Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It...

Jan de Muijnck-Hughes
University of Glasgow, UK

Wim Vanderbauwhede
University of Glasgow, UK

Abstract

Quantitative Type-Systems support fine-grained reasoning about term usage in our *programming languages*. Hardware Design Languages are another style of language in which quantitative typing would be beneficial. When wiring components together we must ensure that there are no unused ports, dangling wires, or accidental fan-ins and fan-outs. Although many wire usage checks are detectable using static analysis tools, such as Verilator, quantitative typing supports making these extrinsic checks an intrinsic aspect of the type-system. With quantitative typing of bound terms, we can provide design-time checks that all wires and ports have been used, and ensure that all wiring decisions are explicitly made, and are neither implicit nor accidental.

We showcase the use of quantitative types in hardware design languages by detailing how we can retrofit quantitative types onto SystemVerilog netlists, and the impact that such a quantitative type-system has when creating designs. Netlists are gate-level descriptions of hardware that are produced as the result of synthesis, and it is from these netlists that hardware is generated (fabless or fabbed). First, we present a simple structural type-system for a featherweight version of SystemVerilog netlists that demonstrates how we can type netlists using standard structural techniques, and what it means for netlists to be type-safe but still lead to ill-wired designs. We then detail how to retrofit the language with quantitative types, make the type-system sub-structural, and detail how our new type-safety result ensures that wires and ports are used once.

Our ideas have been proven both practically and formally by realising our work in Idris2, through which we can construct a verified language implementation that can type-check existing designs. From this work we can look to promote quantitative typing back up the synthesis chain to a more comprehensive hardware description language; and to help develop new and better hardware description languages with quantitative typing.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Language features; Software and its engineering → Domain specific languages; Software and its engineering → System modeling languages

Keywords and phrases Hardware Design, Linear Types, Dependent Types, DSLs, Idris, SystemVerilog, Netlists

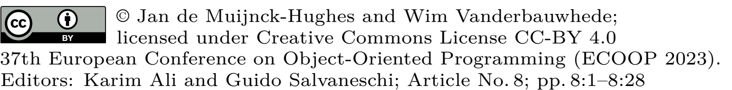
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.8

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.4`
Software (Source Code): `https://github.com/border-patrol/linear-circuits`
archived at `swh:1:dir:7faf6fa5d5aaccec3287a1359a52a16acf1be1f5`

Funding The work is funded by EPSRC grants: Border Patrol (EP/N028201/1) and AppControl (EP/V000462/1).

1 Introduction

Quantitative Type System (*QTS*) support fine-grained reasoning about term usage in our *programming languages*, such that we can control precisely how many times a bounded term can be used. We are even starting to see *QTSs* being introduced into general purpose programming languages, for example: Linear Haskell [7], Idris2 [10], and Granule [33]. Issues around term usage are, however, not limited to programming languages.


© Jan de Muijnck-Hughes and Wim Vanderbauwhede;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 8; pp. 8:1–8:28

Widely used Hardware Description Languages (*HDLs*) capture not only how hardware designs behave but also how they are physically structured. SystemVerilog is a leading industry standard *HDL* that supports the modelling, verification, and fabrication of hardware designs [14, 23]. Within SystemVerilog a hardware design consists of a high-level structural description of hardware in which modules represent physical units of design, are connected with wires, and each module contains descriptions of the modules behaviour. These behavioural descriptions describe how values are either initially placed on a wire, modified, or distributed through other channels. From this behavioural description, the design is *verified* (tested) to ensure that it behaves as intended. Verification happens using a mixture of static analysis tooling (to externally analyse the design), and testbench creation (in the same language) that is then simulated. Once the design has been verified, a *synthesis* tool converts the high-level description into a gate-level netlist; depending on the complexity of the described design, synthesis can take hours to complete. Netlists sit at the end of the synthesis process that takes high-level hardware designs and generates a single description from which hardware is then generated for fabrication or deployed in fabless environments. These netlist descriptions are also written in SystemVerilog, and the low-level gates offered can differ per hardware platform.

Term usage within *HDLs* is, unfortunately, different to that seen in general purpose programming languages. Bounded terms represent, quite literally, a physical resource that is to be used. Figure 1 presents several illustrative valid hardware designs as specified in Verilog, a subset of SystemVerilog. Each module has an explicit *header* (also known as a portgroup) that details the module's interface with the outside world. Portgroups detail the typed ports in the interface and the direction that signals can flow through each port. Within modules, we can connect ports to logic gates (or other synthesisable terms), introduce internal wiring, and make direct assignments between ports. Now let us focus on the wiring in our illustrative examples: Figure 1a presents a valid repeated use of a wire; Figure 1b presents a potentially invalid fan-in in which the two inputting wires are negated and wired into the outputting port; Figure 1c presents bit-vector indexing that requires repeated term usage; and Figure 1d presents use of an intermediate variable to reroute part of a bit-vector, and that the bit-vector's last port is not used. All these examples are structurally valid designs yet the wiring decisions (if a wire is to be used or not, and how often it is to be used) are not always explicit. Existing tools, such as *Verilator*[1], support *external* static analyses of hardware designs that includes reasoning about wire usage. Such static analyses can detect, amongst other things, if wires are unused, and if outputting ports are driven by multiple wires. If such wiring issues are not resolved they could physically damage the hardware and produce ill-behaved hardware.

An issue arising from using external tooling is that their checks are external to the design being captured, thus allowing a *time-of-check time-of-use* issue to exist between when a design is checked and when a design is used. With the rising popularity of *QTS* we must ask ourselves: Can quantitative reasoning about term usage also be applied to *HDLs*? With such quantitative reasoning we can start to embed wire usage constraints directly into our *HDL*'s type-system, such that our external checks (using external tools) now become an intrinsic aspect of the language design itself. By using a more expressive quantitative type-system we can ensure that any and all wiring decisions are explicit, and that ill-wired designs cannot be expressed as they are ruled out by the type-system as ill-typed designs.

[1] `https://verilator.org/`

```
module Example(output logic c,
               input logic a);
  nand(c, a, a);
endmodule;
```

(a) Repeated Wire Use.

```
module Example(output logic c,
               input logic a,b);
  wire logic temp;
  assign temp = b;
  assign temp = a;
  not(c, temp);
endmodule;
```

(b) Fan-In.

```
module Example(output logic c,
               input logic[1:0] ab);
  and(c, ab[0], ab[1]);
endmodule;
```

(c) Bit-Vector Indexing.

```
module Example(output logic c,
               input logic[2:0] ab);
  wire logic temp;
  temp = ab[0];
  and(c, temp, ab[1]);
endmodule;
```

(d) Intermediate Wiring.

Figure 1 Example (System)Verilog.

1.1 Contributions

Existing efforts in developing calculi for verifying hardware designs in Verilog-like languages have not sought to reason about quantitative wire usage [28, 31, 26]. Leaning on the idea of *featherweight* languages [21, 36, 34, 24] we have explored a type-based approach to reasoning about linear wire usage for *netlists* using Quantitative Type Theory (*QTT*) [4]. By capturing a valid subset of Verilog we can type existing designs to ensure that all wires are used exactly once, and showcase our approach. Moreover, we now have a foundation for future work that can investigate ways in which the calculi can be extended up the synthesis chain *formally* (inline with the formal specification); and *safely* (such that known properties of the specification still hold).

Specifically, our contributions are:

- We provide a formal unrestricted type-system (Circuits) for Verilog netlists, and introduce notions of type-(un)safety based on denotational semantics relating to graph topology.
- We provide a formal restricted type-system (CirQTS) that is resource-oriented, and detail how it provides fine-grained resource tracking for wires and ports without the need for user annotations. Using CirQTS we ensure that wire and ports can only be connected to once and we detail how we can supply new Verilog primitives that support explicit weakening of a wire's linearity safely. This weakening enables modelling of existing valid designs, and ensure wiring decisions are explicitly made.
- We put theory into practice by building a mechanised verified implementation of Circuits and CirQTS in Idris2 such that we can type-check existing valid Verilog netlists (modulo implementation restrictions and technology mapping) to detect wire usage violations. The implementation of Circuits and CirQTS has been made freely available online, and can also be found in the accompanying supplementary material.

With Circuits and CirQTS we have developed a better understanding of how to: fundamentally type wiring in Verilog; and produce a type-system that makes wiring safer. Moreover, with CirQTS we have also begun to understand what impact linear wire usage has on circuit design. Although we can reason linearly about wire usage, we have found that

sometimes linear usage is too restrictive and dependent on the intended behaviour of the design. It remains an open question over how best we can weaken linearity in more nuanced ways needed for hardware design.

1.2 Outline

Section 2 details a formal abstract syntax for Verilog netlists that our two type-systems type. Our unrestricted netlist language (CIRCUITS) is formally presented in Section 3, and the linearly wired variant (CIRQTS) in Section 4. For both calculi we detail their *wire-safety*, our approach to type-safety as an interpretation to a graph, and how CIRCUITS is wire unsafe but CIRQTS is not. With linear wiring, wire endpoints and ports, cannot be reused. We detail in Section 5 new *easy-to-add* primitives to Verilog (as primitive gates), that support valid weakening of linearity to allow wires to be split and combined. Section 6 details how we have mechanised our realisation of both languages in Idris2, and Section 7 discusses the results of using CIRQTS for designing netlists. From our experimentation we discuss the efficacy of our approach in Section 7. We end by relating our contributions to the wider world in Section 8.

2 A Syntax for Netlists

Figure 2 presents the shared abstract syntax for CIRCUITS and CIRQTS. Both languages abstract over a subset of Verilog's concrete syntax required for gate-level modelling. The syntax is unremarkable, which highlights the simplicity that accompanies netlist language design.

$$
\begin{aligned}
n : \mathbb{N} &::= \text{Natural Numbers} \\
d : \mathcal{D} &::= \mathsf{I} \mid \mathsf{O} \mid \mathsf{IO} \\
i : \mathcal{I} &::= \mathsf{Logic} \mid [i\,;n] \\
g &::= \mathsf{mux2}\,(e,e,e,e) \mid b\,(e,e,e), b \in \{\mathsf{and}, \mathsf{or}, \ldots\} \mid u\,(e,e), u \in \{\mathsf{not}, \ldots\} \\
e &::= \varphi \mid \mathsf{port}\; i\; d\; \mathsf{as}\; \varphi\; \mathsf{in}\; e \mid \mathsf{gate}\; g\; \mathsf{as}\; \varphi\; \mathsf{in}\; e \mid \mathsf{wire}\; i\; \mathsf{as}\; \varphi\; \mathsf{in}\; e \mid \mathsf{assign}\; e \leftarrow e\; \mathsf{in}\; e \\
&\quad \mid \mathsf{stop} \mid (\mathsf{index}\; e\; n) \mid (\mathsf{cast}\; e\; f) \mid (\mathsf{readFrom}\; e) \mid (\mathsf{writeTo}\; e)
\end{aligned}
$$

Figure 2 Abstract Syntax for CIRCUITS and CIRQTS.

