Syrup

a programming language for digital circuits

Conor McBride

October 24, 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.

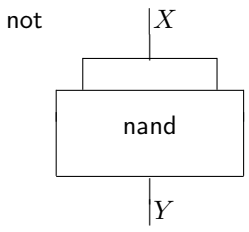
A component should have

- 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.

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

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.

Comments If Syrup sees two consecutive minus signs, like this `--`, it disregards them and the rest of the text on that line. You can now add textual explanations to your code!

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.

We can use a formal grammar to be precise about which signal types exist: John Levine teaches grammars because we really use them.

⟨*type*⟩ ::= `<Bit>`
| `[`⟨*typeList*⟩`]`

⟨*typeList*⟩ ::=
| ⟨*types*⟩

⟨*types*⟩ ::= ⟨*type*⟩
| ⟨*type*⟩`,`⟨*types*⟩

That is, ⟨*typeList*⟩ is the syntax of possibly empty lists of types which are *separated* by commas when there are two or more types in the list.

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. Conveniently, though, it is not compulsory to unpack cables all the way down to the individual bit wires. A name can stand for a cable. That makes it sensible to build components which group related inputs into cables.

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.

$$\begin{array}{lcl}
\langle expression \rangle & ::= & \langle name \rangle \\
& | & \texttt{[}\langle expressionList \rangle\texttt{]} \\
& | & \langle name \rangle\texttt{(}\langle expressionList \rangle\texttt{)} \\
\langle expressionList \rangle & ::= & \\
& | & \langle expressions \rangle \\
\langle expressions \rangle & ::= & \langle expression \rangle \\
& | & \langle expression \rangle\texttt{,}\langle expressions \rangle
\end{array}$$

Special Syntax for Boolean Connectives Syrup supports the notations

- $!X$ for $\mathsf{not}(X)$
- $X\&Y$ for $\mathsf{and}(X, Y)$
- $X|Y$ for $\mathsf{or}(X, Y)$

with the usual operator priorities: ! above & above |, so that you can write disjunctive normal forms with no parentheses:

$$X\&Y \mid !X\&!Y$$

means

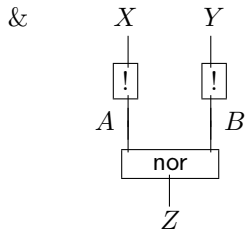
$$(X\&Y) \mid ((!X)\&(!Y))$$

means

$$\mathsf{or}(\mathsf{and}(X, Y), \mathsf{and}(\mathsf{not}(X), \mathsf{not}(Y)))$$

4 How to Turn Hardware Into Syrup

Suppose you have a circuit diagram like this ('and' constructed from 'not' and 'nor'):



How do you turn that into Syrup? Make sure you have given each wire in the diagram a different name, including the internal wires. Now, there are three basic steps.

1. Write the *declaration* on a new line, starting at the far left. That establishes the external connectivity of the component. Here, we have two bits in and one bit out.

   ```
   <Bit> & <Bit> -> <Bit>
   ```

   We could also have written

   ```
   and(<Bit>,<Bit>) -> <Bit>
   ```

   meaning the same thing in less fancy notation.

2. On another new line, starting at the far left, write the *head equation* of the component's *definition*. That names the inputs and outputs: it should match up with the definition. Follow the equation with `where`.

   ```
   X & Y = Z where
   ```

3. Now, on subsequent lines, *indented by at least one leading space*, write the body equations. You should have one equation for each component used in the diagram, showing that the outputs come from the component fed by the inputs.

   ```
   X & Y = Z where
     A = !X
     B = !Y
     Z = nor(A,B)
   ```

   The indentation tells Syrup that you are continuing with the definition of `X & Y`, not starting a new thing.

So, the declaration states the signal types that the component accepts, the head equation deals with the wires coming into and going out of the component, and the body equations show how to wire up all the components that are used inside.

Some variations are permitted, and these can save time and space. The body equations can come in any order (with one caveat for circuits involving memory). You can also combine two equations into one by listing the left hand sides and listing the right hand sides: that is helpful for comprehension when two components correspond to one high level step, e.g. 'negate both inputs':

```
X & Y = Z where
  A,B = !X,!Y
  Z = nor(A,B)
```

Moreover, if you have an equation defining a variable

- as the single output of a component
- used exactly once

then you can substitute the component expression for the variable and get rid of the equation. We can have

```
X & Y = Z where
  Z = nor(!X,!Y)
```

and then

```
X & Y = nor(!X,!Y)
```

with no `where` as there are no body equations to introduce. The upshot is that if you can define a one-output component by a Boolean expression, you can just write that expression on the right of the = in the head equation and be done with it.

