Towards Type-Safety in asynchronous Hardware Description

David Lanzendoerfer, Andreas Westerwick

o2s - OpenSourceSupport GmbH

mail@o2s.ch

3180 YOLD

Abstract

*Circuitry written in commonly used hardware description languages like VHDL are exceptionally hard to formally verify and remind us of assembler-type languages, lowest possible instructions/gates enriched with some artificial abstraction. In this paper we show a more natural abstraction, namely from circuits to categories [1] and use the purely-functional language Haskell to embed a domain specific language for digital hardware description. Through implementing LavaArrows in Haskell, we get an intuitive arrow notation [**?**] for free as well as type declarations that may show proofs of your circuit's formal correctness [**?**].*

I. Introduction

This is VHDL gone wrong and we can fix it.

II. Contributions of this work

How can we solve all these problems inherited by the language your hardware producer provides to describe hardware? These are our contributions:

- A mathematically sound DSL for digital hardware description, as sub-language of Haskell, utilizing the notion of arrows founded in category theory.
- Deprecating traditional 'testbenches' and hardware verification languages through formal verification and turing-complete testing in Haskell (e. g. QuickCheck [**?**]).
- Blazingly fast and ordered synthesis compared to written VHDL code.

[1] No prior knowledge of category theory required to read further

III. Categories of circuitry

Now follows the founding ideas of LavaArrows.
Consider a simplified schematic of a curcuit:

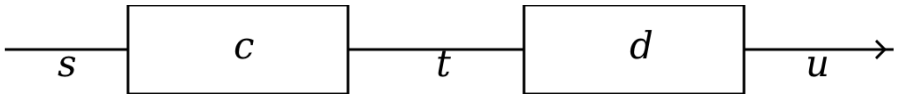

Figure 1: *A very simple circuit: c and d are components*

When you view *s t* and *u* not as pieces of wire, but as digital states, you might want to draw a different picture:

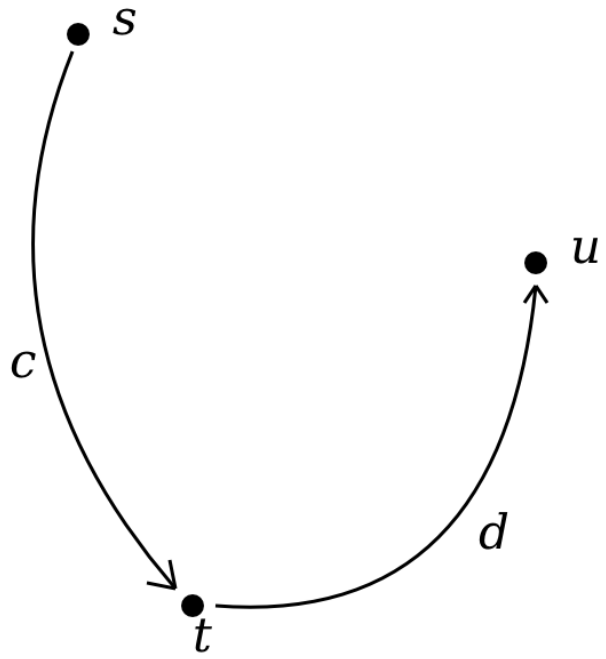

Figure 2: *A very simple category: c and d are morphisms*

Components are now arrows from wire states to wire states, they take zero or more wires and put zero or more wires out.
This corresponds to a 1-category, a category that hosts objects and morphims from one object to another, where:

- There is a morphism that goes from one object to itself.
- There is a notion of chaining two morphisms together, associatively.

The identity morphism corresponds to no component built, the wire state does not change; Describing a component that does nothing at all is like having no component at all.
Chaining two morphisms, which in our specific category are called components, can be done through wiring them together and not give a name to the piece of wire in between:

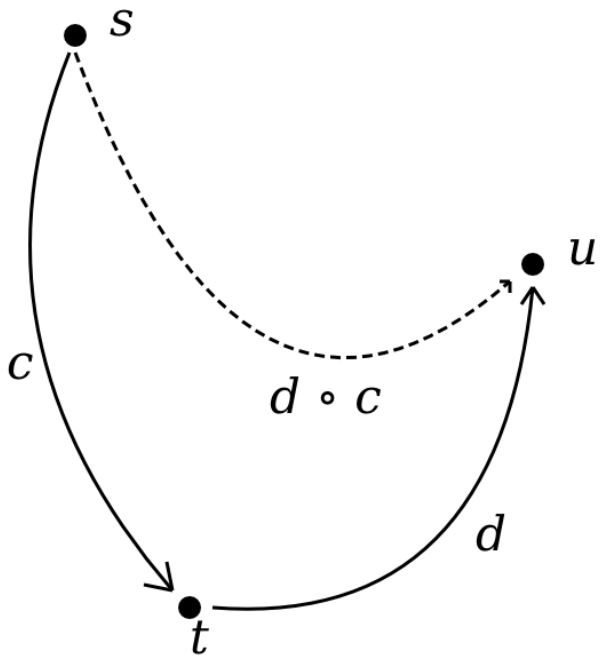

Figure 3: *Composition: c and d chained together give rise to a new morphism*

Now that we have shown an isomorphism from a synchronous circuit to a category with logical components as morphisms, it follows that your whole circuit is indeed an arrow by chaining all components of your circuit together to form a single large component.

We cannot view components as mathematical functions purely, since each component or rather its primitives need to be synthesized. That means our language is able to alter wire states, which will come handy for testing purposes, but cannot yet express synthesis.
What our arrows lack is effects on the netlist for synthesis.
Fortunately, Haskell provides us with an arrow type class that can invite effects [**?**]:

```
class Arrow a where
  pure  :: (s → t) → a s t
  (>>>)  :: a s t → a t u → a s u
  first :: a s t → a (s, u) (t, u)
```

From above there are three axioms we must satisfy to show that components are arrows:

- We must show that for all s and t any purely mathematical function from s to t can be made a component with input s and output t
  This is simply a function changing wire states without having any effect on the netlist, where "wire states" is anything you can come up with.
- There is a notion of chaining two components together; We already showed that components can be seen as categories, which gives us composition for free (Figure **??**):

```
c >>> d = d ∘ c
```

- For all components there is a notion of bypassing said component for all u
  In other words we are having full effects on our netlist but only changing the wire state in the first tuple element, ignoring the second completely. From 'first' - especially in combination with the other two axioms we can become member of other interesting type classes, such as Applicative [**?**].

Since arrows can capture any effect on netlists, asynchronous circuits are now possible as long as the underlying primitive description language (e. g. VHDL) allows for asynchronous constructs.

We can have effects on the netlist for our circuitry but how these work concretely depends on the synthesis itself. Another architecture may allow new kinds of synthesis that would not work before: In Section IV we will discuss the synthesis of Xilinx' VHDL.

IV. LavaArrows: The Language

LavaArrows depends on the package 'xilinx-lava', a library for generating circuits for Xilinx FPGAs with layout [**?**], for generating netlists and primitive gates.
Here is the type declaration of a Lava primitive from this package:

```
and2 :: (Bit, Bit) → State Netlist Bit
```

Most Lava primitives from this package operate in the lazy state monad to build the netlist. We want to use arrows instead so that we end up with type declarations like this, where the effects are captured by the arrow, not the return type:

```
and2 :: (Bit, Bit) ↠ Bit
```

I.

example 1

V. Hardware Verification in LavaArrows

VI. Comparison with related work

Table 1: *Synthesis benchmark*

VII. Outstanding work

References

[Cla00] Koen Claessen and John Hughes,
QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs
in International Conference on Functional Programming (ICFP).
2000

[Hug98] John Hughes, *Generalising Monads to Arrows* in Science of Computer Programming. November 1998

[Pat01] Ross Paterson, *A New Notation for Arrows*
in International Conference on Functional Programming (ICFP).
September 2001

[Pat03] Ross Paterson, *Arrows and Computation* in The Fun of Programming,
edited by Jeremy Gibbons and Oege de Moor, Palgrave.
2003, 201-222

[Pat08] Conor McBride and Ross Paterson, *Applicative Programming with Effects*
in Journal of Functional Programming.
2008

[Sig13] Satnam Singh, *The xilinx-lava package*,
`http://hackage.haskell.org/package/xilinx-lava`
Semptember 2013,
version 5.0.1.8

[Wa14] Philip Wadler, *Propositions as Types*. updated June 2014