptions into netlists is similar to Lava’s approach. Descriptions can be given multiple interpretations by the standalone language processor.

2.10 High-level Hardware Synthesis

At a higher level we might hope to abstract from timing behaviour by compiling behavioural descriptions into synchronous or asynchronous circuits. Several such systems are based on ideas closely related to functional programming.

SAFL [Mycroft and Sharp 2003] is a first-order pure functional language with a strict semantics where the only program schema on offer is tail recursion. As each function in a SAFL description is mapped to a hardware block, the key task of its FLaSH compiler is to schedule the use of these blocks when they are called from multiple places in the source program.

A similar approach was taken in the design of the SASL first-order stream processing language [Frankau and Mycroft 2003]. Tail-recursive functions define streams, where each iteration yields zero or more elements. Unlike Cryptol, functions can be defined by recursion over scalar (non-stream/vector) types. Static allocation is ensured by an affine type scheme that ensures streams are read at most once. In contrast to our model and that of the synchronous languages we discuss in §3.1, streams are not clocked: explicit handshaking is used to signal completion and demand more input. Under the typing constraints this allows arbitrary streams to be merged in finite space, whereas in the synchronous language Lustre the streams would need to be on the same clock.

The ongoing “geometry of synthesis” project of Ghica [Ghica 2007; Ghica et al. 2011] interprets a higher-order imperative language – a variant of Reynolds’s Idealised Algol – into various kinds of logic. It relies on Reynolds’s Syntactic Control of Interference as realised by an affine type system to eliminate conflicting writes to shared state. Unlike Johnson’s approach (§2.2) it is fully automatic.

Bluespec [Arvind and Nikhil 2008; Nikhil 2011] schedules sets of guarded commands into time slots where the actions are executed transactionally. It began with a syntax close to Haskell’s, with many of its structuring facilities, and has since adapted to the SystemVerilog and SystemC ecosystems while retaining many of its novel features.
3. RELATED WORK

Having reviewed the state-of-the-art in describing digital synchronous circuits as functional programs, we briefly discuss some areas that lie alongside ours: we point into the voluminous literature on synchronous programming languages and algebraic techniques for hardware description, consider the role of relational models, and sketch some of the issues with formal functional models.

3.1 Synchronous Languages

The synchronous programming languages have deployed similar ideas to those of sequential digital circuits to achieve deterministic concurrency in software, and reactive systems more generally. Berry [1999] argues forcefully for determinacy:

Nondeterministic systems are harder to specify, and it is not even trivial to define a good notion of behavior and equivalence for them, while execution traces are perfectly adequate for deterministic systems. Debugging nondeterministic systems can be a nightmare since transient bugs may not be reproduced. Analyzing systems is also much more difficult since the state space tends to explode. Therefore, it is important to reserve nondeterminism for places where it is really mandatory, i.e., interactive systems, and to forget about it for reactive systems. Historically, it was long thought that concurrency and non-determinism had to go together. [...] The main merit of synchronous languages is probably to have reconciled concurrency and determinism.

The DSLs for this class of systems that Berry [1989] called for are thoroughly surveyed by Benveniste et al. [2003]. Here we content ourselves with but a taste.

A central strand in this tradition is concerned with synchronous dataflow, or what might loosely be thought of as generalised circuits. The canonical such language is Lustre [Halbwachs et al. 1991] which extends the simple semantics of §1 with a notion of sampling: values can be present or absent at each instant. (In a constructive circuit all values are always present.) Clocks are used to statically guarantee that a signal is used only when it is present, which ensures that the corresponding Kahn network can be implemented with finite buffers [Caspi 1992]. Note that these do not coincide with a hardware designer’s notion of a clock as they need not be periodic. A variant of Lustre that included some constructs for expressing floorplans was proposed for hardware design [Rocheteau and Halbwachs 1991].

