Categorical Semantics and Modal Types for Hardware Description

Kai Pischke

MCompSci (Part C)
Trinity Term 2023

UNIVERSITY OF OXFORD
DEPARTMENT OF COMPUTER SCIENCE

9972 words (using texcount)

Abstract

We present a new lambda calculus-like language for reasoning about hardware description with higher-order function abstractions. We show that separating the types of wires and higher-order constructs lets us avoid the name-sharing problem commonly encountered by functional hardware description languages. Using nominal sets and equivariant functions, we give new categorical semantics to the typed language, which keeps wire and circuit terms separate while allowing each to contain free variables of either variety. Furthermore, we prove normalisation and soundness results, showing that all well-typed terms can be reduced to a simpler fragment of the language in a finite number of steps while preserving the semantics. We also consider the application of guarded types for describing synchronous circuits and give a behavioural semantic model of a synchronous term language using the topos of trees. Finally, we implement a proof-of-concept compiler featuring unification-based typed inference, which is able to extract synthesisable Verilog output from a range of examples.

Acknowledgements

I am extremely grateful to my supervisor, Sam Staton, whose passion for programming languages has been a constant source of inspiration. I also thank Sean Moss for his valuable feedback and for sharing some of his extensive knowledge of category theory. Nobuko Yoshida provided many interesting discussions, and I very much appreciate her patience in listening to some of my early ideas. Finally, I would like to thank Jon Fowler for initially introducing me to hardware description. I, of course, also would like to express my appreciation for all my wonderful friends, without whom my time at Oxford would not have been the same.

Contents

Chapter 1

Introduction

1.1 Motivation

Modern computer technology is becoming more complex, and intensive tasks are increasingly being offloaded to specialised hardware. While programming languages used to design software have developed over time to provide more abstractions, hardware description languages such as Verilog and VHDL remain very low-level, tedious to use, and offer few safety guarantees. The purpose of this thesis is to address some of these challenges by exploring the semantics of a minimal functional hardware description calculus with a robust type system, with the goal of demonstrating possible improvements to the usability and safety of hardware description languages.

1.2 Existing Work

The use of functional programming languages for hardware description has a long history. Functional approaches to hardware design were initially pioneered in the 1980s through the work of Mary Sheeran and others who introduced the early hardware description languages Ruby [30] and μFP [31]. These languages made the critical insight that recursive definitions in functional languages served as a natural specification for feedback loops in hardware. This work led to the development of the Lava language [8], which emphasised using Haskell for multiple interpretation of hardware definitions. The idea was to facilitate both simulation and synthesis. Christiaan Baaij built on the foundations of Lava to develop the modern functional hardware description language Cλash [4] in Haskell, leading to some industrial adoption. Other languages, such as Bluespec [3], have also focused on industrial use, drawing on both Verilog and Haskell.

More recently, the success of string diagrams and categorical semantics for quantum [1] and probabilistic [34, 33] programming models has led to a line of work by Dan Ghica formalising circuits in a categorical manner [15, 16, 17]. This has opened up the possibility of purely equational reasoning about circuits, which is beneficial from the standpoint of hardware verification and the development of higher-level abstractions for hardware design.

Much of the existing work on functional hardware description has focused on embedded languages in Haskell, and much of the current work on circuit semantics has yet to be applied to functional languages. Hence, there is significant scope for work bridging the gap between existing formal and foundational approaches to hardware semantics and the more practical functional approaches to hardware design. While the infrastructure of Haskell is beneficial from an industrial standpoint, it is also helpful to study the semantics from the perspective of a minimal calculus. The success of categorical approaches to domain-specific programming language applications such as probabilistic programming [10] suggests that this is a promising direction for further exploration.

1.3 Contributions

This thesis introduces a new term language for reasoning about hardware, both combinational (Chapter 2) and synchronous (Chapter 3). In order to allow beta-reduction of higher-order functions without allowing the reduction of circuits themselves, we split our language into separate classes of wire and circuit terms. This poses an interesting challenge in defining rigorous semantics, as each class of terms can contain free variables from the other class. We solve this problem by proposing novel semantics using Markov categories [33] and nominal sets [14]. Nominal sets are a recent development in formally defining structures modulo α-equivalence by studying permutation actions on names. To the best of our knowledge, neither Markov categories nor nominal sets have been previously applied to hardware description.

