Syrup

a programming language for digital circuits

Conor McBride

October 6, 2018

1 Introduction

Syrup is a small domain-specific programming language designed for digital circuits. It is intended mainly for use in the teaching and assessment of the digital logic components of CS106 *Computer Systems and Organisation* and CS111 *Computational Thinking* at the University of Strathclyde. By moving from circuit diagrams on paper to circuits programmed in Syrup, we hopefully facilitate improved automation of feedback.

In previous years, we used an ascii-art circuit language called CVI (the Circuit Validation Interpreter, and also 106, of course). Circuits looked like diagrams, but drawing ascii art in an ordinary text editor is tricky, even if you have picked up the knack. It is time to move on.

2 Overview

Circuits are made from components. Components are made from other components. Syrup is a language for defining new components from existing components.

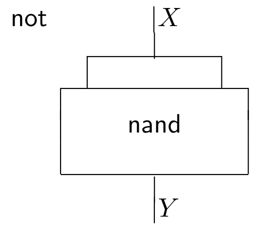
A component should have

- a **declaration**, which gives it a unique name and specifies its external connectivity
- a **definition**, which explains how it is constructed.

For example, let us build a 'not' gate from a 'nand' gate.

```
not(<Bit>) -> <Bit>
not(X) = Y where
  Y = nand(X,X)
```

This renders in textual form the diagram (which was a nuisance to typeset).



The way you *use* a component is

$$name(input_1, \ldots input_n)$$

e.g., `nand(X,X)`.

A component declaration names the component and explains what *types* of signals we expect on each of the input and output wires. Left of the `->` arrow, we give the name and input types, made to look like part of a grammar. There are different types of signals in Syrup, just as in real life we use different types of plugs and sockets for your headphones and for mains electricity. The simplest type of signal is

```
<Bit>
```

When we write `not(<Bit>) ->`... We are saying '`not(`e`)` makes notational sense, whenever e is meaningful as a bit', as well as '`not` has one input and that input is a bit signal'. Right of the `->` arrow, we say what signals come *out* of the component. In this case, we also expect a bit.

The declaration tells us nothing about what is *inside* the component, but already it tells us how we would plug things together with it.

It is the *definition* which tells us how the component computes its output from its input. A definition is a system of equations.

$$\textit{head}\ \texttt{where}\ \textit{body}_1\ \ldots \textit{body}_n$$

The 'head' equation deals with the external connectivity of the component. Its left hand side, `not(X)`, stands for any use of the component, and it names the input, `X`, corresponding to the wire labelled X in the diagram, for future reference. Its right hand side, `Y` indicates which signal is the output.

In general, when we write an equation in a Syrup program, its left hand side is an act of *naming*, and its right hand side makes *use* of named things.

Specifically, the input `X` has been named in the head equation, and the output `Y` is being used, which means that we rely on the 'body' equations to tell us what `Y` is. And that happens: we have a body equation for each component inside the circuit, showing how it is wired. The equation

```
Y = nand(X,X)
```

tells us what that `Y` is, namely the output from a `nand` gate, whose inputs are both connected to `X`, the input of the circuit.

So, every *wire* in the diagram (X, Y) corresponds to a *variable* in the program (`X`, `Y`), and every *component* in the diagram (the `not` we are building and the `nand` we are using) corresponds to a *function* in the program (`not`, `nand`). The equations in the program tell us how the components are wired up.

It may seem strange to bring things into existence by telling stories about them that we hope are true, but that is the nature of existence.