More recently there has been an effort to lift the features of ML to this synchronous dataflow paradigm. Higher-order functions have been treated by Caspi and Pouzet [1998] and Colaço et al. [2004], and pattern matching by Hamon [2006], resulting in the language Lucid Synchrone. Here clocks are formalised as types. The language also supports hierarchical state machines. The compiler can optionally ensure that a program has a finite-state implementation using a simple test that is sound but not complete. Caspi and Pouzet observe that this work connects synchrony to the deforestation techniques of Wadler [1990] for functional programs.

---

3An interactive system is one that takes control of the interaction. Berry cites operating systems, databases and the internet as examples.

The other main thread of the synchronous language tradition is the imperative paradigm as exemplified by Esterel [Potop-Butucaru et al. 2007]. Sequential and parallel composition are provided, and the usual battery of control constructs including loops and exception handling as well as some specialised ones such as preemption and suspension. Communication is provided by signals which are broadcast within a scope; in each instant they are either present or absent. A semantics of Esterel is given by translation into the constructive circuits that we discussed in \S 1, whose theory was developed for just this purpose.

The synchronous languages share many issues with hardware design. For instance, finite-state machines that are reactive (responding at every instant, also termed input enabled by process algebraists) or deterministic individually may in combination lose these properties [Maraninchi and Halbwachs 1996]. This issue is subsumed by the notion of causality, that of determining when a variable contains a valid value and what that value is. In the traditional circuit semantics of \S 1, causality is ensured by the dictum that “all loops must contain a delay”. (Similarly the notion of guardedness in process algebra is a causal notion [Milner 1989].) The clocks of the synchronous dataflow languages ensure this kind of safety while Esterel uses a specific analysis.

In contrast to behavioural synthesis, these languages are more predictable: timing behaviour is manifest in the source text, and all constructs are deterministic. As for circuits, the assumption of synchrony allows worst-case timing analysis to be performed separately from the logical design.

3.2 Algebraic Techniques

We briefly survey some algebraic approaches to describing circuits: the first two are in the tradition of process algebra, and the last algebraic specification. Where the functional programming techniques discussed earlier emphasise higher-level structure, these languages can be seen as providing alternative notation and semantics for the circuits themselves.

Cardelli and Plotkin [1981; 1982] adapted (what became) Milner’s SCCS [1983; 1989] into a “high level chip assembly language” – a notation for describing circuits and layouts purely structurally. This language is deeply embedded into ML, which serves as a metalanguage for composition and parametrisation. A continuous-time behavioural semantics for circuits is given at a much lower level than our synchronous one. Recently Park and Im [2011] have developed a linearly-typed higher-order functional notation for a similar purpose.

Milne [1985] developed the process algebra CIRCAL in the same tradition. It can describe both synchronous and asynchronous systems through the judicious introduction of non-deterministic choice. Due to its semantic neutrality it can be used at all levels of abstraction, which can be connected by refinement relations. More recently it has been extended to reconfigurable hardware [Milne 2006].

The FUNNEL compiler of Stavridou [1993] translates circuits expressed as recursion equations into the algebraic specification language OBJ, with the goal of specifying, simulating and verifying them. One could consider OBJ to be a functional programming language where higher-order functions have been sacrificed for very powerful reasoning principles, such as equational rewriting and fully-automatic proofs by induction. The ACL2 theorem prover used by Hunt Jr. and his collabora-
tors to verify various microprocessors has made a similar trade off [Hunt Jr. et al. 2010].

As OBJ itself is first-order, sequential behaviour was initially modelled as a global history, with sets of tuples of the form \((w, value, time)\) where \(w\) is some enumeration of wires, \(time\) is a natural number and \(value\) is a Boolean [Stavridou 1993, §4.3.3]. Later a mild extension to OBJ allowed the use of pseudo-second order functions, yielding “a powerful first-order calculus for reasoning about first-order functions” that could represent sequential behaviour directly. We note that both approaches preclude the use of circuit combinators (§2.1) as these are even higher-order.