We restrict our modelling to simple synthesisable datatypes ($\mathcal{I}$) as seen in Verilog: 4-state logic bits and their aggregation into bit-vectors. The set of supported gates (g) is indicative, as many netlists are dependent on the technology (gates) as supported by the underlying hardware. For our investigation we explored two input multiplexers, binary logic gates, and unary logic gates, as this presents an expressive enough set of gates to create interesting designs. Although we do not consider gates with n-ary outputs, such as demultiplexers, our setting does support them as we shall see in Section 4.2. Whilst this language seems overly restrictive when compared to state-of-the-art *HDLs*, the syntax and type-system can be extended to deal with n-ary gate syntax, and new primitive gates added.

Most notable from Figure 2 is that the top-level module header is implicit, and sequencing (recall that Verilog is imperative) of statements is realised as continuations on desired sub-terms. Circuit designs represent a single module with a predefined set of ports (supporting uni-directional and bi-directional signals), as such we can represent the module header as a series of bespoke let-bindings that introduce bound terms (with variables being represented

by φ) into the corresponding scope. Declarations for gates and internal wires follow this same pattern, as does the direct assignment (connection) of ports and wires which is supported by the assign statement. Verilog also supports anonymous gate declarations, to keep our minimal calculi small we have purposefully not incorporated them into the syntax but note their addition is straightforward. Finally, the stop expression indicates the end of the netlist specification.

The final set of expressions are required to capture type-safe wiring of ports and wires to gates. Naturally, bit-vectors can be indexed to access individual wires in the vector. We do not support slicing as it can be elaborated/synthesised into our core syntax. The remaining expressions are however, not explicit in Verilog's concrete syntax. Nonetheless, they can be inserted automatically through syntax elaboration.

The first of these hidden expressions supports the insertion of bi-directional ports into gates. Gates only have ports that are inputting or outputting, but bound ports may be bi-directional. Cast, like index, is an in-place operation on ports and endpoints, that supports transformation of the given direction to supplied direction f.

The final two expressions relate to inserting internal wires into declared gates. Verilog's concrete syntax does not discriminate between ports and wire endpoints. We thus introduce two projection terms to support such access, one that supports reading from a channel, and the other to support writing to a channel.

We end this section with Figure 3 that illustrates how Figure 1d is encoded in the shared syntax.

port Logic O as c in
port Logic I as ab in
wire Logic as $temp$
  in assign (writeTo $temp$) $\longleftarrow$ (index ab 0)
  in gate and (c, (readFrom $temp$), (index ab 1)) as ga
  in stop

Figure 3 Figure 1d in the shared Syntax.

3 A Structural Type-System

The abstract syntax from Section 2 helps direct well-formed netlists through its syntactical structure. Nonetheless, a type-system will categorically ensure that the netlists are *well-formed*. This section introduces a simple, yet *flawed*, type-system for CIRCUITS in which term usage is unrestricted, but nonetheless respects how netlists are typed. Practically speaking, the flaw allows uncontrolled fan-ins and fan-outs to be inserted unchecked into the design, and for ports and channels to be left dangling (unused). Section 4 details how quantitative wire usage ensures that ports and channels are used exactly once.

3.1 Types and Contexts

Figure 4 details the (unsurprising) set of types and typing context for CIRCUITS. Ports types ($\mathcal{P}$) are parameterised by how signals flow from ports, together with the port's shape as captured by the given datatype. Similarly, wire types ($\mathcal{W}$) are parameterised by the wire's shape, the provided datatype. Further, gates are given their own type ($\mathcal{G}$), and the unit type ($\mathbb{1}$) signifies the end of a design.

$$t : \textsc{Type} ::= \mathcal{P}\,(i, d) \mid \mathcal{W}\,(i) \mid \mathcal{G} \mid \mathbb{1}$$
$$\Gamma ::= \emptyset \mid \Gamma + x : t$$

Figure 4 Types and Contexts for CIRCUITS.

3.2 Typing Rules

Figure 5 details the typing rules for CIRCUITS. The introduction rules for natural numbers and datatypes are not presented and are implicitly given in Figure 2. Much like the rest of the definition of CIRCUITS, the rules are for the most part unsurprising and follow standard norms. Rule STOP introduces the unit type, signalling the end of the design. Rules MUX,BIN, & UN type gates and, as in Verilog itself, gates present their output ports then their input ports such that all ports have the same type, which for netlists will be: Logic. The Rules PORT,WIRE, & GATE adapt the standard presentation for let-bindings in which the body is extended with the introduced variable. Direct assignment (Rule ASSIGN) of an output port to an input port happens if they both have the same shape i.e. datatype. Indexing vector-shaped ports (Rule INDEX) returns a single port from the array with the shape of the contained elements, and the same signal flow. Further, a compile-time bounds check is required to ensure safe vector-indexing.

Gates and direct assignment only support ports with flow input or output. Casting (Rule CAST) supports safe transformation of a bidirectional port to either an input or output, which we formally express as follows:

▶ **Definition 1** (Valid Casting of Directions). *Given $a, b : \mathcal{D}$ the safe transformation of flow a to b, which we denote as* ValidCast (a, b)*, is:*

ValidCast (a, b)	I	O	IO
I	×	×	×
O	×	×	×
IO	✓	✓	×

We have purposefully presented a *strict* interpretation of casting, and do not include the *identity* cast. During the mechanised elaboration of terms (Section 6), we wished to ensure that casts are only inserted if the cast would change the direction. Inclusion of the identity cast would not weaken our result.

The final two rules, Rules READ & WRITE detail how a wire's endpoints are accessed by projection. It is here that the typing rules become somewhat counter-intuitive. Intuition tells us that projecting a wire to *read its contents* would return an outputting port, and conversely *writing to a wire* would require its inputting port. Instead we have swapped the directions: reading gives an input; writing gives an output. Such a swap supports the *natural* typing rules when checking gates and direct assignments. That is, by swapping the expected flow of information during projection we do not need to check if a port marked output will go to an input and vice-versa. Checking for direction equality thus simplifies the typing rules, and checking, for CIRCUITS.

$$\textsc{Var}\ \frac{\varphi : t \in \Gamma}{\Gamma \vdash \varphi : t} \qquad \textsc{Stop}\ \frac{}{\Gamma \vdash \mathsf{stop} : \mathbb{1}}$$

$$\textsc{Mux}\ \frac{\Gamma \vdash o : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \quad \Gamma \vdash c : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \quad \Gamma \vdash a : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \quad \Gamma \vdash b : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I})}{\Gamma \vdash \mathsf{mux2}\,(o, c, a, b) : \mathcal{G}}$$

$$\textsc{Bin}\ \frac{\Gamma \vdash o : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \quad \Gamma \vdash x : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \quad \Gamma \vdash y : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \quad b \in \{\mathsf{and}, \mathsf{or}, \ldots\}}{\Gamma \vdash b\,(o, x, y) : \mathcal{G}}$$

$$\textsc{Un}\ \frac{\Gamma \vdash o : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \quad \Gamma \vdash i : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \quad u \in \{\mathsf{not}, \ldots\}}{\Gamma \vdash u\,(o, i) : \mathcal{G}}$$

$$\textsc{Port}\ \frac{i : \mathcal{I} \quad d : \mathcal{D} \quad \Gamma + \varphi : \mathcal{P}\,(i, d) \vdash b : \mathbb{1}}{\Gamma \vdash \mathsf{port}\; i\; d\; \mathsf{as}\; \varphi\; \mathsf{in}\; b : \mathbb{1}} \qquad \textsc{Wire}\ \frac{i : \mathcal{I} \quad \Gamma + \varphi : \mathcal{W}\,(i) \vdash b : \mathbb{1}}{\Gamma \vdash \mathsf{wire}\; i\; \mathsf{as}\; \varphi\; \mathsf{in}\; b : \mathbb{1}}$$

$$\textsc{Gate}\ \frac{\Gamma \vdash g : \mathcal{G} \quad \Gamma + \varphi : \mathcal{G} \vdash b : \mathbb{1}}{\Gamma \vdash \mathsf{gate}\; g\; \mathsf{as}\; \varphi\; \mathsf{in}\; b : \mathbb{1}}$$

$$\textsc{Assign}\ \frac{d : \mathcal{I} \quad \Gamma \vdash i : \mathcal{P}\,(d, \mathsf{I}) \quad \Gamma \vdash o : \mathcal{P}\,(d, \mathsf{O}) \quad \Gamma \vdash b : \mathbb{1}}{\Gamma \vdash \mathsf{assign}\; i \longleftarrow o\; \mathsf{in}\; b : \mathbb{1}}$$

$$\textsc{Index}\ \frac{d : \mathcal{I} \quad f : \mathcal{D} \quad n : \mathbb{N} \quad \Gamma \vdash p : \mathcal{P}\,([d\,; m], f) \quad [n < m]}{\Gamma \vdash (\mathsf{index}\; p\; n) : \mathcal{P}\,(d, f)}$$

$$\textsc{Cast}\ \frac{d : \mathcal{I} \quad a, b : \mathcal{D} \quad \Gamma \vdash p : \mathcal{P}\,(d, a) \quad \mathsf{ValidCast}\,(a, b)}{\Gamma \vdash (\mathsf{cast}\; p\; a) : \mathcal{P}\,(d, b)}$$

$$\textsc{Read}\ \frac{d : \mathcal{I} \quad \Gamma \vdash c : \mathcal{W}\,(d)}{\Gamma \vdash (\mathsf{readFrom}\; c) : \mathcal{P}\,(d, \mathsf{I})} \qquad \textsc{Write}\ \frac{d : \mathcal{I} \quad \Gamma \vdash c : \mathcal{W}\,(d)}{\Gamma \vdash (\mathsf{writeTo}\; c) : \mathcal{P}\,(d, \mathsf{O})}$$

Figure 5 Simple Typing Rules for CIRCUITS.