We then give a modal type system for preventing purely combinational cycles, with a semantics based on the topos of trees [7]. The topos of trees provides a framework to reason about guarded recursion by looking at the set of possible values at each clock cycle and the ways in which these sets are related. This model has previously only been applied to software-based synchronous programs. To summarise, our significant contributions are:

- A new term language for combinational and synchronous circuits (Section 2.5)
- A new categorical semantic model using nominal sets and equivariant functions for the combinational fragment (Section 2.9)

- A new categorical behavioural model using the topos of trees for the synchronous fragment (Section 3.2.2)
- A new type system using the later modality for guarded feedback and recursion (Section 3.3)
- A type inference algorithm (Subsection 4.4 in Chapter 4)
- A translation algorithm and full implementation of a compiler producing Verilog output from the term language (Chapter 4)

We additionally prove two main theorems which motivate the correctness of the implementation in Chapter 4. In particular, the soundness result (Theorem 2.9.2) shows that the simplification steps made during compilation preserve the semantics. The normalisation result (Theorem 2.7.1) suggests that every well-typed program is compiled in a finite number of steps. Together, these theorems show that the implementation always terminates and produces a semantically correct output circuit.

1.4 Structure

- Chapter 2 introduces a term language and semantics for combinational circuits.
- Chapter 3 introduces the later modality and extends the term language to synchronous circuits.
- Chapter 4 describes the structure of the compiler implementation.
- Chapter 5 gives examples and empirical results for the compiler.
- Chapter 6 concludes the discussion with an overview and directions for future work.
- Chapter 7 contains the technical details of longer proofs and generated output code.

Chapter 2

Combinational Circuits

This Chapter introduces a simple, functional calculus for describing combinational circuits. We start by describing some of the interesting semantic features of hardware description (Section 2.2). We then present a categorical view of the structure of circuits (Sections 2.3 and 2.4). This leads to the definition of the term language λ_{comb} (Section 2.5). Next, we discuss a type system for the term calculus (Section 2.6) and finish by giving our language an equational theory and categorical semantics (Sections 2.7 and 2.9). We also prove two key results: a normalisation theorem for well-typed terms and a soundness result for our equational theory. These theorems will become relevant in Chapter 4 when we consider the implementation of a compiler for this term language.

2.1 Hardware Description Describes Structure

Hardware description is a very low-level endeavour. While we are often happy for compilers to make significant changes to the structure of our software so long as they preserve the functionality, this type of *extensional* abstraction is much more challenging to achieve at the hardware level. Furthermore, unlike software, where copying references is cheap, hardware circuits do not support cheap copying of circuitry, and so resource utilisation and related concerns, such as tradeoffs between circuit depth and size, are of fundamental importance. We are, therefore, primarily interested in languages for describing the structure of circuits rather than just modelling their behaviour, and we will start by looking at *intensional* semantic models for circuits.

2.2 Hardware as an Effect of Synthesis

There are many types of effects in normal software programming. Input/output, nondeterminism, and randomness are all effects. While most imperative languages do not offer careful control of effects via the type system, more principled approaches to effectful computation have been developed. The two main approaches are effect type systems [20] and monads [25]. More recent work has focussed on algebraic effects [5, 32], which tries to look at the algebraic properties satisfied by different computational effects.

It seems counterintuitive to think of hardware description as an effect. After all, there is not anything unusual going on as a side effect of the synthesis step. However, the structure of the hardware we synthesise reflects the structure of the source expression. Applying a circuit causes that circuit to be synthesised, so hardware description is not pure. This can be seen by considering the following two expressions, which we will interpret as circuits with the free name a as an input wire.

$$\text{let } b = f\ a \text{ in } (b, b) \qquad \text{and} \qquad (f\ a, f\ a)$$

If we represent them diagrammatically, we see that they represent different circuits.

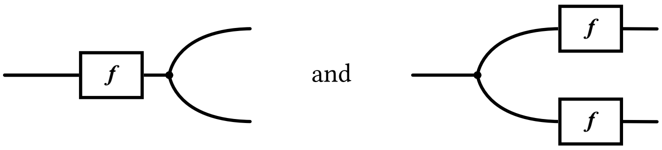


Figure 2.1: Copying and sharing circuitry.

In particular, one of the circuits uses twice as many gates as the other, so although they implement the same behaviour, they do not describe equivalent hardware. This problem has been referred to as the node-sharing problem in the literature. Various solutions have been proposed, such as using immutable references [11], manually tagging nodes with names [4], and using monads [4, 8].

