2024-05-23 Working Paper - SWIB

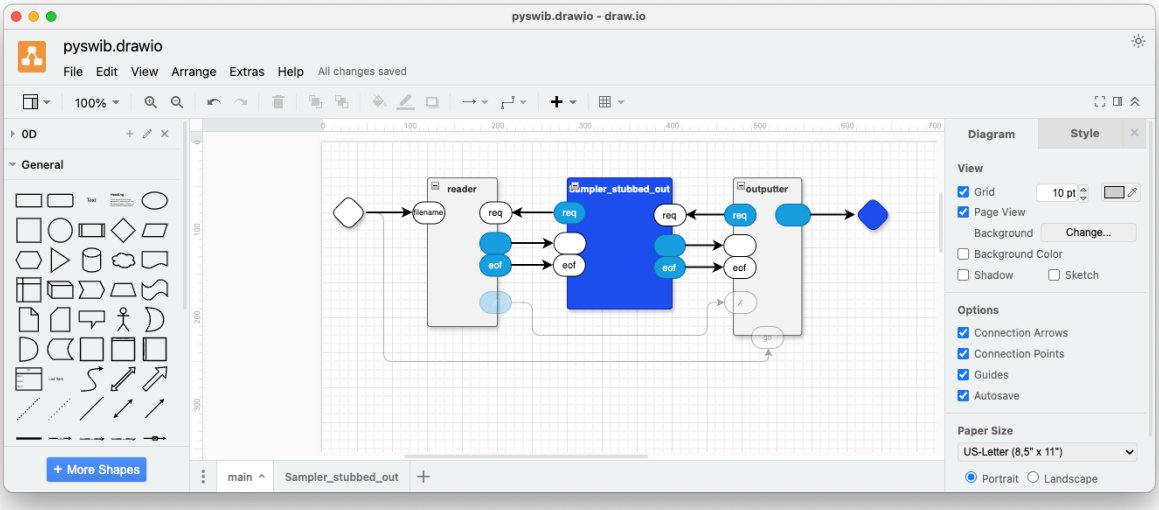
Goal

See the first working paper “2024-05-22-WP Working Paper SWIB Syntax”

Step 5. Iterate Again, Revelation About Blocking

At this point, I have a framework for inserting Python SWIBs. The `reader` reads input from a given file. The `outputter` requests strings and prints them on the console. The middle component - `Sampler_stubbed_out` - contains nothing and is just a pass-through.