5 Definitions with Multiple Outputs

Some components, notably adders, have more than one bit of output. That does not bother Syrup. E.g., a 'half adder' has two input bits and two output bits, so

```
hadd(<Bit>,<Bit>) -> <Bit>,<Bit>
```

Both inputs are used to compute both outputs, thus:

```
hadd(X1,Y1) = C2,S1 where
  C2 = X1 & Y1
  S1 = xor(X1,Y1)
```

Variables have one *definition* site in a pattern: every wire carries signal from *one* place. Variables may have any number of *use* sites in an expression: wires may carry signals to zero or more places. Note that it is a healthy habit to give names to wires in arithmetic circuits which document their *place values*.

Now we may use our half adder to make a 'full adder'.

```
fadd(<Bit>,<Bit>,<Bit>) -> <Bit>,<Bit>
fadd(X1,Y1,C1) = C2,Z1 where
  D2,D1 = hadd(X1,Y1)
  E2,Z1 = hadd(D1,C1)
  C2 = xor(D2,E2)
```

Notice how the equations where we use hadd have one expression on the right, but two pattern variables on the left: that is one for each of the two outputs.

It is permitted, as we did with the two nots above, to put multiple expressions each with multiple outputs on the right of an equation, as long as the total number of patterns on the left matches up with the total number of outputs on the right. However, the resulting code is very likely to be confusing, as it will be hard to tell which things come from which components.

Likewise, you may invoke a multiple output component anywhere that a list of that number of inputs is expected. E.g., if we have

```
swap(<Bit>,<Bit>) -> <Bit>,<Bit>
swap(X,Y) = Y,X
```

then we may write expressions like

```
mux(C,swap(X1,X0))
```

feeding the mux its second and third outputs from swap. Again, doing so may reduce the number of wires that need names at the cost of some clarity.

6 Definitions with Cables

Syrup has cables to keep wires tidy. Suppose we want to add two 2-bit numbers and a carry-in, producing a 2-bit sum and a carry-out. That would make five bits in and three bits out, but these belong in particular groups.

```
adc2(          [<Bit>,<Bit>],
               [<Bit>,<Bit>],
                      <Bit>)
     -> <Bit>,[<Bit>,<Bit>]
```

Each of our 2-bit numbers is grouped in a cable, as indicated by the square brackets, leaving just the carry bits unpacked. I have also laid out the declaration using space to show intended place values in columns. As I did so, I took care to ensure that all but the first line were indented, so that Syrup knows the multiple lines are all part of the same declaration.

In the resulting definition, I can use square bracket syntax in patterns to access the wires inside the cables, and in expressions to group outputs into cables.

```
adc2(      [X2,X1],
           [Y2,Y1],
               C1)
    =   C4,[Z2,Z1] where
  C2,Z1 = fadd(X1,Y1,C1)  -- add the 'ones'
  C4,Z2 = fadd(X2,Y2,C2)  -- add the 'twos'
```

Now, I could follow the same pattern to define a 4-bit adder, like this:

```
adc4(         [<Bit>,<Bit>,<Bit>,<Bit>],
              [<Bit>,<Bit>,<Bit>,<Bit>],
                                 <Bit>)
     -> <Bit>,[<Bit>,<Bit>,<Bit>,<Bit>]
adc2(      [X8,X4,X2,X1],
           [Y8,Y4,Y2,Y1],
                     C1)
    =  C16,[Z8,Z4,Z2,Z1] where
  C2 ,Z1 = fadd(X1,Y1,C1)  -- add the 'ones'
  C4 ,Z2 = fadd(X2,Y2,C2)  -- add the 'twos'
  C8 ,Z4 = fadd(X4,Y4,C4)  -- add the 'fours'
  C16,Z8 = fadd(X8,Y8,C8)  -- add the 'eights'
```

but there is a neater way to solve the problem: use *nested* cables to group each 4-bit number into two groups of 2-bit numbers: the 'big end' and the 'little end'. We may now use adc2 to add more in one go. As long as we carry from the little end to the big end, everything will work.

```
adc4(         [[<Bit>,<Bit>],[<Bit>,<Bit>]],
              [[<Bit>,<Bit>],[<Bit>,<Bit>]],
                                   <Bit>)
     -> <Bit>,[[<Bit>,<Bit>],[<Bit>,<Bit>]]
adc4(      [X84,X21],
           [Y84,Y21],
                C1)
    =  C16,[Z84,Z21] where
  C4, Z21 = adc2(X21,Y21,C1)  -- add the 'twos and ones'
  C16,Z84 = adc2(X84,Y84,C4)  -- add the 'eights and fours'
```

Notice, in particular, that we did not have to unpack cables all the way down to the individual bits. Instead, we unpacked just enough to get the cables expected by adc2. As a result, the definition of adc4 has the same structure has that of adc2: add the little ends with the carry=in, send the middle carry into the big end adder, collect the outputs and send on the carry-out.