However, ambiguity in determining what is shared and what is copied is only a problem if we constrain the semantics to validate the substitution:

$$\text{let } x = e \text{ in } e' \equiv e'[e/x]$$

If we instead incorporate this effect implicitly as a byproduct of let binding, we do not have to resort to explicit treatment of effects. A similar scenario arises in any call-by-value language with effects. For example, the following two probabilistic programs are also not equivalent.

$$\text{let } x \ = \text{random() in } (x, x) \qquad \text{and} \qquad (\text{random()}, \text{random()})$$

Both hardware and probabilistic programming are *commutative* effects. Although binding a name to a circuit causes sharing at the hardware level, the order in which we bind two different names is irrelevant. We can see that swapping the order of two nested let expressions describes the same synthesised hardware, as is shown in Figure 2.2.

$$\text{let } a = \ldots \text{ in let } b = \ldots \text{ in } h(a, b) \qquad \equiv \qquad \text{let } b = \ldots \text{ in let } a = \ldots \text{ in } h(a, b)$$

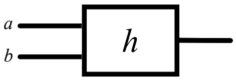


Figure 2.2: Commutativity example.

Commutativity suggests that it is undesirable to use a metalanguage of effects, such as a monadic language, which makes the control flow explicit. This would impose an unnecessarily arbitrary ordering on the evaluation.

There is also another similarity between hardware circuits and probabilistic programs: any names not used in an expression can safely be ignored. This is known as discardability.

$$\text{let } a = \ldots \text{ in } () \qquad \equiv \qquad ()$$

As is observed in [33], many effects with this structure are modelled by Markov Categories. We will see more formally in Section 2.4 that circuit description can be modelled in precisely this way.

2.3 Background on Category Theory

We will use category theory to provide a synthetic setting for modelling circuits. Familiar readers may safely skip this brief overview; those looking to learn more can find a detailed introduction in [19].

We start by defining a category, which is simply a collection of objects and arrows between objects satisfying some simple axioms.

Definition 2.3.1: Category

A category $\mathbb{C}$ consists of:

1. a class of objects, $\text{Ob}(\mathbb{C})$
2. a class of morphisms, $\text{Hom}(\mathbb{C})$.
3. a composition operator $\circ$

Each morphism in $\text{Hom}(\mathbb{C})$ is associated with a source and target object in $\text{Ob}(\mathbb{C})$ and we write $m : X \to Y$ to show that a morphism m has source object X and target object Y. Morphisms can be composed with an associative operator $\circ$ which takes two compatible morphisms $m_1 : X \to Y$ and $m_2 : Y \to Z$ and produces a composite morphism $m_2 \circ m_1 : X \to Z$. We require $\circ$ to be associative and for each object X to have a unique identity morphism $\text{id}_X : X \to X$ which acts as a left and right unit for $\circ$.

If the class of morphisms between any two objects are a set, we say the category is locally small, and most examples of categories we will use will have this property. Therefore, we will sometimes refer to the Hom-sets rather than the Hom-classes of a category.

Mappings between categories which preserve structure are known as functors.

Definition 2.3.2: Functor

A functor F between categories A and B is a mapping of objects in $\text{Ob}(A)$ to $\text{Ob}(B)$ and morphisms in $\text{Hom}(A)$ to $\text{Hom}(B)$ such that $F(\text{id}_X) = \text{id}_{F(X)}$ and $F(f \circ g) = F(f) \circ F(g)$.

The topos of trees discussed in section 3.2.2 will, in fact, be a category whose objects are themselves the functors between other categories, and the later modality will correspond with a functor from this category to itself (an endofunctor).

Similarly to how functors are structure-preserving mappings between objects, we can define natural transformations which are structure-preserving mappings between functors.

Definition 2.3.3: Natural Transformations

A natural transformation η is a mapping between functors $F : A \to B$ and $G : A \to B$, defining for each object in X a morphism $\eta_X : F(X) \to G(X)$. We refer to this morphism as the component of η at X. It must satisfy the condition for any morphism $f : X \to Y$ in the category A that $\eta_Y \circ F(f) = G(f) \circ \eta_X$.

We will primarily work with categories equipped with some notion of parallel composition. These categories are known as monoidal categories.

Definition 2.3.4: Monoidal Category

A monoidal category $(\mathbb{C}, \otimes, \mathcal{I})$ is a category $\mathbb{C}$ equipped with a tensor product $\otimes$. This tensor product is a bifunctor $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ which is associative with identity $\mathcal{I}$ (up to natural isomorphisms satisfying coherence conditions). If the natural isomorphism can be replaced by equality, we call the category strict monoidal, and if there is a swap isomorphism $\mathbf{swap}_{AB} : A \otimes B \to B \otimes A$ satisfying $\mathbf{swap}_{AB} \circ \mathbf{swap}_{BA} = \mathbf{id}_{A \otimes B}$, we say the category is symmetric monoidal.

When we have the product $\times$ as $\otimes$, we say the category is cartesian. And if we can additionally associate an object to each class of morphisms $\mathrm{Hom}(X, Y)$ (which we call the internal Hom) we say the category is cartesian-closed. This allows us to talk about categories which, in some sense, contain function spaces, and we will use a cartesian closed category to represent the higher-order constructs of our term language in section 2.9.

