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.

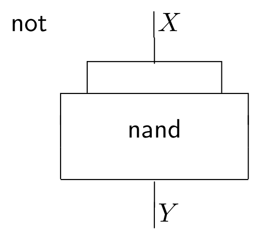
A component should have

- 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 in the nature of existence.

3 Syntax

Syrup is a formal notation with rules that must be followed if the machine is to make sense of your program.

A *program* is a sequence of *statements* which are written in a file or in a text area of a web browser.

Indentation The first line of a statement must have no whitespace at the beginning: it is anchored to the left. Subsequent lines of the statement must be *indented* by at least one space, to show that they are part of the same statement and not a whole new statement.

Statements These come in three varieties:

declarations introduce components by naming them and specifying their external connectivity (how many wires come in, how many come out, and what types of signals are carried on those wires) — as it were, what's 'outside the box'

definitions explain how components are constructed (the explanation consists of a system of equations which relate components and wires)

experiments request data about components in actual use

The declaration of a component must precede its definition. Each component should have at most one definition. A component with no definition is called a *stub*. Syrup will let you build up new components from old stubs, by way of planning, but experiments involving stubs are likely to prove inconclusive.

Names Components and wires get names. In Syrup, a valid name begins with a letter and is followed by zero or more letters or digits. Some names are not permitted because they are *keywords* and have special meaning in Syrup, e.g., `where` and `experiment`.

We use names to tell things apart. Different components must have different names: those names are *global*. Within each component, the different wires must have different names. However, the names of the wires are *local* to the definition of a component: they make sense within that one definition, but they are not meaningful outside the definition. In particular, you may use the same name for a wire in the definition of one component as you use in the definition of another. We will call a lot of wires `X`.

Using Components The notation for using a component in a declaration or a definition is to give its name, followed by a set of parentheses `( )` containing the component's inputs, separated by commas. What are the inputs? In a declaration, we write the *signal types* of the inputs. On the left hand side of a head equation, we write *patterns* which give names to the inputs. On the right hand side of an equation (head or body), we write *expressions* which explain how the inputs are to be computed.

Signal Types The simplest signal type is `<Bit>`, the type of one-bit signals. We can build more complex signal types by collecting signals together in *cables*. A cable type is a comma-separated list of zero or more signal types sheathed in square brackets the way actual cables wrap bundles of wires in rubber, e.g.,

```
[<Bit>, <Bit>, <Bit>, <Bit>]
```

is a cable containing four one-bit signals (perhaps representing a single hexadecimal digit). Cable types allow us to refer to a bunch of signals tidily by a single name, thus simplifying more complex constructions. The cable type

```
[<Bit>]
```

is not the same thing as

```
<Bit>
```

in that the former has an extra layer of sheathing. Cable types can contain smaller cable types, e.g.,

```
[[<Bit>, <Bit>], [<Bit>, <Bit>]]
```

is the type of cables which contain two cables, each of which contains two one-bit wires. Meanwhile, the cable type

```
[]
```

characterizes a layer of rubber sheathing with no signal-bearing copper inside.

$$
\begin{array}{lcl}
\langle \textit{type} \rangle & ::= & \texttt{<Bit>} \\
 & | & \texttt{[}\langle \textit{typeList} \rangle\texttt{]} \\
\langle \textit{typeList} \rangle & ::= & \\
 & | & \langle \textit{types} \rangle \\
\langle \textit{types} \rangle & ::= & \langle \textit{type} \rangle \\
 & | & \langle \textit{type} \rangle\texttt{,}\langle \textit{types} \rangle
\end{array}
$$

Declarations A declaration looks like

⟨*name*⟩`(`⟨*typeList*⟩`) ->` ⟨*typeList*⟩

That is, we have the component name, then its input signal types listed in parentheses, then an arrow, then the list of output signal types.

Definitions A definition looks like

⟨*name*⟩`(`⟨*patternList*⟩`) =` ⟨*expressionList*⟩ `where` ⟨*equations*⟩

The *patterns* explain how to analyse and name the inputs. The expressions explain how to generate the outputs. The equations look like

```
⟨equations⟩  ::=
              |  ⟨patternList⟩ = ⟨expressionList⟩ ⟨equations⟩
```

A definition may omit `where` if it has zero equations. Each equation has patterns which name internal signals, and expressions which explain how to compute them.

Patterns The simplest pattern is just a variable, naming a signal. A variable can stand for a one-bit signal or the signal transmitted on a whole cable. However, we may also write cable patterns, allowing us to unsheath the individual signals from inside a cable:

```
⟨pattern⟩      ::=  ⟨name⟩
                 |  [⟨patternList⟩]

⟨patternList⟩  ::=
                 |  ⟨patterns⟩

⟨patterns⟩     ::=  ⟨pattern⟩
                 |  ⟨pattern⟩,⟨patterns⟩
```

Within any given component, all the names in all the patterns must be different from each other.

Expressions The simplest expression is just a variable. We also have uses of components, and we have cable expressions, which sheathe a bunch of signals generated by a list of expressions.

```
⟨expression⟩      ::=  ⟨name⟩
                    |  [⟨expressionList⟩]
                    |  ⟨name⟩(⟨expressionList⟩)

⟨expressionList⟩  ::=
                    |  ⟨expressions⟩

⟨expressions⟩     ::=  ⟨expression⟩
                    |  ⟨expression⟩,⟨expressions⟩
```