3.3 Wire-(Un)Safety

As programming languages are computational it makes sense to reason about their type-safety using their operational semantics: How they compute! Standard syntactic approaches require that we describe how terms reduce (progress) during operation, and prove that once a value is reached that the types have been preserved (preservation). *HDLs* on the other hand, not only describe how signals flow across wires, but also the structure of the circuit itself. At the level of netlists, *HDLs* are not computational languages and there is no reduction of terms. We must, therefore be concerned with a design's physical structure rather than its behaviour. As such we will reason about our type system's correctness based on its *wire-safety*, that is how well the design has been wired together, as opposed to how the design behaves. Specifically, we will use a denotational approach in which we use the circuit design as the instructions to

construct a graph, and assess the design's wire-safety using the constructed graph's topology. We stress that these results provide assurances over how circuits are wired and not how they behave. Behavioural safety requires more work [22, 28].

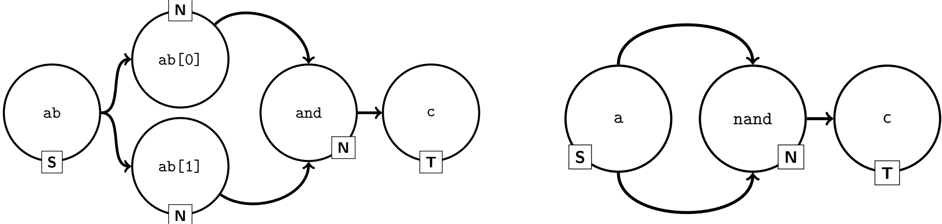
We remark that our denotational approach to wire-safety is inspired by the use of definitional interpreters [38, 37] to prove type-soundness [3]. We elide technical details, highlighting the main results only, and remark that the accompanying artefact (which provides a mechanisation of our contributions, see Section 6) contains the full details.

As a reminder, the "safety" result we present here is purposefully unsafe to capture how a structural type-system does not provide guarantees over the sub-structural property that is wire-safety. The next section Section 4 describes a more (type) safe way to determine wire-safety.

3.3.1 Graphs for Hardware

Digital circuits (netlists) are *just* multigraphs. Leaf vertices map to a netlist's inputting and outputting ports, and internal vertices represent logic gates. Wires too can be represented by pairs of internal vertices (one for each endpoint, with a single edge to connect the two), and edges in our graph model wiring between ports, channels, and gates. To better reason about our graph's topology we categorise leaf vertices as being sources (S; output ports), targets (T; input ports), or unknown (U; bidirectional ports). Internal nodes (N) capture gates, casting, and projection of wires. Figure 6 shows how Figures 1a and 1c can be viewed as a hardware graph, and that the edges represent the wiring between ports, gates, and wires. More formally we can describe such graphs as:

▶ **Definition 2** (Hardware Graph). *Let $H = \langle vs, es \rangle$ be a directed multigraph where vs is a set of vertices, and es is a list of edges. Vertices in H will be labelled with an element of $\{S, T, U, N\}$.*



(a) Figure 1c as a Hardware Graph.

(b) Figure 1a as a Hardware Graph.

Figure 6 Example Hardware Graphs for CIRCUITS.

To reason about wire usage we need to define what it means for our hardware graph to be wired. We can determine the expected degree a vertex has directly from its categorisation: source vertices will have zero incoming edges, and zero or more outgoing edges; leaf vertices will have zero or more incoming edges, and zero outgoing edges; bi-directional vertices will have zero or more incoming or outgoing edges; and internal vertices will have have zero or more incoming or outgoing edges. Using $\mathsf{deg}^{+}(v, es)$ and $\mathsf{deg}^{-}(v, es)$ to calculate the out and in degrees for a vertex v from a given list of edges es, we can compare the given degrees with the expected degrees. If the relations hold then the circuit has been wired. We formally present this definition as:

▶ **Definition 3** (A Wired Hardware Graph). *Let $H = \langle vs, es \rangle$ be our hardware graph, the wiring of H is well-formed, $C(H)$, if:*

$$\forall v \in vs \begin{cases} S & \mathsf{deg}^{+}(S, es) \geq 0 \wedge \mathsf{deg}^{-}(S, es) \equiv 0 \\ T & \mathsf{deg}^{+}(T, es) \equiv 0 \wedge \mathsf{deg}^{-}(T, es) \geq 0 \\ U & \mathsf{deg}^{+}(U, es) \geq 0 \vee \mathsf{deg}^{-}(U, es) \geq 0 \\ N & \mathsf{deg}^{+}(N, es) \geq 0 \wedge \mathsf{deg}^{-}(N, es) \geq 0 \end{cases}$$

Interpretation of terms will result in either: an error related to binding; a hardware graph; a vertex; the empty value ($\perp$) of the empty type. Port definitions are interpreted into leaf vertices following the port's direction, whereas gates, wires, and casts interpret into internal nodes. The end of a specification is interpreted into the empty type. Port usages, and channel projection, inserts edges into the graph. We can also strengthen the interpretation by making it type-directed (represented by $[\![t]\!]_T$), where terms from Circuits are mapped to hardware graph concepts.

Formally we can denote the act of interpretation as follows:

▶ **Definition 4** (Interpretation). *Let Σ_i be an interpretation environment and $H_i = \langle vs, es \rangle$ be a Hardware Graph. We denote the interpretation of Circuits specification $(\Gamma \vdash e : t)$ as:*

$$(\Sigma, H_i)[\![e]\!] ::= \mathsf{Error} \mid \mathsf{Done}\,(H_o, v : [\![t]\!]_T)$$

where H_o is the resulting hardware graph.

The interpretation environment (Σ) stores the result of interpretation for bound terms and is passed around as an input to interpretation explicitly. The accumulator graph (H) carries the resulting graph as we traverse sub-terms. Here subscripts i and o denote an inputting and resulting value. We cannot *just* return a complete graph for each interpretation step as not all terms return a graph. Take ports, for example, they return a vertex when interpreted during the construction of edges, and stop returns the empty value.

More information about the interpretation can be found in the accompanying artefact which contains the mechanisation (Section 6) of Circuits.

3.3.2 Well-Typed Circuits are Valid Hardware Graphs

With this high-level description of interpretation we can detail our strong interpretation soundness result, which is our wire-safety result.

▶ **Theorem 5** (Strong Interpretation Soundness).

$$\frac{\emptyset \vdash e : \mathbb{1} \qquad [\![e]\!] = \mathsf{Done}\,(\Sigma, H, v)}{\Sigma \equiv \emptyset \qquad v \equiv \perp \qquad C(H)}$$

Strong interpretation soundness states that interpretation of a closed specification will result in a valid wired hardware graph. A weak soundness result would be that we can construct *just* a valid hardware graph.

3.3.3 Proof Sketch

Our soundness proof needs to ensure that our interpretation will result in a valid hardware graph. We can structure our proof using Siek's "three easy lemmas" [41] but adapted for interpretation as seen with existing work [3] and adapted to build hardware graphs. Siek's

approach requires that we divide the proof into three sub-lemmas that, when combined, cover all aspects of the interpretation process: 1) interpreting primitives; 2) ensuring that variables are well-scoped; and 3) that doing interpretation will produce a value.

The first lemma deals with typing primitives.

► **Lemma 6** (Primitives). *Well-typed primitives are not stuck and will interpret correctly.*

Proof. By induction on typing derivations. ◄

The only primitive term in Circuits is stop, the rest contain interpretable sub-terms. The term stop will always interpret.

The next lemma addresses environment lookups. As in existing work [3] we must first define a consistency relation ($\models$) between typing contexts and interpretation environments. Such a relation ensures that as the typing context grows, so will the interpretation environment.

► **Definition 7** (Consistent Interpretation Environments).

$$\textit{EmptyEnv}\quad \frac{}{\emptyset \models \emptyset} \qquad \textit{ExtendEnv}\quad \frac{\Gamma \models \Sigma}{\Gamma + \varphi : t \models \Sigma + (\varphi, v : [\![\ t\ ]\!]_T)}$$

Using our consistency relation ($\models$) we can describe well-scoped environment lookups. When given an interpretation environment that is consistent with a typing context, we can map bound variables within the typing context to their equivalent bindings in the corresponding interpretation environment. Thus when given a well-scoped variable $\varphi \in \Gamma$ such that φ has type t, then there is a value $(v : [\![\ t\ ]\!]_T)$ in the interpretation environment bound to φ.

► **Lemma 8** (Lookup). *Well-typed interpretation environment lookups are not stuck and will interpret correctly.*

$$\frac{\Gamma \models \Sigma \qquad \varphi : t \in \Gamma \qquad x \equiv [\![\ t\ ]\!]_T}{(\varphi, v) \in \Sigma \qquad v : x}$$

Proof. By structural induction over the interpretation environment. ◄

As environment lookups are guided by the typing-context, interpretation will succeed because all references to bound terms exist. Otherwise lookup will fail.

The final lemma, which we provide here, details interpretation.

► **Lemma 9** (Interpretation). *If the interpretation returns a result then the result is a valid hardware graph.*

$$\frac{\Gamma \vdash e : t \qquad \Gamma \models \Sigma \qquad (\Sigma, H_i)[\![\ e\ ]\!] ::= (\mathsf{Done}\ res)}{res = (H_o, v) \qquad v : t' \qquad t' \equiv [\![\ t\ ]\!]_T}$$

Proof. By induction on typing derivations. ◄

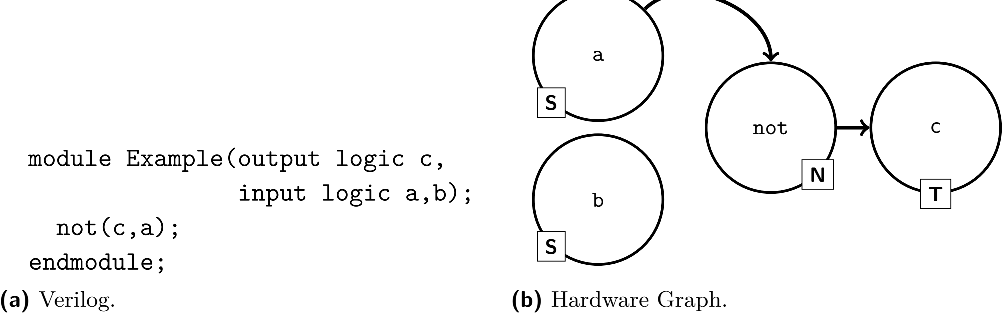
We know that within definitional interpreters the strong soundness property [3] corresponds to the classic strong soundness property used in proving type soundness [47]. With our final lemma we can show that well-typed Circuits specifications *will* construct correctly wired hardware graphs, if not an error will occur. Interpretation will finish when the term stop is reached. The consistency relation ensures that the typing contexts and interpretation environment remain consistent as we step across binders.