2.4 Categorical Models of Combinational Circuits

We will formalise a new structural model of circuits, in a similar vein to Ghica's behavioural model [16], and will frame this using Markov categories, originally developed to study probability theory [13]. This will help motivate an initial term language for describing circuits based on a modification of the CD calculus [33].

A natural place to start our discussion of digital circuits is with combinational circuits. Such circuits consist only of basic gates interconnected acyclically with wires. We do not allow any form of feedback, and there are no sequential components such as clocks or buffers. For example, we can consider an elementary combinational circuit with only two gates, as in Figure 2.3, which adds together two bits.

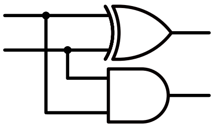


Figure 2.3: Half adder circuit.

This might be represented programmatically as a Verilog module.

```
module HalfAdder(a,b,s,c);
    input a,b; output s,c;
    xor(s,a,b); and(c,a,b);
endmodule
```

As we can see, the semantics of the description really just depends on how components are connected together. We will take a categorical approach, modelling circuits as morphisms in a category where the objects are the types of input and output wires. Informally, we have the corresponding relationships in our model:

- circuits ⇝ morphism
- wire types ⇝ objects
- connecting circuits ⇝ composing morphisms

Composing morphisms will then correspond to connecting circuits together. In the simplest case, we will treat all wires in the same way, so the objects of our category will be natural numbers corresponding to some number of wires in parallel. A circuit with m inputs and n outputs would be a morphism $c : m \to n$. An example with 2 inputs and 3 outputs is shown in Figure 2.4.

Figure 2.4: Circuits correspond to morphisms.

The parallel composition of circuits and wires then corresponds to a tensor product $\otimes$ in our category, as shown in Figure 2.5. Since we are assuming all wires are of the same type, the category has objects $\mathbb{N}$ and tensor product $+$ and is known as a PROP (Definition 2.4.1).

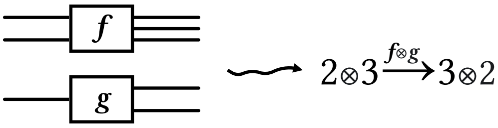


Figure 2.5: Parallel composition is tensor product.

Definition 2.4.1: PROP

A Product of Permutations (PROP) Catgeory is a strict symmetric monoidal category $(\mathbb{C}, \otimes, \mathcal{I})$ with a distinguished object X such that every other object Y can be expressed as $Y = X^{\otimes n}$ for some $n \in \mathbb{N}$.

Any circuit can thus be further broken down into a sequential and parallel composition of atomic elements, gates and operations on wires. Going back to the half adder circuit in Figure 2.3, we can decompose it as a copying of the input wires followed by a swap of two of the wires, and a composition with the two gates at the end (shown graphically in Figure 2.6). We can also write it in a slightly more verbose form as the composition of the constituent morphisms.

$$(\mathbf{xor} \otimes \mathbf{and}) \circ (\mathbf{id}_1 \otimes \mathbf{swap} \otimes \mathbf{id}_1) \circ (\mathbf{copy} \otimes \mathbf{copy})$$

Alternatively, it is slightly more readable to use a flipped version of composition, the sequencing operator ;, which composes morphisms from left to right, instead of $\circ$, which composes them from right to left.

$$(\mathbf{copy} \otimes \mathbf{copy})\,;(\mathbf{id}_1 \otimes \mathbf{swap} \otimes \mathbf{id}_1); (\mathbf{xor} \otimes \mathbf{and})$$

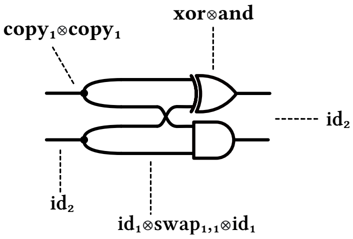


Figure 2.6: Half adder circuit expanded.

The existence of swap highlights the fact that our category should be symmetric monoidal, meaning there are isomorphisms **swap** : $A \otimes B \to B \otimes A$ which swaps two wires. The other operations on wires, *copy* and *delete* (Figure 2.7), are also special morphisms. These morphisms should satisfy some standard laws (Figure 2.8), which means they form a comonoid. In the PROP category, they are given by **copy** : $1 \to 2$ and **delete** : $1 \to 0$.

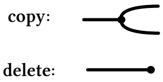


Figure 2.7: Copy and delete.

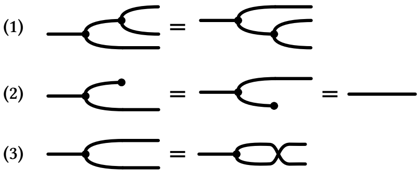


Figure 2.8: Comonoid laws.

