Wisteria: a modern and general Structural Description Language

Victor Miquel
École Normale Supérieure
Paris, France
victor.miquel@ens.psl.eu

ABSTRACT

Wisteria is a work-in-progress description language that aims to capture the essence of Register Transfer Level (RTL) Hardware Description Languages (HDLs) and other similar programming languages that rely on structural composition. This paper briefly describes and explains core design choices in Wisteria and their relevance for this type of language.

1 INTRODUCTION

RTL HDLs roughly fall into three categories: legacy languages such as Verilog and VHDL, that carry many flaws but are still the *de facto* industry standard, embedded languages such as Chisel[1] or Aetherling[2], which allow to tap into the expressiveness of the host language at the cost of slightly less natural syntax and more language design constraints, and standalone languages such as Filament[4] or Spade[5], that display the opposite trade-off.

The aim of Wisteria is to find the sweet spot: capturing just enough of the structure of the languages that rely on structural composition (including RTL HDLs, synchronous programming languages such as Lustre[3], and some Visual Programming Languages such as Simulink or Unreal Engine's Blueprints). The evaluation metric would then be whether it allows implementing the embedded and standalone languages mentioned above as libraries in Wisteria, while avoiding the awkwardness of embedding in a general purpose programming language. A nice bonus effect would be that once two such languages are implemented as libraries in this language, it should become very easy to bring the two together as a whole if they have complementary features.

If successful, this approach would allow to take the best of both the embedded and standalone approaches, and hopefully even make for a strong candidate for industry adoption if it can indeed bring all these research ideas under a single framework.

This paper describes and explains the core design choices in the current Wisteria iteration.

2 ORIENTED TYPES

In their most basic form, circuit/node ports have traditionally been represented in two ways: either function-like, with input ports as arguments and output ports as return values, or circuit-like, with all ports as arguments, annotated with direction information.

One must additionally consider more subtle interfaces, such as ready-valid handshakes, as it is very tempting to represent such an interface as a single composite port, containing both input signals and output signals. These ports with mixed direction (not to be confused with **`inout`** ports) are rather natural to express in the circuit-like style, but they require special support.

A first solution, implemented in Chisel[1] for instance (although worded differently here), is to add orientation information to types. There are two orientation tags: $\bot$ (= this component is driven outside of the current context and can therefore only be read) and $\top$ (= this component must be driven exactly once in the current context). Primitive types are tagged $\top$, and there are 3 additional type operations beside the usual algebraic data type operations:

- **`out`**, which discards any orientation information in the type and tags everything as $\top$.
- **`in`** (= **`flip`** $\circ$ **`out`**), which similarly tags everything as $\bot$.
- **`flip`**, which flips any orientation tag within the type, turning $\bot$ into $\top$ and vice versa.

A ready-valid buffer would look like this with oriented types:

```
struct Rv[T] { valid: out Bit, data: out T, ready: in Bit }
// or `struct Rv[T] { valid: Bit, data: out T, ready: flip Bit }`
circ buffer[T](buf_in: flip Rv[T], buf_out: Rv[T]) {...}
```

Because this feels somewhat unnatural, I propose to add another layer: *orientable* types. In an orientable type, the type's components are either tagged as $\rightarrow$ (the default) or $\leftarrow$, as opposed to $\bot$ and $\top$ in oriented types. They come with the following additional operations:

- **`uni`** (uniform), the analog of **`out`**, tagging everything as $\rightarrow$.
- **`rev`** (reverse), the analog of **`flip`**, switching $\rightarrow$ and $\leftarrow$ tags.
- **`send`**, mapping an orientable type to an oriented type by mapping $\rightarrow$ to $\top$ and $\leftarrow$ to $\bot$.
- **`recv`** = **`send`** $\circ$ **`rev`**. ($\rightarrow$ to $\bot$, $\leftarrow$ to $\top$)
- **`drive`** = **`send`** $\circ$ **`uni`**. ($\rightarrow$ to $\top$, $\leftarrow$ to $\top$)
- **`read`** = **`recv`** $\circ$ **`uni`**. ($\rightarrow$ to $\bot$, $\leftarrow$ to $\bot$)

Resulting in this easier to read version in proper Wisteria:

```
struct Rv[T] { valid: Bit, data: uni T, ready: rev Bit }
circ buffer[T](buf_in: recv Rv[T], buf_out: send Rv[T]) {...}
```

Although there is not enough space here to go into details, in Wisteria local signals of orientable type `T` need to meet the same requirements as ports of oriented type **`drive`** `T`. That is, the combined effect of all the uses of the signal must add up to driving each component of the signal exactly once. The type-checker's main rule regarding driving is that once a component of a signal is found to be driven in one of the signal's uses, it is inferred to be only read in the other uses of the signal. An optional rule that will require further testing is one that allows inferring that a component is driven in a specific signal use if it is the only use left that is compatible with that. In the presence of many polymorphic functions, this last rule does help resolve ambiguities, but it feels somewhat fragile.

While Wisteria's statement and expressions are not the object of this paper, the type of expressions are (fine-grained) oriented types.