We only check the strong soundness result after the interpretation process has finished. Sub-terms return components of a hardware graph, and not a complete graph itself, we need to ensure that the soundness result is applied to a whole specification.

3.4 Towards True Wire-Safety

Our definition of a wire-safety, and thus safety of Circuits type-system, is not one that leads to designs that are inherently *safely wired*: Designs in which wires are used once, and wire usage is clear. The type-system for Circuits is not sub-structural, and will admit designs in which ports (and wires) are left dangling or partially used. We know this because our definition for wiredness ($\mathcal{C}\,(H)$) is too loosely specified. We can show this by changing the definition of $\mathcal{C}\,(H)$ to one where the expected degree of leaf nodes is *greater than zero* to *equal to one*. Such a change now invalidates various graphs in which ports and wires were left dangling, and were once admitted by Circuits type-system.

Consider, for example, the netlist and its corresponding hardware graph in Figure 7. The netlist has a dangling input `b`. It is unclear from the immediate context if `b` is supposed to be dangling or included in the design; gates in Verilog are n-ary. The change in definition of $\mathcal{C}\,(H)$ supports identification of ill-wired designs, as the expected out degree of for b is one but its given degree is zero.



(a) Verilog. **(b)** Hardware Graph.

Figure 7 Example Netlist with Ambiguous Wiring.

Changing the definition of $\mathcal{C}\,(H)$ is still, however, not enough to get safe designs. We need to reason about bidirectional ports, remember that bit-vectors can be indexed, ensure that channel endpoints are accessed once, and that gates are only wired into once. We need a sub-structural type-system to reason about wire/port usage. We will show this in the next section, and provide a better definition of what it means for a hardware graph to be *well-wired*.

4 A Sub-Structural Type-System

The illustrative designs from Figure 1 are not linear in their bound term usage. There are repeated variables, dangling wires, and parts of bit-vectors left unused. Trying to fit *wire usage constraints* into existing quantitative systems is hard. Generally speaking, *QTSs* such as Atkey's *QTT* [4], and those based around linear logic [45, 44] and graded semirings [33], are designed to reason about term usage within general purpose programming languages. Specifically, linear systems require that bounded terms are used *exactly once*; affine systems require that bounded terms are used *at most once*; graded systems require that bounded terms are used *at most n-times*; and *QTT* allows bounded terms to be used linearly, and unbounded terms have unrestricted usage. *None-One-Tonne* systems do not have a fine grained usage modality. *HDLs* capture circuit behaviour as well as structure.

A key design challenge when *linearising* Circuits type-system is to know what *usage* means for *HDLs*, as bounded terms do not have singular usage. Within Circuits wires have two endpoints that are used in separate locations, and bit-vector shaped wires/ports can

be indexed in strange and wonderful ways. Roughly, projection of endpoints and indexing implies that each sub-term may be partially used in a design and type-checking needs to take this into account. Further, while ports/wire have bounded usage, gates are unbounded. It is not clear how QTT and Granule can be co-opted to encode these domain specific usages as part of a semiring structure. Our approach to the type-system for CirQTS must be different, for now.

4.1 Types and Usages

Figure 8 details the additional infrastructure required to make Circuits linear. Inspiration is taken from QTT to base our system of usages (Figure 8a) on the *none-one-tonne* semiring. Things are free until used, and some things will always be free to use i.e. unbounded usage. In practice, however, this can be distilled to the ordinary boolean semiring as our usage requirements are more conservative and constrained, and we make no link between compile time and runtime quantities as QTT is used in practice [10].

$$R ::= 0 \mid 1 \mid \omega$$

(a) Usages.

$$\begin{array}{rcl|c} i : \mathcal{I} & ::= & \mathsf{Logic} & [i\,;n] \\ u : \textsc{Usage}_D(i) & ::= & R & \{u_j : \textsc{Usage}_D(i)\ ^{j \in 1..n}\} \\ \mathsf{Init}_D\ (u) & ::= & 1 & \{\mathsf{Init}_D\ (u_j)\ ^{j \in 1..n}\} \end{array}$$

(b) DataTypes.

$$\begin{array}{rcl|l|c|c} t : \textsc{Type} & ::= & \mathcal{P}\ (i, d) & \mathcal{W}\ (i) & \mathcal{G} & \mathbb{1} \\ u : \textsc{Usage}_T(t) & ::= & \textsc{Usage}_D(i) & (\textsc{Usage}_D(i), \textsc{Usage}_D(i)) & \omega & \omega \\ \mathsf{Init}_T\ (u) & := & \mathsf{Init}_D\ (i) & (\mathsf{Init}_D\ (i), \mathsf{Init}_D\ (i)) & \omega & \omega \end{array}$$

(c) Types.

$$\Gamma ::= \emptyset \mid \Gamma + (\varphi : t, u : \textsc{Usage}_T(t)) \mid \Gamma \pm (\varphi : t, u : \textsc{Usage}_T(t))$$

(d) Contexts.

$$\Gamma_i \vdash e : t \dashv \Gamma_o$$

(e) Judgements.

Figure 8 Types and Contexts for CirQTS.

Figure 8b presents how we capture datatype usage. Recall that the types for ports and channels are indexed by a datatype. The shape of a port/wire's datatype will direct the usage we need to track. Logic-shaped ports and channels will have a single wire to use, and bit-vectors an array of elements to use. Bit-vectors are, however, multidimensional and the usages should reflect that when indexing bit-vectors. When initialising usages for datatypes we set them to be free.

Figure 8c presents how we specify usages for bindable types. Ports will have to keep track of their usage based on the usage of the datatype they are indexed by, and wires will have a pair of usages (one for each endpoint). Gates (and unit) will be left unrestricted as their use is unrestricted. The syntax for Circuits/CirQTS enforces that stop can only be used once. For newly introduced ports and wires, their initial usage will be free.

Figure 8d details how we situate our usages in the type-system. Following existing work [46] we must annotate our typing context to keep track of the usage of bounded terms. Given the different shapes of our boundable types, we cannot use linear algebra to capture usage updates. Instead we take a simpler approach by envisaging our types (and type-system)

as a stateful resource in which the usage of bound term is a state. Specifically not only can our context be extended, but we can also update the state of a bound term's resource: its usage.

We differ from standard linear typing approaches by not relying on context splitting. Rather our judgement formation (Figure 8e), and typing rules, are more algorithmic. Successfully typing a term will may result in a new updated context. This mirrors the algorithmic presentation of the Linear Lambda Calculus (LLC) [45, Figure 1-6].

4.2 New Syntax for Indexing

$$
\begin{aligned}
(\mathsf{index}\, e\, n) &\longrightarrow (\mathsf{index}\, \varphi_p\, \{n_i{}^{i\in 1..m}\}) \\
(\mathsf{readFrom}\, e) &\longrightarrow (\mathsf{readFrom}\, \varphi_c) \mid (\mathsf{readFromAt}\, \varphi_c\, idx) \\
(\mathsf{writeTo}\, e) &\longrightarrow (\mathsf{writeTo}\, \varphi_c) \mid (\mathsf{writeToAt}\, \varphi_c\, idx)
\end{aligned}
$$

Figure 9 Syntax Changes required for CirQTS.

We can now start to describe the typing rules for CirQTS as presented in Figure 10. We must first realise, however, that the abstract syntax (Figure 2) is too expressive. The trouble stems from n-dimensional indexing of bit-vectors. Type-checking each (sub)term in our stateful typing updates the states held within the typing context. How do we know which sub-usage of the bit-vector for which bound term in the typing-context needs to be updated? Indexing a one-dimensional bit-vector is simple, but for deeper indexing (i.e. indexing a projection or an index of an index) updating the state is impossible. We need the location within the context. Thus, we must flatten nested indices to a single term, and have separate terms for when wire endpoints are also projected. This transformation step occurs during elaboration, and Figure 9 presents the replacement terms required. How we can adapt the type-system to support nested indexing is not clear.

4.3 Typing Rules

Figure 10 presents the salient typing rules for CirQTS. Missing are the rules for the *write* projection as they mirror the rules for the *read* projection. The presented rules are not too dissimilar from those in Figure 5. For each rule we reason, using custom predicates, about the state of the usage resource for each binder, or expect context transitions to enforce our linear wiring property.

Rule Var ensures that bound variables are only used *if* they are completely free; ruling out use of partially used variables that have been used through indexing. Such a predicate means that once a port/wire has been indexed it can only be used through indexing.