While we can describe the connectivity of the wiring within a circuit using only copy and delete operations, working directly with individual wires becomes cumbersome for larger circuits. Therefore, it is often helpful to group together multiple physical wires, representing them by a single virtual wire carrying multiple bits. In a PROP these wires are simply labelled by some number of bits, but we may also want to distinguish wires carrying different types of data, so modelling circuits by a more general monoidal category can be helpful.

Extending copy delete operations to arbitrary types of wires, we end up with a copy-delete (CD) category [10]. A CD category (Definition 2.4.2) is a symmetric monoidal category for which every object has copy and delete operations satisfying the commutative comonoid structure, with an additional technical requirement that copying and deleting behave well with the tensor product $\otimes$.

Definition 2.4.2: CD Category

A symmetric monoidal category $(\mathbb{C}, \otimes, \mathcal{I})$, with morphisms $\mathbf{copy}_X$ and $\mathbf{delete}_X$ for each object X in $\mathbb{C}$ satisfying the laws in Figures 2.8 and 2.9.

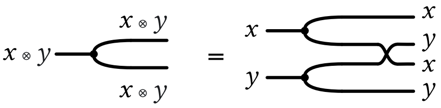


Figure 2.9: Compatibility conditions.

We are particularly interested in circuits constructed by assembling gates from some predetermined primitive set. We can thus extend our categorical framework by considering some set of basic gates $\mathcal{G}$. Each gate is associated with a type of input and output wires, and our category $\mathbb{C}$ should have a distinguished morphism for each gate in $\mathcal{G}$. For example, we might consider $\mathcal{G} = \{\mathbf{and}, \mathbf{or}, \mathbf{not}\}$. In the PROP model, these would be represented by morphisms $\mathbf{and} : 2 \to 1$, $\mathbf{or} : 2 \to 1$, $\mathbf{not} : 1 \to 1$.

As was discussed in Section 2.2, all disconnected circuits behave the same way, so we require that each gate is *discardable*: deleting its outputs is the same as deleting its inputs (Figure 2.10). Putting this all together, we see that circuits are described by morphisms in a Markov category (Definition 2.4.3).

Definition 2.4.3: Markov Category

A Markov category is a CD category $(\mathbb{C}, \otimes, \mathcal{I})$ where every morphism additionally satisfies the discardability condition in Figure 2.10.

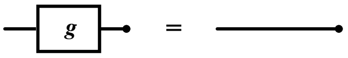


Figure 2.10: Discardablity condition.

2.5 Combinational Circuit Language

The first iteration of our programming language is a higher-order extension of the CD calculus introduced by Dario Stein [33] as an internal language of CD categories. The language consists of pairs, projections and abstractions. Unlike the original CD calculus, we will make a distinction between two classes of terms: *wire* terms and *circuit* terms, and each of these sets of terms has its own types. We also extend the language with lambda abstractions on circuits and replace primitive let binding with circuit abstraction. Circuits will take wires as inputs and produce wires as outputs. Each circuit is constructed by combining elementary circuits (gates) from a predefined set of primitives $\mathcal{G}$. The terms of the language are given in Definition 2.5.

Definition 2.5.1: Terms of λ_{comb}

$$
\begin{aligned}
W &:= \ a \mid () \mid (W, W) \mid \text{fst } W \mid \text{snd } W \mid C\ W \\
C &:= \ x \mid \nu a.W \mid \lambda x.C \mid C\ C \mid g & \qquad\qquad (g \in \mathcal{G})
\end{aligned}
$$

We will let a range over names (representing wires) and we will let x range over variables (representing circuits), and we will assume the set of names and variables are disjoint. There are two types of binders in our calculus which achieve different things:

- $\nu a.W$ is a circuit abstraction, representing a circuit.
- $\lambda x.C$ is a function abstraction, representing a transformation on circuits.

The abstraction $\nu x.w$ represents a circuit taking wire x as input and giving wire w as output. Note that names in our language are slightly different from names in the π-calculus or ν-calculus; in particular, there is no explicit mechanism for comparing or communicating names. However, names will allow us to refer to wires at specific points in our circuit. Their only property is their identity, and they can not be instantiated with a value in the same way as variables. So, for example, we might write a NAND circuit using an AND gate and a NOT gate.