Stavridou [1994] applied these techniques to “Gordon’s computer”, a standard example for mechanical verification of hardware, and also reviews other equational approaches to describing circuits.

3.3 Relational models

A reason to shift away from functions is to avail the designer of the traditional top-down program development methodology based on refinement [de Roever and Engelhardt 1998], where a specification is transformed into a more deterministic and detailed artifact expressed in the same language. Sheeran [1990] followed this train of thought when proposing a relational calculus of circuits called Ruby. Here combinational circuits and their specifications are taken to be strongly-typed relations on instantaneous values, with streams of such values used for sequential networks. As in \(\mu\text{FP}\), higher-order circuit combinators are given geometric interpretations.

The ultimate result of refinement in Ruby is a \textit{causal} relation, which are those that are functionally determined in a way familiar from database theory and logic programming: there must exist a partitioning of the fields of all relations into inputs and outputs where the latter is determined by the former. This excludes the bidirectional dataflow of busses and MOS circuits which are naturally modelled relationally. The T-Ruby system of Sharp and Rasmussen [1997] can simulate and generate synthesisable VHDL for this subset.

Ruby has been applied to similar systems as \(\mu\text{FP} –\) regular and arithmetic circuits [Jones and Sheeran 1993], and innovatively, butterflies such as FFTs. However as we saw with \(\mu\text{FP}\), the purely combinatory style can make for awkward descriptions. Indeed the Lava approach, with its extensive battery of testing and verification tools and ad hoc combining forms, has shown that supporting exploration with instant feedback trumps formal dexterity during the design process. The recent Wired project [Axelsson et al. 2005] combines these themes in a language for capturing very low-level properties of chip design.

We note the extensive literature on modelling circuits in a higher-order logic [Camilleri et al. 1986] (etc.) but it takes us too far afield to review it here.

3.4 Other models of “boxes and wires”

Another mode of generalisation is to focus on general ways of composing “boxes and wires” diagrams, and investigate their equational properties. Category theorists claim that these find their natural expression as some kind of monoidal category, and indeed these structures and their “string diagrams” have been surveyed at length by Selinger [2011].

These models are constructed using combinators, and therefore suffer from the
plumbing problem. Braibant [2011] models circuits in the Coq proof assistant using such an approach and it is clear that while the algebra is pleasant one would struggle to comprehend the syntactic expression of a circuit without an accompanying diagram. This tension has been substantially resolved for a particular set of combinators – the Arrows of Hughes [2000] – by the notation of Paterson [2001], which allows us to write pointwise or point-free definitions at our discretion. It remains to be shown what advantage Arrows have over monads for describing circuits.

The Hume project has developed a “box calculus” [Grov and Michaelson 2010] that supports the refinement of computational boxes connected by wiring described in a finite-state coordination language.

3.5 On formal functional models for synchronous digital circuits

To reason about our circuits using a proof assistant, we need an accurate formal model for them. Here we discuss a few of the traditional models.

In general we wish to reason in two ways. Firstly we would like to transform our circuits using equational reasoning, and as we saw above the domain models support this mode very well; such techniques scale easily as they are largely independent of the size of the state space. Secondly we wish to show that particular circuits have specific properties, for which temporal logic in general [Manna and Pnueli 1992], and its automation in the form of model checking [Clarke et al. 1999], has proven very successful. However as observed by Matthews [2000, §7.6], by encapsulating state our stream models sometimes make assertions more difficult to write than in explicit-state formalisms. Day et al. [2000] discuss moving between these representations for a shallowly-embedded HDL.

Most systems we discuss here implicitly appeal to the synchronous isomorphism:

\[
\text{Signal} (\alpha, \beta) \simeq (\text{Signal} \alpha, \text{Signal} \beta)
\]

where \text{Signal} \alpha is a type that captures the temporal behaviour of a wire. Intuitively this characterises systems with non-blocking components that communicate in globally-synchronised rounds; it requires functions \text{Signal} \alpha \to \text{Signal} \beta to be length preserving, which clearly does not hold in asynchronous settings.