Rule Stop describes the end conditions for each bound term's usage, as dictated by the $\mathsf{CanStop}\,(t,u)$ predicate that requires that all bound terms must be totally used. That is all ports and wires must be used for the design to type-check. How we define $\mathsf{CanStop}\,(t,u)$ impacts on what it means for a design to be *totally used*. Verilator, for example, does not require output ports to be fully used in its static analysis. That is, Verilator is affine for outputs but linear for inputs, whereas CirQTS is linear for all wires/ports. We stress that the termination conditions for CirQTS are not closed for debate. Rather it supports discussing the conditions under which ports and wires are said to be sufficiently used. Verilator's termination choice, for example, can be replicated by requiring that $\mathsf{CanStop}\,(t,u)$ requires inputting ports and wire endpoints be used.

$$\textsc{Var}\ \frac{(x : t, o) \in \Gamma \qquad [\mathsf{IsFree}\,(o)]}{\Gamma \vdash x : t \dashv \Gamma \pm (x : t, \mathsf{Use}\,(o))} \qquad \textsc{Stop}\ \frac{\{\mathsf{CanStop}\,(t, u) \mid (x : t, u) \in \Gamma\}}{\Gamma \vdash \mathsf{stop} : \mathbb{1} \dashv \emptyset}$$

$$\textsc{Cast}\ \frac{d : \mathcal{I} \qquad a, b : \mathcal{D} \qquad \Gamma_1 \vdash p : \mathcal{P}\,(d, a) \dashv \Gamma_2 \qquad \mathsf{ValidCast}\,(a, b)}{\Gamma_1 \vdash (\mathsf{cast}\, p\, a) : \mathcal{P}\,(d, b) \dashv \Gamma_2}$$

$$\textsc{Mux}\ \frac{\Gamma_2 \vdash c : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_3 \qquad \begin{array}{c}\Gamma_1 \vdash o : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \dashv \Gamma_2 \\ \Gamma_3 \vdash a : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_4\end{array} \qquad \Gamma_4 \vdash b : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_5}{\Gamma_1 \vdash \mathsf{mux2}\,(o, c, a, b) : \mathcal{G} \dashv \Gamma_5}$$

$$\textsc{Bin}\ \frac{\Gamma_2 \vdash x : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_3 \qquad \begin{array}{c}\Gamma_1 \vdash o : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \dashv \Gamma_2 \\ \Gamma_3 \vdash y : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_4\end{array} \qquad b \in \{\mathsf{and}, \mathsf{or}, \ldots\}}{\Gamma_1 \vdash b\,(o, x, y) : \mathcal{G} \dashv \Gamma_4}$$

$$\textsc{Un}\ \frac{\Gamma_1 \vdash o : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \dashv \Gamma_2 \qquad \Gamma_2 \vdash i : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_3 \qquad u \in \{\mathsf{not}, \ldots\}}{\Gamma_1 \vdash u\,(o, i) : \mathcal{G} \dashv \Gamma_3}$$

$$\textsc{Assign}\ \frac{d : \mathcal{I} \qquad \Gamma_1 \vdash i : \mathcal{P}\,(d, \mathsf{I}) \dashv \Gamma_2 \qquad \Gamma_2 \vdash o : \mathcal{P}\,(d, \mathsf{O}) \dashv \Gamma_3 \qquad \Gamma_3 \vdash b : \mathbb{1} \dashv \emptyset}{\Gamma_1 \vdash \mathsf{assign}\, i \leftarrow o\, \mathsf{in}\, b : \mathbb{1} \dashv \emptyset}$$

$$\textsc{Ports}\ \frac{i : \mathcal{I} \qquad d : \mathcal{D} \qquad \Gamma + (\varphi : \mathcal{P}\,(i, d), \mathsf{Init}_T\,(\mathcal{P}\,(i, d))) \vdash b : \mathbb{1} \dashv \emptyset}{\Gamma \vdash \mathsf{port}\, i\, d\, \mathsf{as}\, \varphi\, \mathsf{in}\, b : \mathbb{1} \dashv \emptyset}$$

$$\textsc{Wire}\ \frac{i : \mathcal{I} \qquad \Gamma + (\varphi : \mathcal{W}\,(i), \mathsf{Init}_T\,(\mathcal{W}\,(i))) \vdash b : \mathbb{1} \dashv \emptyset}{\Gamma \vdash \mathsf{wire}\, i\, \mathsf{as}\, \varphi\, \mathsf{in}\, b : \mathbb{1} \dashv \emptyset}$$

$$\textsc{Gate}\ \frac{\Gamma_1 \vdash g : \mathcal{G} \dashv \Gamma_2 \qquad \Gamma_2 + (\varphi : \mathcal{G}, \mathsf{Init}_T\,(\mathcal{G})) \vdash b : \mathbb{1} \dashv \emptyset}{\Gamma_1 \vdash \mathsf{gate}\, g\, \mathsf{as}\, \varphi\, \mathsf{in}\, b : \mathbb{1} \dashv \emptyset}$$

$$\textsc{Index}\ \frac{d : \mathcal{I} \qquad f : \mathcal{D} \qquad idx \coloneqq \{{m_i}^{\,i \in 1..j}\} \qquad (\varphi_p : \mathcal{P}\,([d\,;n], f), u) \in \Gamma \qquad [\mathsf{IsFreeAt}\,(u, idx)]}{\Gamma \vdash (\mathsf{index}\, \varphi_p\, idx) : \mathcal{P}\,(\mathsf{HasTypeAt}\,([d\,;n], idx), f) \dashv \Gamma \pm (\varphi_p : \mathcal{P}\,([d\,;n], f), \mathsf{UseAt}\,(u, idx))}$$

$$\textsc{Read}\ \frac{d : \mathcal{I} \qquad (\varphi_c : \mathcal{W}\,(i), (u_r, u_w)) \in \Gamma \qquad [\mathsf{IsFree}\,(u_r)]}{\Gamma \vdash (\mathsf{readFrom}\, \varphi_c) : \mathcal{P}\,(d, \mathsf{I}) \dashv \Gamma \pm (\varphi_c : \mathcal{W}\,(i), (\mathsf{Use}\,(u_r), u_w))}$$

$$\textsc{ReadAt}\ \frac{d : \mathcal{I} \qquad idx \coloneqq \{{m_i}^{\,i \in 1..j}\} \qquad (\varphi_c : \mathcal{W}\,(i), (u_r, u_w)) \in \Gamma \qquad [\mathsf{IsFreeAt}\,(u_r, idx)]}{\Gamma \vdash (\mathsf{readFromAt}\, \varphi_c\, idx) : \mathcal{P}\,(d, \mathsf{I}) \dashv \Gamma \pm (\varphi_c : \mathcal{W}\,([d\,;n]), (\mathsf{UseAt}\,(u_r, idx), u_w))}$$

Figure 10 Typing Rules for CirQTS.

The rules for gates (Rules Mux, Bin, & Un), casting (Rule Cast), and assignment (Rule Assign) shows how the linearity checking can be propagated *silently* through the type-system. Key to Rules Cast, Mux, Bin, & Un operation is that the premises for ports updates the stateful typing-context. A port can only be used if it is either a variable (Rule Var) or is the result of indexing a port, or a projection of a wire. Threading the updated state left-to-right along the typing rules ensures that ports cannot be reused, as each sub-term uses the latest version of the context possible.

Terms that introduce binders (Rules Ports,Wires,& Gate) are not complicated, and extend the typing-context with a new variable binding with a default usage state.

The final rules deal with indexing ports, and channel projection. Rule Index becomes a variant of Rule Var in which the typing conditions require that instead of the referenced port being completely free, the port must be free at the specified index. Importantly, the return type in Rule Index is not the inner type of φ_p but is, instead, the type of element at the end of the presented index. The rules for projection also follow on from Rule Var in that they resolve references. Whereas Rule Read acts on a port entirely, Rule ReadAt adapts the structure for Rule Index but the usage resource associated with the wire's input endpoint is analysed/updated. Writing to a wire (rules not shown) mirrors those for reading but affect the other resource.

4.4 Wire-Safety

Our approach to reasoning about wire-safety in CirQTS does not differ much from Section 3.3. The core differences relate to how Hardware Graphs are defined, and what it means for a Hardware Graph to be well wired. We highlight the key differences in approach.

To better reason about our graph's topology we extend the definition of Hardware Graph from Definition 2 and label each vertex with its expected in/out degree. More formally:

▶ **Definition 10** (Hardware Graph). *Let $H = \langle vs, es\rangle$ be a directed multigraph graph where vs is a partitioned set of vertices, and es a list of edges. Vertices in H are labelled with their minimum in/out degree, and are defined as $v^{o,i}$ where: o is the expected out-degree; i the expected in-degree; and the shape of each v is taken from $\{S^{(n,0)}, T^{(0,n)}, U^{(n,n)}, N(n,m)^{(n,m)}\}$.*

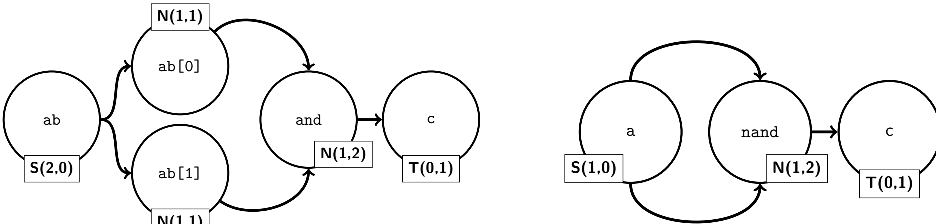
Like the previous definition of a hardware graph we must define what it means for such a graph to be wired. Again, we do so by checking that the expected degrees for each vertex must match the degrees as calculated from the set of edges. Our previous definition, however, is too permissive. Must refine our definition to ensure that the given degrees will match the expected degrees. Moreover, we must ensure that bi-directional ports are used in one direction only, thus we require an exclusive disjunction ($\veebar$) between the calculated in/out degrees and the expected degrees. Formally we present his definition as:

▶ **Definition 11** (Well-Wired Hardware Graph). *Let $H = \langle vs, es\rangle$ be our hardware graph, the wiring of H is well-formed, $\mathcal{W}(H)$, if $\forall v^{(o.i)} \in vs$:*

$$v^{(o.i)}\begin{cases} S^{(o,0)} & \deg^{+}(S^{(o,0)}, es) \equiv o \wedge \deg^{-}(S^{(o,0)}, es) \equiv 0 \\ T^{(0,i)} & \deg^{+}(T^{(0,i)}, es) \equiv 0 \wedge \deg^{-}(T^{(0,i)}, es) \equiv i \\ U^{(o,i)} & \deg^{+}(U^{(o,i)}, es) \equiv o \wedge \deg^{-}(U^{(o,i)}, es) \equiv 0 \\ & \veebar \\ & \deg^{+}(U^{(o,i)}, es) \equiv 0 \wedge \deg^{-}(U^{(o,i)}, es) \equiv i \\ N^{(o,i)} & \deg^{+}(N^{(o,i)}, es) \equiv o \wedge \deg^{-}(N^{(o,i)}, es) \equiv i \end{cases}$$

Figure 11 shows how Figures 1a and 1c can be viewed using the amended hardware graph. With the new hardware graph definition correspondences (Figure 11a) and discrepancies (Figure 11b) can be seen between the expected and given degrees for the vertices in each example. This demonstrates the ability of our new hardware graph definition to more accurately reason about wiring. If we compare these graphs with those in Figure 6 we can better see the accuracy. Both Figure 11a and Figure 6a presents graphs in which there are

no discrepancies. Figure 11b, however, now shows a discrepancy for vertex a in which its expected out degree is one but its given out degree is two, this was not seen in Figure 6b as there is insufficient information to decide.