LATTE '25, March 30, 2025, Rotterdam, Netherlands

A good illustration of Wisteria's driving-checking and inference system is its `=` operator, a symmetric connection operator with signature `circ connect[T](lhs: send T, rhs: recv T)`. Indeed, `a = b;` and `b = a;` are completely equivalent; if in the first one the generic parameter is inferred to be `X`, in the second it will be inferred to be `rev X` (given the same context).

3 CLOCKS

In legacy HDLs, signals do not have any kind of timing information attached to them. The downstream compilation passes have to infer timing information based on flip-flops, trying to figure out what would make the most sense given the starting clock domain, final clock domain, and additional timing constraints written in a different file.

I believe that a better solution to this problem is to systematically attach timing information to signals (an idea borrowed from Lustre[3]), at the type level, such that the situations on which downstream passes were previously getting confused when missing appropriate timing constraints just don't type-check, instead of producing obscure timing error or simply being silent logic errors, and that designs that type-check have *obvious* semantics.

Here are examples of what clocked types could look like:

- `Bit@sys_clk`: a physical wire, whose binary value is stable and valid between each rising edge of a certain `SYS_CLK` clock signal.
- `Uint[8]@const`: a constant unsigned integer of width 8, that can be computed at compile-time but can also live within a circuit.
- `Nat@comptime`: A heap-allocated unbounded integer, that can only live during compilation.
- `Real@continuous`: An analog value that can evolve continuously with no particular time constraints.

Although the examples above already showcase the diversity of applications offered by attaching clocks to types, one further step is required to make them shine to their full potential. That is to have clocks themselves be `@comptime` values of user-definable types. For instance, one could make it such that `sys_clk + 1` is defined and corresponds to physically the same thing as `sys_clk`, but stating that for logic purposes the first tick is irrelevant.

This could allow writing the signature below, where **`given`** `Reg[clk]` guarantees/requires that the `clk` clock supports registers and the plus operation described above.

```
circ my_pipeline[clk](x: read Bit@clk, y: drive
    Bit@(clk+3)) given Reg[clk] {...}
```

In particular, this system would allow defining clock types equivalent in features to Filament[4]'s event and Aetherling[2]'s sequences.

4 VERIFICATION

Verification is at the heart of both hardware design and synchronous languages. Furthermore, while we haven't dived into the type-system, the previous sections hint that this language would allows writing `Uint[a*((b*c)+1)]` and `Uint[a+(c*(a*b))]`, or `(clk+k)+3` and `(clk+3)+k`, and that these should be equal.

Of course it would be unreasonable to just throw everything at SMT solvers and model-checkers wait for answers that may not come. Instead, I propose to make extensive use of assertions, for both type equality and specifying safety properties. The idea being that for most compilation passes, the compiler is allowed to assume that all assertions hold (and to produce errors if it just so happens to hit an obvious absurdity), keeping things deterministic and fast. Then the language toolchain would provide means to work on proving the assertions as a separate task, at various levels of genericity: in their most polymorphic forms, after monomorphisation, or even just monitoring the remaining undecided assertions during interactive simulation.

Implicit or global assertions could also help with generic parameter inference, such as when trying to infer the value of `b` when `a` and `a + b` are known, a problem which commonly arises with the extensive use of bitwidths in hardware design.

5 MORE ON THE TYPE SYSTEM

So far, the type-system has been largely glossed over. From a user-experience perspective, I am going for a Rust-inspired system:

- Inference: Required to amortize the cost of switching from the mostly untyped legacy languages to a strongly typed language.
- Generics: Polymorphism is non-negotiable in a modern programming language, and generics provide a greater sense of control compared to the implicit type variables from the ML family.
- Typeclasses: Revisited as relations between compile-time values, they provide a good solution to talk about properties and relations between types, clocks and other compile-time values.

Together with the driving, clock and assertion systems described in the previous sections, I believe that will make for a very robust yet flexible type-system, able to detect a very large class of problems during type-checking, and a large portion of the remaining issues at verification time with the mandatory assertions required to pass type-checking.

6 FUTURE WORK

Wisteria's design is mostly concerned with it being a good *description* language, focusing on allowing the user to clearly and concisely express their intent, as well as avoiding the need for external template engines. It goes without saying that a good description language is nothing without good integration with other tools in the design workflow. This includes integration with downstream tools: compiling to lower-level languages and tracing back to the source code paths that fail to meet timing, and integration with upstream tools: providing an API to manipulate and generate circuits using general purpose software programming languages.

Once I reach satisfactory results for synchronous RTL design, I would like to explore how well the language scales to describing asynchronous circuits, mixed discrete-continuous systems, and high-level synthesis (this last one is most certainly the toughest, but I believe that given adequate primitive types, flexible abstract clocks and synchronization primitives, and a way to chose between various alternative circuit implementations, it could be achievable).

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[2] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 408–422. https://doi.org/10.1145/3385412.3385983

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320. https://doi.org/10.1109/5.97300

[4] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (June 2023), 25 pages. https://doi.org/10.1145/3591234

[5] Frans Skarman and Oscar Gustafsson. 2022. Spade: An HDL Inspired by Modern Software Languages. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 454–455. https://doi.org/10.1109/FPL57034.2022.00075