This isomorphism underpins laws that allow stateful components to be combined and decomposed, such as the one shown in §2.1. As we observed there, our \text{Signal} \alpha domain of Figure 1 does not not satisfy this isomorphism as it contains junk in the form of partial streams \( x_0 :> \ldots :> x_n :> \bot \), where \( \bot \) is the least-defined sequence [Winskel 1993, §8.2]. These preclude the definition of an injective \text{zip}. We note that Kahn networks and other domains based on prefix orders have the same deficiency.

While preferring this model, Caspi [1992] observes we could also take \text{Signal} \alpha to be some set of functions \text{nat} \to \alpha, which supports the operations of Figure 1 while satisfying the synchronous isomorphism. (This is an environment or reader monad.) Unfortunately it also admits junk in the form of the non-causal functions \text{Signal} \alpha \to \text{Signal} \beta whose behaviour at time \( n \) depends on the value of their arguments at time \( m > n \). Abbott et al. [2005] have studied these containers in categorical and type-theoretic settings; see also Bertot and Komendantskaya [2008].

This attempt to identify \text{Signal} \alpha with the set of causal infinite streams over \alpha suggests the use of corecursion [Coquand 1993]. Such an approach was advocated
by Paulin-Mohring [1995] who used the support for coinductive types built into the Coq proof assistant to describe a multiplier and establish some properties of it.

Caspi and Pouzet [1998] show how to compose corecursive descriptions from systems of recursion equations for their higher-order synchronous dataflow language Lucid Synchrone (see §3.1), but it is unclear that it can be used in proof assistants where corecursive definitions are typically required to take particular syntactic forms. Such constraints guarantee productivity of the definition and hence well-definedness of the sequence. Note these also rule out the higher-order combinators of §2.

The literature on models of dataflow and streaming computation is too vast to review here; we only point to some closely related recent work. Hughes et al. [1996], Barthe et al. [2004] and Abel [2010] propose sized types as a compositional way of ensuring productivity. The “fast and loose reasoning” of Danielsson et al. [2006] does not apply to unstructured recursion equations, though some may consider a unique fixed-point property [Hinze and James 2011] to be something of a replacement; see also the work of Matthews [1999] mentioned in §2.7. Broy and Stølen [2001] use prefix-ordered domains to specify interactive systems. Möller and Tucker [1998] provide further pointers to formal stream-based models for hardware.

4. CONCLUDING REMARKS

Here we have focused on surveying how functional programming has been used to describe, design and validate synchronous hardware. Jantsch and Sander [2005] situate this model of computation in a spectrum of those relevant to the construction of embedded systems, including the codesign of hardware and software. The reader can find surveys of HDLs in other styles in McEvoy and Tucker [1990b], Stavridou [1993, Chapter 3] and Claessen [2001, §1.11], while Johnson [1983, Chapter 1] and Sheeran [2005] provide more historical perspective on the early days of this tradition. Sharing in EDSLs is discussed at length by Kiselyov [2011].

The central goal of all of these systems is to make higher-assurance hardware easier to design, and to find a good trade-off between formal rigour and ease of use. This is a problem of increasing interest as FGPAs and other reprogrammable logic becomes commonplace [Cardoso et al. 2010], and the development of custom hardware structures for computation kernels, or even coprocessors like the Reduceron (§2.6.2), is not always worth the cost of full-blown verification using proof assistants. Hope may lie in automatic state-space traversal techniques [Clarke et al. 1999], but these too require expertise quite distant from hardware design. Random testing as epitomised by QuickCheck [Claessen 2001, Chapter 4] is an alternative that works well when effects can be tamed, as they are in a purely functional setting.