(a) Figure 1c as a Hardware Graph. **(b)** Figure 1a as a Hardware Graph.

Figure 11 Example Hardware Graphs for CirQTS.

Interpretation of Verliog to hardware graphs is as before, however, to reason more accurately about bit-vector wiring we calculate the degree of a vertex based on the size of its datatype. The shape logic has size one, and bit-vectors have the size of its element multiplied by the size of the bit-vector. We also need to take into account that type-checking will alter the state associated with bound types. Thus, we must return both the accumulated hardware graph (H_o) and the resulting interpretation environment (Σ_o).

▶ **Definition 12** (Interpretation). *Let Σ_i be an interpretation environment and $H_i = \langle vs, ws \rangle$ be a Hardware Graph. We denote the interpretation of a CirQTS design ($\Gamma_i \vdash e : t \dashv \Gamma_o$) as:*

$$(\Sigma_i, H_i)[\![\, e \,]\!] ::= \mathsf{Error} \mid \mathsf{Done}\,(\Sigma_o, H_o, v : [\![\, t \,]\!]_T)$$

where Σ_o and H_o are the updated environment and resulting hardware graph.

With this new interpretation definition we must also update our interpretation soundness result accordingly.

▶ **Theorem 13** (Strong Interpretation Soundness).

$$\frac{\emptyset \vdash e : \mathbb{1} \qquad [\![\, e \,]\!] = \mathsf{Done}\,(\Sigma, H, v)}{\Sigma \equiv \emptyset \qquad v \equiv \bot \qquad \mathcal{W}\,(H)}$$

and also the final lemma which details interpretation.

▶ **Lemma 14** (Interpretation). *Interpretation of a design is not stuck and will return a result containing a valid hardware graph.*

$$\frac{\Gamma_i \vdash e : t \dashv \Gamma_o \qquad \Gamma_i \models \Sigma_i \qquad (\Sigma_i, H_i)[\![\, e \,]\!] ::= (\mathsf{Done}\; res)}{res = (\Sigma_o, H_o, v) \qquad \Gamma_o \models \Sigma_o \qquad v : t' \qquad t' \equiv [\![\, t \,]\!]_T}$$

Note that we also require a consistency relation on outgoing typing contexts and interpretation environments, as well as the inputting ones. This is required, as stepping through each sub-term modifies the type-level state, and returns a new interpretation environment. Further we need to extend the consistency relation to account for context updates.

▶ **Definition 15** (Consistent Interpretation Environments under Context Updates).

$$\textsc{UpdateEnv}\;\frac{\Gamma \models \Sigma}{\Gamma \pm \varphi : t \models \Sigma \pm (\varphi, v : [\![\, t \,]\!]_T)}$$

With these revised definitions we can now reason about the type system of CirQTS and its linearity guarantees. The proof that well-typed designs are well-wired hardware graphs does not change from the simply-typed proof sketch in Section 3.3.

An interesting aspect of our formulation, is that we can use the same wire-safety result defined here to show that designs with Circuits specification will be unsafe as the resulting hardware graphs are not linearly wired.

Unfortunately, we are still not done. CirQTS is too restrictive in its approach to linear wiring. Our type-system, as it stands, will rule out valid designs by removing the ability to fan-out and fan-in when it is benign to do so. Verilog designs require that we can split and join wires together i.e. weaken linearity. The next section discusses how we can extend the *syntax* to support this.

5 Weakening Linearity for Free with new Gates

Figure 12a presents a variant of Figure 1a with many potential linear violations, specifically that the use of `a` is non-linear and that `b` is unused. From this example it is, however, unclear what the designer's intended wiring was. What was accidental: the repeated use of `a`, or specification of `b` as an input? The linearity encoded within the type-system for CirQTS is too restrictive. There will be valid cases, for example driving n-ary logic gates with the same input (Figures 1a and 12a), in which we need to weaken linearity to support valid designs. At the same time though, we need to report accidental dangling and repeated wiring.

```
module Example0(output logic c,
                input logic a,b);
  nand(c, a, a);
endmodule;
```

(a) Unclear usage.

```
module Example0(output logic c,
                input logic a);
  wire logic temp0, temp1;
  split(temp0, temp1, a);
  nand(c, temp0, temp1);
endmodule;
```

(b) Clear usage.

Figure 12 Example showing unclear usage violation, and new syntax to make usage explicit.

Figure 13 presents two new gate primitives that support wire duplication (splitting), and wire joining (collection). With these new primitives designers must now *explicitly* state when wires are to be duplicated (i.e. split) and when they are merged (i.e. drive the same output). Presenting them as primitives addresses issues of backwards compatibility as the SystemVerilog standard [23] supports new gate primitives to be introduced: No new syntax changes are required! More so, our wire-safety result from Section 4.4 need not change either.

With the addition of these new primitives we can use internal wiring to rewrite Figure 12a as Figure 12b if the duplication was required, and if not replace an `a` with the `b`. Again we stress that the implicit wiring decision (duplication) is now an explicit one. Figure 14 illustrate this further by comparing the resulting hardware graph and circuit diagrams for Figure 12b.

6 Mechanisation and Realisation in a Dependently-Typed Language

We have mechanised the implementations of Circuits and CirQTS using Idris2 [10, 9], a general purpose dependently-typed language. Leveraging Idris2's support for dependent types we can provide both a verifiable *practical* type-checker for each language, but also formally

$$e ::= \ldots \mid \mathsf{collect}\,(e, e, e) \mid \mathsf{split}\,(e, e, e)$$

(a) Syntax.

$$\textsc{Collect}\ \frac{\Gamma_1 \vdash o : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \dashv \Gamma_2 \quad \Gamma_2 \vdash i_a : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_3 \quad \Gamma_3 \vdash i_b : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_4}{\Gamma_1 \vdash \mathsf{collect}\,(o, i_a, i_b) : \mathcal{G} \dashv \Gamma_4}$$

$$\textsc{Split}\ \frac{\Gamma_1 \vdash o_a : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \dashv \Gamma_2 \quad \Gamma_2 \vdash o_b : \mathcal{P}\,(\mathsf{Logic}, \mathsf{O}) \dashv \Gamma_3 \quad \Gamma_3 \vdash i : \mathcal{P}\,(\mathsf{Logic}, \mathsf{I}) \dashv \Gamma_4}{\Gamma_1 \vdash \mathsf{split}\,(o_a, o_b, i) : \mathcal{G} \dashv \Gamma_4}$$

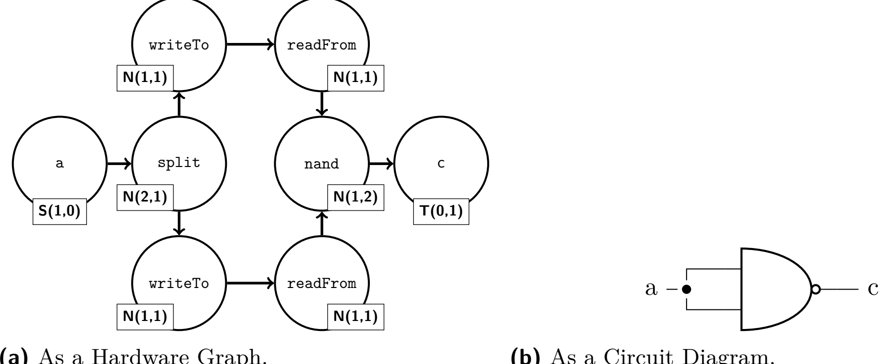
(b) Typing Rules.

```
primitive collect ( output logic o, input logic a, b);
   assign o = a;
   assign o = b;
endprimitive;

primitive split ( output logic a, b, input logic i);
   assign a = i;
   assign b = i;
endprimitive;
```

(c) Verilog Primitives.

Figure 13 New Primitive Gates to Safely “Weaken” Linearity.



(a) As a Hardware Graph. **(b)** As a Circuit Diagram.

Figure 14 Figure 12b as a Hardware Graph and Circuit Diagram.

reason about wire-safety. Thus, our implementations primarily support the type-checking of netlists (modulo syntax restrictions from Section 2) written in Verilog. With Circuits we can check if it is valid Verilog, with CirQTS we can check if the netlists are linearly wired. Further, for both type-checkers we incorporated a soundness check (modelled on the type-safety result from CirQTS) to show which designs accepted by Circuits are in-fact *wire unsafe*.

Both Circuits and CirQTS have been realised as intrinsically well-typed well-scoped Embedded Domain Specification Languages (*EDSLs*) within Idris2 using well documented techniques [27, 2, 29, 25]. The terms, types, and usages for our languages translate directly into standard dependently (and non-dependently) typed data structures. Reading of valid Verilog netlists comes from a frontend Domain Specification Language (*DSL*) coupled with an elaborator (also known as the type, scope, and usage checker) to construct intrinsically typed terms.

There are three interesting aspects with our approach to the mechanisation: relation between soundness check and our formal proof; intrinsic linearity checking; and error reporting.

Existing work has demonstrated how *well-typed* definitional interpreters can be realised within dependently-typed languages [40, 35] to provide a mechanised runtime. We borrow this approach to not only reason about our type-safety result (wire-safety) as code (i.e. its mechanisation) but make the proof an integral aspect of our tool's operation.

The core *EDSLs* representing CirQTS are intrinsically typed both structurally and sub-structurally. Our approach is inspired by *Leftover Typing* [1] in which variable usage updates a type-level state iff a variable is "free" to be used. We use a bespoke list quantifier (presented in Figure 15) paired with a generic update datatype and decision procedure (Figure 16) to provide a type-safe updating of the typing context respective to the predicate being asserted. The type-level constraints that act on variables, that enable our linearity, are instances of these decidable constructs.

```
data Elem : (p  : type -> Type)
         -> (x  : type)
         -> (xs : List type)
         -> Type
  where
    Here : (prfSame : x = y)
        -> (prfSat  : p y)
                   -> Elem p x (y::xs)
    There : (rest : Elem p x     xs)
                 -> Elem p x (y::xs)
```

(a) Decidable Predicate.

```
isElemSat : DecEq type
         => (f : (x : type)
              -> DecInfo (n x) (p x))
         -> (x  : type)
         -> (xs : List type)
               -> DecInfo (ElemNot n p x xs)
                          (Elem      p x xs)
```

(b) Decision Procedure.