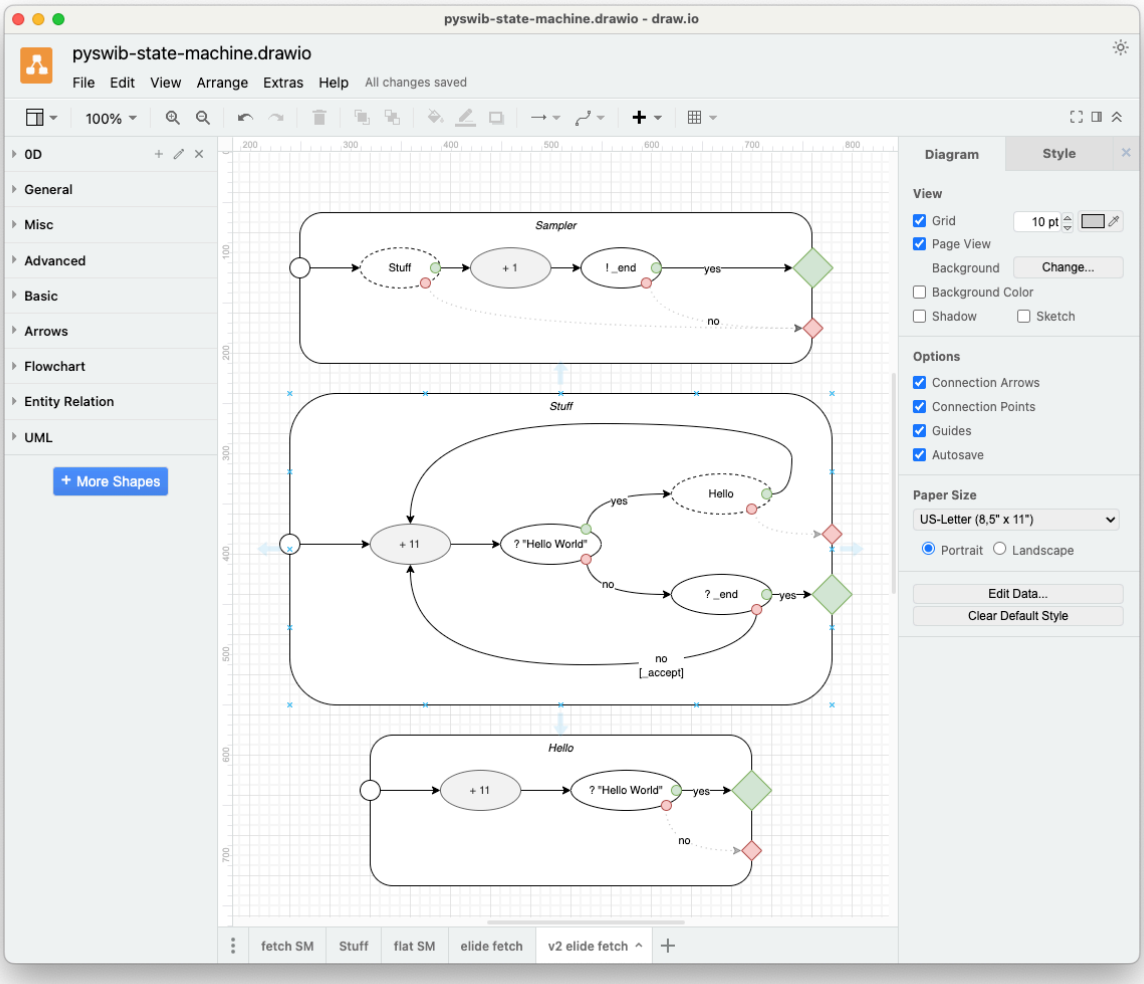
I intend to replace `Sampler_stubbed_out` with Python code generated from another diagram which contains a Receptor attached to a bunch of Python code.



`Make` runs this diagram and produces identity output given input from `test.txt` which contains

```
La la la,
Hello World!
De de de
```

The diagrams in Step 4, led to further refinements and to two revelations. The new diagram is



Here, the diagrams contains `receptrons` that have 1 input and 2 outputs. The outputs are labelled `yes` and `no` (green and red, respectively). In some cases, e.g. the grayed-out `prefetch` receptron, the `no` output is never used and does not appear on the diagram. Hmm, since `prefetch` is a blocking operation, maybe it can error out. Should I show a `no` output explicitly?

Note that the script boxes also have 1 input and 2 outputs, labelled similarly.

The revelations are:

1. Blocking can be specified explicitly as gray ovals and can be optimized to prefetch the appropriate number of characters. At present, we must specify

the number explicitly. Later we will automate the calculation of prefetch and obviate the need for explicit specification by the programmer. But, not yet. First, we want to test that all of the appropriate bits and pieces of state are being recorded in a useful manner.

2. Receptrons are drawn as ovals with 1 input and 2 outputs. They look like States, but are more specialized. A receptron always has one control-flow input and 2 control-flow outputs. States, on the other hand, can have a more general structure, with several inputs and several outputs. What - if any - is the relationship between receptrons and binary lambda calculus? Receptrons have inputs and outputs that are units of control flow. The output of a receptron is a pair of control-flow units `yes`/`no`. Binary lambda calculus accepts input parameters that form a pair of functions - `true`/`false`, a function to be executed when the `true` branch is fired and another function that is fired when the `false` branch is fired. The output of a binary lambda calculus function is another `true`/`false` pair of functions.