In contrast proof assistants are essential to the verification of complex designs and the refinement processes advocated by Johnson [2001], and indeed Intel’s Integrated Design and Validation (IDV) system appears to have successfully applied this methodology to their designs [Seger et al. 2005; Grundy et al. 2006], though perhaps not as ambitiously as Johnson aspired to. Functional programming techniques underpin all large-scale verification efforts such as the ARM processor models of Fox et al. [2010] and the x86-compatible models of Hunt Jr. et al. [2010].

The systems presented above are all experimental, both in their methodology and the artifacts described with them. Sheeran [2011] has used her various platforms...
to explore different kinds of circuits, and shown that rapid feedback in the form of simulation, testing and model checking is most valuable to the designer. Johnson and Bose [1997] and Seger et al. [2005] make similar observations about their refinement efforts. This is clear evidence that functional programming techniques are a useful substrate for this diverse range of tasks.

The algebraic structure of circuits has much in common with other forms of parallel and distributed programming, which also use parallel prefix (or scan) networks [Sheeran 2011], and butterflies and other networks that are naturally rendered using powerlists [Paterson 2003]. These structures link our domain with the search for higher-level programming abstractions for historically arcane DSP and GPU architectures [Sweeney 2009; Axelsson et al. 2010; Chakravarty et al. 2011] and multicore systems [Keller et al. 2010]. Singh [2007] also proposes adopting concurrency abstractions developed in functional programming settings to hardware.

As we discussed in §2.2 and §2.10, functional programming has been used as a basis for behavioural synthesis. Recently Harrison et al. [2009] propose to extend Johnson’s use of Wand’s compiler/virtual machine split (§2.2) to a concurrent language by using a resumption monad; every element of this agenda poses difficulties for other programming techniques due to their lack of types, higher-order facilities or controlled effects.

Another quintessential dimension of this tradition is the development of increasingly fancy type systems [Kaes 1988; Wadler and Blott 1989; Chakravarty et al. 2005; Diatchki et al. 2005; Peyton Jones et al. 2007] (etc.) that are comfortable to program with. Such techniques have already been shown useful for parametrising circuit generators by vector widths (§2.6.3). Sheard [2007] proposes his Omega language as a vehicle for exploring the use of this machinery in great generality; one eventually might hope to write circuit generators as resource-aware active libraries [Veldhuizen 2004; Sheeran 2011].

Sheard also argues that HDLs should formally recognise the distinction between circuits and their generators; in other words, the staging of descriptions should be manifest, which is certainly necessary to resolve the semantic tensions we saw throughout §2. Kiselyov et al. [2004] and Gillenwater et al. [2010] demonstrate how this idea works in practice. Megacz [2011] is pursuing an approach in which two-level programs with first-order object expressions are flattened into single-level programs which represent object language terms using a generalization of the Arrow programming abstraction due to Hughes [2000].

We also find an argument for meta-programming from the formal reasoning community, where Grundy et al. [2006] have developed two functional languages for representing circuits in a higher-order logic. These involve reification of descriptions into the logic, and not just execution; while this leads to semantic difficulties in a programming setting [Taha 2000], it is quite desirable in a proof assistant.

The limited domain of circuits and fixed-network stream processors often admits appealing diagrammatic representations which can be much easier to reason about than the expressions they visualise, as we saw in §2.1. This is not too surprising as effective circuits need to be mapped to floorplans. What is surprising is that while semantically-wellfounded graphical tools for first-order languages abound [Harel 2009; André and Peraldi-Frati 2000; Maraninchi and Rémond 2001] (etc.), there is a
lack of support for the kind of higher-order programming advocated here.

In closing we observe the renewed interest in functional programming techniques for software due to the increasing use of parallelism and concurrency, and expect to see a similar resurgence in the context of hardware design.

REFERENCES


Fox, A. C. J., Gordon, M. J. C., and Myreen, M. O. 2010. Specification and verification of ARM hardware and software. See Hardin [2010].


Mead, C. and Conway, L. 1980. Introduction to VLSI systems. Addison-Wesley, Reading, MA.


Peter Gammie