Figure 15 Quantification over Lists.

The standard decidable predicate (`Dec`) for decision procedures is lossy when reporting errors, such that it is impossible to know at runtime why the procedure failed. Such predicates are analogous to the `Maybe` datatype. We need an `Either` equivalent to help report useful information when a decidable procedure fails: `Dec` needs to be decidedly informative. Constructive negation is an interesting area of research that supports positive information to be used when reporting errors [5]. Inspired by constructive negation we have used `DecInfo`, which is similar to the definition for decidable but is further indexed by a type that carries showable error messages as well as proofs of contradition. We see this in the type signature for `isElemSat` in Figure 15, where `ElemNot` records why the procedure failed and requires an informative decision procedure: empty list, or element does not satisfy

```
data Update : (o,n : type -> Type)
           -> (use : (x : type) -> (prf0 : o x) -> (y : type) -> (prfN : n y) -> Type)
           -> (old : List type) -> (prf : Elem o x old)
           -> (new : List type)
           -> Type
  where
    UHere : {0 u : (x : type) -> (p0 : o x) -> (y : type) -> (pN : n y) -> Type}
         -> (use : u x pX y pY)
                -> Update o n u (x::xs) (Here Refl pX)
                                (y::xs)
    UThere : (rest : Update o n u     xs          l       ys)
                  -> Update o n u (x::xs) (There l) (x::ys)
```

(a) Instructions.

```
update : (use : {x : type} -> (prf0 : o x) -> (y : type ** prfN : n y ** u x prf0 y prfN))
      -> (old : List type)
      -> (prf : Elem o x old)
      -> (new ** Update o n u old prf new)
```

(b) Procedure.

Figure 16 Safe List Updating.

the provided predicate. Sadly `DecInfo` is not truly positive as it carries a *proof of void*. Making it *positively negative*, that is using only positive information to represent proofs and refutations, is ongoing work.

Using Idris2 to construct our proof-of-concept type-checkers demonstrates the power of mechanising our proofs in a practical general purpose language that supports dependent types.

7 Evaluation

We investigated the efficacy of CIRCUITS and CIRQTS through creation of an illustrative testbench. The testbench was designed to illustrate how well-known circuits can contain implicit wiring decisions (checked against CIRCUITS), and that by using CIRQTS we can make the wiring explicit. Other examples chosen were inspired by various online Verilog tutorials and sought to test the ability of our type-systems to type-check existing designs (CIRCUITS) and to reason about quantitative wire usage for those designs (CIRQTS). The well-known designs chosen were flip flops, full and half adders, and gate-level multiplexers. We provided normal and linear variants of these well-known designs, and Table 1 presents salient modelling information comparing the two variants.

Table 1 Salient Modelling Information for Core Netlists used during Benchmarking.

	Non-Linear				Linear			
Netlist	Inputs	Outputs	Wires	Gates	Inputs	Outputs	Wires	Gates
FlipFlopD	2	2	3	5	2	2	11	9
FullAdder	3	2	3	5	3	2	11	9
HalfAdder	2	2	0	2	2	2	4	4
Mux21	3	1	3	4	3	1	5	5

All examples presented were checked against: CIRCUITS, CIRQTS, and Verilator. We performed these checks for two reasons. First, we wanted to show that designs that were admitted by CIRCUITS but failed the soundness checker, then failed to type check using

CirQTS, and that valid linear designs could also be checked using Circuits. Second, Verilator is a well-known open source static analysis and simulation tool for SystemVerilog. Comparison against Verilator provides a validation step that CirQTS is comparable to existing tooling and that, unlike commercial static analysis tools, can be incorporated into our supplied artefact due to its small installation footprint and permissive licence.

For all examples, we found that type-checking time was negligible, and comparable to Verilator, whether an error was found or not. From this experimentation we made the following observations.

We can Retrofit Linear Types onto Verilog NetLists

The gate-level syntax is the last intermediate representation before fabrication (or deployment to an FPGA). For this subset of Verilog chosen for our featherweight language, we have been successful in introducing linear wirings. From this subset the challenge will be how to promote linearity to the remainder of Verilog, and of course SystemVerilog.

Verbosely Made Implicit Wirings Explicit

The new primitives presented in Section 5 support valid weakening of linearity, specifically splitting and joining of wires. Such weakening, however, comes at the cost of design verbosity. Compare in Figure 17, for example, the implementations of a half-adder in both Circuits and CirQTS. For each wire split, or joined, new wires must be presented.

```
module Example(output logic sum, carry
              ,input  logic a b);

  xor g1(sum,a,b);
  and g2(carry,a,b);

endmodule;
```

(a) Circuits.

```
module Example(output logic sum, carry
              ,input  logic a b);

  wire logic a1,a2,b1,b2;

  split sa(a1,a2,a);
  split sb(b1,b2,b);

  xor g1(sum,a1,b1);
  and g2(carry,a2,b2);

endmodule;
```

(b) CirQTS.

Figure 17 Half-Adder in both Circuits and CirQTS.

Although in the linear setting such redundant wires can be optimised away, their addition to the language makes it verbose. If we are to introduce such linear restrictions elsewhere in SystemVerilog, that is for synthesisable terms, the end user will be presented with a *needlessly* verbose language. Thus raising the question of how best to retrofit existing *HDLs* with quantitative types such that linearity does not get in the way. Perhaps we should not take this approach, and instead look at providing annotations as seen in other extensions to Verilog [48].

Some Ports are Linear but Some are More Linear Than Others

One aspect we found interesting is that CirQTS performs more finegrained resource tracking for inputting n-dimensional bit-vectors when compared to Verilator. Although Verilator can identify unused inputs in designs, for logic and bit-vectors, it does not consistently report

unused inputs for n-dimensional bit-vectors. Consider, for example, the following illustrative netlists presented in Figure 18. The netlist within Figure 18a *is* caught by Verilator as the output port (`out`) is not fully used. The netlist within Figure 18b *is not* caught by Verilator even though the input port (`bc`) is not fully used.

```
module Example(output logic[1:0][1:0] out, input logic a,b);

   and n1(out[0][1],a,b);

endmodule;
```

(a) Caught by Verilator.

```
module Example(output logic out, input logic[1:0][1:0] bc);

   and n1(out, bc[0][1], bc[1][0]);

endmodule;
```

(b) Not Caught by Verilator.

Figure 18 Illustrative Netlists with sparsly used input and output ports.

We noted the potential discrepancy in Section 4.3 when discussing the stopping condition, that is when and how we check for linear usage. Such difference is because unused signals can be safely removed from the design. Specifically, inputting ports can be left unused if they are never connected to anything else in the entire design, and module outputs need to be driven as they could potentially be used later in the design to drive another part of the design.

Such differences in granularity of reporting between CirQTS and Verilator does raise interesting questions about the semantics (veracity) of linear wiring. Whilst outputting ports must be linear in usage, inputting ports are affine. Linear usage restrictions are incredibly restrictive, terms must be used exactly once. In certain circuit designs it is okay to leave the input of a circuit dangling. Perhaps our quantitative type-system requires fine-tuning to ensure that we are affine for inputs, but linear for outputs, or even gradations to specify how much a signal must be used as seen in Granule? Although we could also extend our syntax to support "noops" primitives that *consumes* purposefully dangling wires.... That being said, such allowed dangling is very much a *behavioural aspect* that requires more information that we do not presently give in the type-system about the intended behaviour being realised. Our use of quantitative typing is purely structural and not behavioural. More descriptive behavioural typing is required, and that is a different kettle of fish entirely.

8 Related Work

Section 4 discussed the limitations of modelling linear wiring in existing *QTS*, and mentioned that Verilator performs an extrinsic graph-based check. This section looks to the wider world of *HDL* design and verification, and where our work resides.

8.1 Formal Models of Verilog

There have been several attempts at verifying the *behaviour* of Verilog designs [28, 31, 26]. Many of these works are limited by looking at the behaviour of modules in isolation and not their composition. CirQTS complements these attempts by providing *a* formal account for wiring. Integrating our type-system into existing behavioural semantics would be advantageous.

8.2 BlueSpecVerilog

Verilog-like languages view hardware in the traditional sense of wiring up boxes/gates that communicate concurrently. BlueSpecVerilog (*BSV*) [32] takes a different route by allowing designers to take a slightly higher-level view of hardware and describe circuits as atomic actions on stateful elements i.e. registers. *BSV* builds upon SystemVerilog with strong types and known techniques lifted from the well-known functional language Haskell. Kami and Kôika are two rule-based hardware description languages that capture core behavioural semantics of *BSV* [11, 8]. Both languages are presented as *EDSLs* written in the Coq Theorem Prover. Our approach to wire-safety can compliment existing work relating to *BSV* by enforcing wiring decisions to be explicit rather than implicit. How our work intersects practically is an area for future investigation.

8.3 High-Level Synthesis

Similar to the realisation of Kami and Kôika, High-Level Synthesis (*HLS*) typically approaches hardware design as *EDSLs* in high-level languages such as Haskell and Scala that are then synthesised into hardware descriptions i.e. netlists. Popular *HDLs* that take this approach are PyRTL [13], Lava [20], Chisel [6], Delite [43], ReWrire [39], and Cλash [42]. By embedding their work in an existing general-purpose language these languages can take full advantage of the host language's eco-system to provide guarantees about program composition and behaviour. For instance *Wire Sorts* [12] is an extension to PyRTL to reason about wirings. Many of the approaches described here model circuit level designs as combinators, and treat hardware components as functions that can be translated directly to SystemVerilog. Our work intersects through provision of a means to account for how wires are generated and used in the synthesised designs.

More interestingly are *HDLs* embedded in languages that are *QTS* aware e.g. Haskell. It will be interesting to see how Linear Haskell can be used by the Lava and Cλash to enforce linear wiring.