$$\textbf{nand} = \nu a.\textbf{not}(\textbf{and}\ a)$$

Note that the name a represents the whole pair of input wires. Assuming the signature contained the appropriate gates, we could write the example circuit from Figure 2.3 as

$$\textbf{half-adder} = \nu a.(\textbf{xor}\ a, \textbf{and}\ a)$$

The function abstraction $\lambda x.C$ corresponds to a function taking a circuit c as input and returning a circuit $t[c/x]$ as output. Suppose we wanted to express a circuit such as $\nu a.(c\ a, c\ a)$ where c was some complicated expression. It would be nicer to write let $f\ = c$ in $(f\ x, f\ x)$, and so we introduce let expressions as syntactic sugar for function application.

$$\text{let } x = C_1 \text{ in } C_2 \quad := (\lambda x.C_2)\ C_1$$

We will also introduce let expressions as syntactic sugar for circuit application. Note that using names rather than variables can resolve any ambiguity in the overloading of let.

$$\text{let } a = W_1 \text{ in } W_2 \quad := (\nu a.W_1)\ W_2$$

We will take the approach that products associate to the right, so $(a, b, c) : A \otimes B \otimes C$ will be treated as shorthand for $(a, (b, c)) : A \otimes (B \otimes C)$. And pattern matching in circuit abstractions can be treated as syntactic sugar for nesting lets with projections.

$$\begin{aligned} \nu(a_1, ..., a_n).w \quad := \quad & \nu z.(\text{let } a_1 = \text{fst } z \text{ in} \\ & \quad \text{let } a_2 = \text{fst}(\text{snd } z) \text{ in} \\ & \quad \vdots \\ & \quad \text{let } a_n = (\text{snd}^n\ z) \text{ in } w \qquad\qquad (z \text{ fresh})) \end{aligned}$$

This is extended to let expressions in the obvious way.

$$\text{let } (a_1, ..., a_n) = w \text{ in } z \quad := (\nu(a_1, ..., a_n).z)w$$

This simplifies the expression of many useful circuits. For example, we can now concisely express a full adder (Figure 2.11) and a 1-bit comparator (Figure 2.12) using this syntax.

$$\begin{aligned} \textbf{full-adder} = \quad & \text{let } \textbf{half-adder} = \nu a.(\textbf{xor } a, \textbf{and } a) \\ & \text{in } \nu(a, b, c).\text{let}(s1, c1) = \textbf{half-adder}(a, b) \\ & \text{in let}(s2, c2) = \textbf{half-adder}(s1, c) \\ & \text{in } (s2, \textbf{or}(c1, c2)) \end{aligned}$$

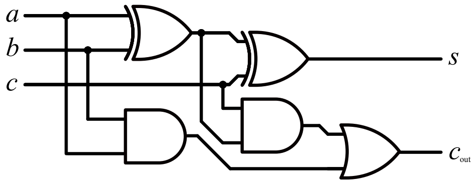


Figure 2.11: Full adder.

$$\begin{aligned} \textbf{comparator} = \quad & \nu(a, b).\ \text{let } (lt, gt) = (\textbf{and}(\textbf{not } a, b), \textbf{and}(a, \textbf{not } b)) \\ & \text{in } (lt, \textbf{not}(\textbf{xor}(lt, gt)), gt) \end{aligned}$$

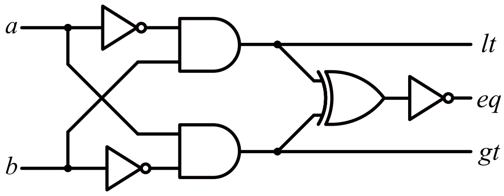


Figure 2.12: 1-bit comparator.

2.6 Types for λ_{comb}

Having introduced the syntax of λ_{comb}, we move on to discuss the types. The language's type system is broken into two parts: types for wires and types for circuits. We will assume a set $\mathcal{T}$ of atomic wire types.

$\underline{\text{Types}_W}$	$\underline{\text{Types}_C}$
$\sigma := \text{unit} \mid \alpha \mid \sigma_1 \times \sigma_2 \qquad (\alpha \in \mathcal{T})$	$\tau := \text{Circ}(\sigma_1, \sigma_2) \mid \tau_1 \to \tau_2$

Each class of circuits is then parameterised by a signature giving the primitive gates and atomic wire types.

Definition 2.6.1: Circuit Signature

A circuit signature $\langle \mathcal{T}, \mathcal{G}, \# \rangle$ is a tuple consisting of a set $\mathcal{T}$ of atomic types, a set $\mathcal{G}$ of atomic gates, and a function $\# : \mathcal{G} \to \text{Types}_C \times \text{Types}_C$ associating each gate with an input and an output type.