(If you are asking yourself "Is there any way I can use one piece of Syrup code to make both definitions?", then you are asking a very sensible question. The answer, for the moment, is "No.", but that answer may change in future.)

7 Experiments

As well as allowing you to declare and define components, Syrup allows you to perform experiments on them, to see if they do what you want.

Truth Tables The simplest form of experiment is to ask for the *truth table* of a known component. You do that with a line that says

`experiment` ⟨*name*⟩

E.g., if you have defined xor correctly, then

```
experiment xor
```

should produce

```
Truth table for xor:
00 | 0
```

```
01 | 1
10 | 1
11 | 0
```

Likewise, given a correct fadd,

```
experiment fadd
```

should produce

```
Truth table for fadd:
000 | 00
001 | 01
010 | 01
011 | 10
100 | 01
101 | 10
110 | 10
111 | 11
```

Notice how the input columns contain no commas, and are separated by a vertical line from the output columns.

If you want the truth tables for the Boolean connectives, you must call them by their 'official' names, `not`, `and`, and `or`, rather than using the operators `!`, `&` and `|`.

If you ask for a truth table for a component involving cables, the table shows the cable structure, but still no commas. For adc2,

```
experiment adc2
```

should produce

```
Truth table for adc2:
[00][00]0 | 0[00]
[00][00]1 | 0[01]
[00][01]0 | 0[01]
[00][01]1 | 0[10]
[00][10]0 | 0[10]
[00][10]1 | 0[11]
[00][11]0 | 0[11]
[00][11]1 | 1[00]
[01][00]0 | 0[01]
```

⋮

```
[10][11]1 | 1[10]
[11][00]0 | 0[11]
[11][00]1 | 1[00]
[11][01]0 | 1[00]
[11][01]1 | 1[01]
[11][10]0 | 1[01]
[11][10]1 | 1[10]
[11][11]0 | 1[10]
[11][11]1 | 1[11]
```

(I left out the middle of the sequence, because it is a large table.) Notice how the square brackets show the grouping of the bits in their cables.

Unit-Testing with Time Sequences Of course, full truth tables can get very large. E.g., for adc4, we would expect 512 lines! Fortunately, we can also try *unit-testing* components. If we write

```
experiment adc4([[10][10]] [[01][10]] 0;
                [[11][11]] [[00][11]] 1)
```

then we should get

```
Simulation for adc4:
0 {} [[10][10]][[01][10]]0 -> 1[[00][00]]
1 {} [[11][11]][[00][11]]1 -> 1[[00][11]]
2 {}
```

What just happened? I fed adc4 a *time sequence* of inputs. We consider steps in time, numbered from 0. At each step, I can feed a component a different input. I present a time sequence as a bunch of valid inputs, separated by *semicolons*: each such input is a load of bits in cables, fitting with the component's expected inputs. Here, I gave two steps, each with two 4-bit numners and a carry-in: at time 0, I asked for 10+6+0; at time 1, I asked for 15+3+1.

The output tells me what happened at each step: at time 0, out came 16; at time 1, out came 19; at time 2, Syrup ran out of input.

The curly braces in the output show how the component's *memory* evolves with time. Here, however, they are empty because adc4 does not make use of memory, so what happens at each time step is independent of history.

When are Components the Same? You can ask if two components are the same by adding the line

`experiment` ⟨*name*⟩ `=` ⟨*name*⟩

giving the names of the components. If the components have the same externally observable behaviour, Syrup will confirm it — the exact definitions of the components can differ if they respond to inputs in the same way.

For example, we can define xor in two different ways, then check that the two definitions do the same job.

```
xor1(<Bit>, <Bit>) -> <Bit>
xor1(X,Y) = !X & Y | X & !Y

xor2(<Bit>, <Bit>) -> <Bit>
xor2(X,Y) = (X | Y) & !(X & Y)

experiment xor1 = xor2
```

produces the response

```
xor1 behaves like xor2
  {} ~ {}
```

The second line of the output tells you how the possible internal memory states of the components line up: components with no memory have just one state, 'empty', so we are being told that the two versions of xor behave the same when both their memories are empty (which is all the time). Of course, which states of one component line up with which states of the other gets much more interesting when they do have memory — we shall see how that works, later.

Meanwhile, if the components have different behaviour, we get a report exhibiting a discrepancy:

```
experiment xor = or
```

produces

```
xor has a behaviour that or does not match
  xor(11) = 0 but or(11) = 1
```

pinpointing the difference between *exclusive* and *inclusive* 'or'.

8 Circuits with Internal Memory

This section will appear soon. It is important for CS106, but not for CS111 (where memory circuits are not a topic we engage with).