8.4 Cava

Coq + Lava (*Cava*) is an industrial research project[2] that looks at providing an *easy to verify HDL* for "network-style" low level circuits, rather than higher level processor designs as seen in *BSV*. *Cava* primarily looks to verify circuit behaviour. Much like Kami and Kôika, *Cava* is an *EDSL* in the Coq Theorem Prover. Interestingly, *Cava* and Circuits/CirQTS share similarities in that they both describe netlists. We differ in that *Cava* syntax is not verilog-like when representing wiring but ours is.

8.5 Formal Models for High-Level Synthesis

We end our discussion of related work by looking at theoretical models for hardware. The *Geometry of Synthesis* series [16, 19, 17, 15, 18] looks at leveraging *Geometry of Interaction* style interpretation as a way to better understand the behavioural and structural aspects of hardware design. Especially, they target a *HLS* combined with category theory to better tell the synthesis story. Fortunately, existing work (Section 8.1) has looked at the synthesis story for Verilog. We have not explored these stories to include *QTS*, we have short-circuited this story by jumping straight to the end. That being said we are, however, interested in exploring how our *QTS* can be included.

[2] `https://project-oak.github.io/silveroak/`

9 Conclusion

CIRQTS demonstrates that quantitative typing, specifically, the idea of linearity, is not just for general purpose programming languages. We can use *fancy types* to be reason about hardware design. With Verilog being so integral to hardware design, our efforts at formal verification (through typing) of the language itself has helped move existing extrinsic checks (as performed by static analyses and simulation, and also caught during synthesis) directly into the type-system, and become an integral part of the language itself. With the correct type-system, errors that were once caught late in the design process will now be caught earlier helping to increase design productivity and enhance the trustworthiness of developed designs.

A question remains though over what precise usage restrictions should be applied to hardware terms, and presented to the system designer. Is it the case that all terms should be restricted equally, or that some terms be restricted less equally than others? Moreover, we have not required designers to annotate their designs. Would it be better to extend/embed Verilog with linear annotations? If such restrictions are given, would some form of gradations *a la* Granule on terms be better for specifying usage? These are all exciting open areas of investigation.

A secondary area of interest is where we place usage information in CIRQTS. We have used usage *annotations* on binders, as is common in QTS. An alternative approach would be to index types directly with their usage, rather than as usage annotations on binders. We think this is an equally valid approach but will require further investigation over soundness and suitability.

Regardless, we have demonstrated how to *retrofit* a sub-structural type-system onto existing syntax modulo minor syntax elaboration. Retrofitting means that we do not need new languages that existing designs must be ported to. However, looking at what syntax changes are needed to make Verilog more conducive to linear wiring, or new primitives, is also a worthy area of future investigation. Our retrofitting approach also opens up future work on what else needs changing (in the type-system) when bringing quantitative typing back up the synthesis to higher-levels of abstraction. A key aspect of which will be design composition in the face of modules and interfaces, and the quasi-dependently-typed parameterisation of modules and interfaces.

Finally, within Idris2 the none-one-tonne rig [30] provides a distinction between code available at runtime $(1, \omega)$ and its usage, and code only available at compiletime (0). We have used elements of QTT to reason about wire usage, but not their movement through the *fourth dimension*: time. Whereas we think of a program's journey as going first through the compiletime and then the runtime, SystemVerilog is a language where designs will journey through many more "times": design, testing, simulation, synthesis, placement & routing, and execution. A fascinating area of investigation would be to see if QTT, or QTT-like structures, can help to better reason about terms as they traverse through the many times of SystemVerilog. Can terms be stratified according to *when* in time they are expected to be? We hope so!

References

1 Guillaume Allais. Typing with Leftovers – A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. In *23rd International Conference on Types for Proofs and Programs, TYPES 2017, May 29-June 1, 2017, Budapest, Hungary*, pages 1:1–1:22, 2017. doi:10.4230/LIPIcs.TYPES.2017.1.

2 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *PACMPL*, 2(ICFP):90:1–90:30, 2018. doi:10.1145/3236785.

3 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017. URL: http://dl.acm.org/citation.cfm?id=3009866, doi:10.1145/3009837.3009866.

4 Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65, 2018. doi:10.1145/3209108.3209189.

5 Robert Atkey. Data Types with Negation. Extended Abstract (Talk Only) at Ninth Workshop on Mathematically Structured Functional Programming, Munich, Germany, 2nd April 2022, 2022. URL: https://youtu.be/mZZjOKWCF4A.

6 Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a Scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225. ACM, 2012. doi:10.1145/2228360.2228584.

7 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158093.

8 Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of Bluespec: a core language for rule-based hardware design. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 243–257. ACM, 2020. doi:10.1145/3385412.3385965.

9 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

10 Edwin C. Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.9.

11 Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP):24:1–24:30, 2017. doi:10.1145/3110268.

12 Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. Wire sorts: a language abstraction for safe hardware composition. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 175–189. ACM, 2021. doi:10.1145/3453483.3454037.

13 Deeksha Dangwal, Georgios Tzimpragos, and Timothy Sherwood. Agile Hardware Development and Instrumentation With PyRTL. *IEEE Micro*, 40(4):76–84, 2020. doi:10.1109/MM.2020.2997704.

14 Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. Verilog HDL and Its Ancestors and Descendants. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi:10.1145/3386337.

15 Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 363–375. ACM, 2007. doi:10.1145/1190216.1190269.

16 Dan R. Ghica. The Geometry of Synthesis – How to Make Hardware Out of Software. In *Mathematics of Program Construction – 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, pages 23–24, 2012. doi:10.1007/978-3-642-31113-0_3.

17 Dan R. Ghica and Alex I. Smith. Geometry of Synthesis II: From Games to Delay-Insensitive Circuits. In Michael W. Mislove and Peter Selinger, editors, *Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2010, Ottawa, Ontario, Canada, May 6-10, 2010*, volume 265 of *Electronic Notes in Theoretical Computer Science*, pages 301–324. Elsevier, 2010. doi:10.1016/j.entcs.2010.08.018.

18 Dan R. Ghica and Alex I. Smith. Geometry of synthesis III: resource management through type inference. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 345–356. ACM, 2011. doi:10.1145/1926385.1926425.

19 Dan R. Ghica, Alex I. Smith, and Satnam Singh. Geometry of synthesis IV: compiling affine recursion into static hardware. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 221–233, 2011. doi:10.1145/2034773.2034805.

20 Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages – 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2009. doi:10.1007/978-3-642-16478-1_2.

21 Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight Go. *Proc. ACM Program. Lang.*, 4(OOPSLA):149:1–149:29, 2020. doi:10.1145/3428217.

22 Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485494.

23 IEEE. *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2017 edition, February 2018. doi:10.1109/IEEESTD.2018.8299595.

24 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

25 Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

26 Wilayat Khan, Alwen Tiu, and David Sanán. VeriFormal: An Executable Formal Model of a Hardware Description Language. In Abhik Roychoudhury and Yang Liu, editors, *A Systems Approach to Cyber Security – Proceedings of the 2nd Singapore Cyber-Security R&D Conference (SG-CRC 2017), Singapore, February 21-22, 2017*, volume 15 of *Cryptology and Information Security Series*, pages 19–36. IOS Press, 2017. doi:10.3233/978-1-61499-744-3-19.

27 Wen Kokke, Jeremy G. Siek, and Philip Wadler. Programming language foundations in Agda. *Sci. Comput. Program.*, 194:102440, 2020. doi:10.1016/j.scico.2020.102440.

28 Andreas Lööw. Lutsig: a verified Verilog compiler for verified circuit development. In Cătălin Hrițcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 46–60. ACM, 2021. doi:10.1145/3437992.3439916.

29 Conor McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In Bruno C. d. S. Oliveira and Marcin Zalewski, editors, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010*, pages 1–12. ACM, 2010. doi:10.1145/1863495.1863497.

30 Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.

31 Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Rosu. A formal executable semantics of Verilog. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*, pages 179–188. IEEE Computer Society, 2010. doi:10.1109/MEMCOD.2010.5558634.

32 Rishiyur S. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*, pages 69–70. IEEE Computer Society, 2004. doi:10.1109/MEMCOD.2004.1459818.

33 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *PACMPL*, 3(ICFP):110:1–110:30, 2019. doi:10.1145/3341714.

34 Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In Wei-Ngan Chin and Aquinas Hobor, editors, *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, page 1. ACM, 2012. doi:10.1145/2318202.2318203.

35 Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34, 2018. doi:10.1145/3158104.

36 Dimitri Racordon and Didier Buchs. Featherweight Swift: a Core calculus for Swift's type system. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 140–154. ACM, 2020. doi:10.1145/3426425.3426939.

37 John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. doi:10.1023/A:1010027404223.

38 John C. Reynolds. Definitional Interpreters Revisited. *High. Order Symb. Comput.*, 11(4):355–361, 1998. doi:10.1023/A:1010075320153.

39 Thomas N. Reynolds, Adam M. Procter, William L. Harrison, and Gerard Allwein. A core calculus for secure hardware: its formal semantics and proof system. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 – October 02, 2017*, pages 122–131. ACM, 2017. doi:10.1145/3127041.3127048.

40 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.

41 Jeremy Siek. Type Safety in Three Easy Lemmas. Online, May 2013. URL: https://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.

42 Gerard J. M. Smit, Jan Kuper, and Christiaan P. R. Baaij. A mathematical approach towards hardware design. In Peter M. Athanas, Jürgen Becker, Jürgen Teich, and Ingrid Verbauwhede, editors, *Dynamically Reconfigurable Architectures, 11.07. – 16.07.2010*, volume 10281 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2010. doi:10.4230/DagSemProc.10281.3.

43 Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embedded Comput. Syst.*, 13(4s):134:1–134:25, 2014. doi:10.1145/2584665.

44 Philip Wadler. Linear Types can Change the World! In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990.

45 David Walker. *Advanced Topic in Types and Programming Languages*, chapter Substructural Type Systems, pages 3–43. The MIT Press, 2004.

46 James Wood and Robert Atkey. A Linear Algebra Approach to Linear Metatheory. In Ugo Dal Lago and Valeria de Paiva, editors, *Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity & TLLA @ IJCAR-FSCD 2020, Online, 29-30 June 2020*, volume 353 of *EPTCS*, pages 195–212, 2020. `doi:10.4204/EPTCS.353.10`.

47 Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994. `doi:10.1006/inco.1994.1093`.

48 Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 503–516, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2694344.2694372`.