We split the typing context into two disjoint parts. Γ is a context assigning wire types to names and Δ is a context assigning circuit types to variables. There are also two different typing judgements: $\vdash_w$ assigning wire terms to wire types and $\vdash_c$ assigning circuit terms to circuit types.

$$\frac{}{\Gamma, a : \sigma, \Gamma'; \Delta \vdash_w a : \sigma}\ (\textsc{typ-name}) \qquad \frac{}{\Gamma; \Delta \vdash_w () : \text{unit}}\ (\textsc{typ-unit})$$

$$\frac{\Gamma; \Delta \vdash_w u : \sigma_1 \quad \Gamma; \Delta \vdash_w v : \sigma_2}{\Gamma; \Delta \vdash_w (u, v) : \sigma_1 \times \sigma_2}\ (\textsc{typ-prod}) \qquad \frac{\Gamma; \Delta \vdash_w s : \sigma_1 \quad \Gamma; \Delta \vdash_c c : \text{Circ}(\sigma_1, \sigma_2)}{\Gamma; \Delta \vdash_w c\, s : \sigma_2}\ (\textsc{typ-circ-app})$$

$$\frac{\Gamma; \Delta \vdash_w s : \sigma_1 \times \sigma_2}{\Gamma; \Delta \vdash_w \text{fst}\ s : \sigma_1}\ (\textsc{typ-proj-1}) \qquad \frac{\Gamma; \Delta \vdash_w s : \sigma_1 \times \sigma_2}{\Gamma; \Delta \vdash_w \text{snd}\ s : \sigma_2}\ (\textsc{typ-proj-2})$$

Figure 2.13: Wire typing rules.

$$\frac{}{\Gamma; \Delta, x : \tau, \Delta' \vdash_c x : \tau}\ (\textsc{typ-var}) \qquad \frac{g \in \mathcal{G} \quad \#g = \langle \sigma_1, \sigma_2 \rangle}{\Gamma; \Delta \vdash_c g : \text{Circ}(\sigma_1, \sigma_2)}\ (\textsc{typ-gate})$$

$$\frac{\Gamma, a : \sigma_1; \Delta \vdash_w u : \sigma_2}{\Gamma; \Delta \vdash_c \nu a.u : \text{Circ}(\sigma_1, \sigma_2)}\ (\textsc{typ-circ-abs}) \qquad \frac{\Gamma; \Delta, x : \tau_1 \vdash_c c : \tau_2}{\Gamma; \Delta \vdash_c \lambda x.c : \tau_1 \to \tau_2}\ (\textsc{typ-func-abs})$$

$$\frac{\Gamma; \Delta \vdash_c c_1 : \tau_1 \to \tau_2 \quad \Gamma; \Delta \vdash_c c_2 : \tau_1}{\Gamma; \Delta \vdash_c c_1\, c_2 : \tau_2}\ (\textsc{typ-func-app})$$

Figure 2.14: Circuit typing rules.

The typing rules satisfy various standard properties. These generally come in the form of 2 or 4 implications, as there are 2 choices of typing judgement ($\vdash_w$ and $\vdash_c$) and 2 choices of context (Γ and Δ). Because the typing rules TYP-CIRC-APP and TYP-CIRC-ABS introduce circuit typing judgements into wire typing derivations and vice versa, we will mostly prove each of these groups of results simultaneously by induction.

Theorem 2.6.1: Exchange

$$\Gamma, a : \sigma_1, b : \sigma_2, \Gamma'; \Delta \vdash_w u : \sigma_3 \implies \Gamma, b : \sigma_2, a : \sigma_1, \Gamma'; \Delta \vdash_w u : \sigma_3 \tag{2.1}$$

$$\Gamma, a : \sigma_1, b : \sigma_2, \Gamma'; \Delta \vdash_c c : \tau \implies \Gamma, b : \sigma_2, a : \sigma_1, \Gamma'; \Delta \vdash_c c : \tau \tag{2.2}$$

$$\Gamma; \Delta, x : \tau_1, y : \tau_2, \Delta' \vdash_w u : \sigma \implies \Gamma; \Delta, y : \tau_2, x : \tau_1, \Delta' \vdash_w u : \sigma \tag{2.3}$$

$$\Gamma; \Delta, x : \tau_1, y : \tau_2, \Delta' \vdash_c c : \tau_3 \implies \Gamma; \Delta, y : \tau_2, x : \tau_1, \Delta' \vdash_c c : \tau_3 \tag{2.4}$$

Proof: By induction on the typing derivations. The base cases TYP-NAME and TYP-VAR both validate the theorem and TYP-UNIT and TYP-GATE do not use the context at all. The rest of the cases are straightforward from the induction hypothesis.