The revised textual form of the test program `sampler.swib` is now

```
⊢ Sampler ⊣

: Sampler ^=
  Stuff +1 _end

: Stuff ^=
  <<<
    +len("Hello World")
    [*
      | "Hello World": Hello
      | ⊥: _break
      | *: .
    ]
  >>>

: Hello ^=
  +len("Hello World")
  "Hello World"
```

I hand-compiled this to the following bytecodes (in file `sampler.bytecode`). Here, an instruction consists of 3 parts:

1. An action opcode, always prefixed by `@`.

2. Exactly 2 arguments to the action.

3. A control flow code, always prefixed by `.`.

```
@script "Sampler" _                     ..
  @enter "Sampler"  _                   .next
  @push-fresh-accumulator _             .next
  @call "Stuff" _
    @mark-yes _ _
      @prefetch 1 _                     .next
      @peek-end _ _
        @mark-yes _ _
          @accept-and-append _ _        .next
          @send-accumulator ✓ _         .out
        @mark-no _ _
          @send-string "" ✗             .out
        @mark-end _ _                   .out
    @mark-no _ _
      @send-string "" ✗                 .out
    @mark-end _ _                       .next
    @pop-accumulator _ _                .next
    @exit "Sampler" _                   .quit
@end-script _ _                         ..

@script "Stuff" _                       ..
  @enter "Stuff" _                      .next
  @push-fresh-accumulator _ _           .next
  @loop _ _
    @prefetch 11 _                      .next
    @peek "Hello World" _
      @mark-yes _ _
        @call "Hello" _
          @mark-yes _ _
            _ _ _                       .continue
          @mark-no _ _
            @send-string "" ✗           .break
          @mark-end                     .out
      @mark-no
        @peek-end
          @mark-yes
            @send-accumulator ✓         .break
          @mark-no _ _
            @accept-and-append _ _      .continue
          @mark-end _ _                 .out
      @mark-end _ _                     .out
    @mark-end-loop _ _                  .next
    @pop-accumulator _ _                .next
```

```
    @exit "Stuff" _                     .quit
@end-script _ _                         ..


@script "Stuff" _                       ..
  @enter "Stuff" _                      .next
  @push-fresh-accumulator _ _           .next
  @prefetch 11 _                        .next
  @peek "Hello World" _
    @mark-yes _ _
      @accept-and-append _ _            .next
      @send-accumulator ✓ _             .out
    @mark-no _ _
      @send-string "" ✗                 .out
    @mark-end _ _
  @pop-accumulator _ _                  .next
  @exit "Stuff" _                       .quit
@end-script _ _                         ..
```

The action opcodes can be one of

`@script` `@enter` `@exit` `@push-fresh-accumulator` `@prefetch` `@peek` `@peek-end` `@call` `@accept-and-append` `@send-accumulator` `@send-string` `@mark-yes` `@mark-no` `@mark-end` `@loop` `@loop-end`.

The control-flow codes[1] can be one of

`..` `.next` `.out` `.break` `.continue` `.quit`

[1] This isn't the first time that this has been tried. SNOBOL allows a next-line-number at the end of each statement ("card"). SNOBOL uses the concept of generalized GOTO, whereas these cfcodes have very specific ("structured") meanings.

The first cfcode `..` is a noop. The cfcode .out specifies a jump forward to the next `@end-mark`, i.e. the bottom of a `yes`/`no` choice block. The binary receptrons - `@peek`, `@peek-end`, and `@call` generate a `yes`/`no` choice. On `yes`, the next instruction is the one following the `@mark-yes` action code, whereas on `no`, the next instruction is the one following the `@mark-no` action code.

An argument is either a string, an integer, a `yes`/`no` port name `✓`/`✗` or a don't care `_`.

Many of these symbols contain characters that are not legal in Python, so we'll just make everything strings for now[2].

For example, an instruction will look like:

```
["@send-accumulator", "✓", "_", ".out"]
```

This is "nice and regular". It doesn't need to be human-readable, just machine-readable. No edge cases means that writing code for this should be easier.

An "instruction" is a list of 4 items - action, arg1, arg2, cfcode.

[2] We can optimize later. OTOH, it may turn out that this all works "fast enough", so we might not need to waste any time optimizing.

Appendix - See Also

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist "programming simplicity"]
Discord *https://discord.gg/Jjx62ypR (Everyone welcome to join)*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*