Theorem 2.6.2: Weakening

$$a \notin \mathrm{dom}(\Gamma) \quad \text{and} \quad \Gamma; \Delta \vdash_w u : \sigma \implies \Gamma, a : \sigma'; \Delta \vdash_w u : \sigma \tag{2.5}$$

$$a \notin \mathrm{dom}(\Gamma) \quad \text{and} \quad \Gamma; \Delta \vdash_c c : \tau \implies \Gamma, a : \sigma; \Delta \vdash_c c : \tau \tag{2.6}$$

$$x \notin \mathrm{dom}(\Delta) \quad \text{and} \quad \Gamma; \Delta \vdash_w u : \sigma \implies \Gamma; \Delta, x : \tau \vdash_w u : \sigma \tag{2.7}$$

$$x \notin \mathrm{dom}(\Delta) \quad \text{and} \quad \Gamma; \Delta \vdash_c c : \tau \implies \Gamma; \Delta, x : \tau' \vdash_c c : \tau \tag{2.8}$$

Proof: By induction on the typing derivations using Theorem 2.6.1 to put the contexts in the appropriate forms for the TYP-CIRC-ABS and TYP-FUNC-ABS cases.

Theorem 2.6.3: Substitution

$$\Gamma, a : \sigma_1; \Delta \vdash_w u : \sigma_2 \quad \text{and} \quad \Gamma; \Delta \vdash_w s : \sigma_1 \implies \Gamma; \Delta \vdash_w u[s/a] : \sigma_2 \tag{2.9}$$

$$\Gamma, a : \sigma_1; \Delta \vdash_c c : \tau \quad \text{and} \quad \Gamma; \Delta \vdash_w s : \sigma_1 \implies \Gamma; \Delta \vdash_c c[s/a] : \tau \tag{2.10}$$

$$\Gamma; \Delta, x : \tau_1 \vdash_w u : \sigma \quad \text{and} \quad \Gamma; \Delta \vdash_c d : \tau_1 \implies \Gamma; \Delta \vdash_w u[d/x] : \sigma \tag{2.11}$$

$$\Gamma; \Delta, x : \tau_1 \vdash_c c : \tau_2 \quad \text{and} \quad \Gamma; \Delta \vdash_c d : \tau_1 \implies \Gamma; \Delta \vdash_c c[d/x] : \tau_2 \tag{2.12}$$

Proof Sketch: By induction on the typing derivation.

Proof on page 53

2.7 Normalisation and Equational Theory

Next, we move on to looking at the ways in which terms relate to each other. A good first step will be to define single-holed contexts. In fact, we must define four different varieties of context, as there is one for each combination of term and hole. The notation we propose uses the calligraphic letter to indicate the kind (wire or circuit) of term, and the subscript indicates the kind of the hole.

$$\mathcal{W}_w = [\cdot] \mid (\mathcal{W}_w, W) \mid (W, \mathcal{W}_w) \mid \mathsf{fst}\ \mathcal{W}_w \mid \mathsf{snd}\ \mathcal{W}_w \mid C\ \mathcal{W}_w \mid \mathcal{C}_w\ W \tag{2.13}$$

$$\mathcal{W}_c = [\cdot] \mid (\mathcal{W}_c, W) \mid (W, \mathcal{W}_c) \mid \mathsf{fst}\ \mathcal{W}_c \mid \mathsf{snd}\ \mathcal{W}_c \mid C\ \mathcal{W}_c \mid \mathcal{C}_c\ W \tag{2.14}$$

$$\mathcal{C}_w = [\cdot] \mid \nu a.\mathcal{W}_w \mid \lambda x.\mathcal{C}_w \mid \mathcal{C}_w\ C \mid C\ \mathcal{C}_w \tag{2.15}$$

$$\mathcal{C}_c = [\cdot] \mid \nu a.\mathcal{W}_c \mid \lambda x.\mathcal{C}_c \mid \mathcal{C}_c\ C \mid C\ \mathcal{C}_c \tag{2.16}$$

Using this definition, we can continue to define a reduction as a relation $\hookrightarrow$ on pairs of wire terms and on pairs of circuit terms by the rules

$$\frac{}{(\lambda x.c)d \hookrightarrow c[d/x]}\ (\beta) \qquad \frac{c_1 \hookrightarrow c_2}{\mathcal{W}_c[c_1] \hookrightarrow \mathcal{W}_c[c_2]}\ (\gamma_w) \qquad \frac{c_1 \hookrightarrow c_2}{\mathcal{C}_c[c_1] \hookrightarrow \mathcal{C}_c[c_2]}\ (\gamma_c)$$

A value is a wire or circuit for which no further $\hookrightarrow$ reductions are possible. We will write $u \Downarrow u'$ to indicate that there is a sequence of reductions for a wire term which result in a value u' and we will write $u \Downarrow$ if all reduction sequences end in a value. A key result is that $\hookrightarrow$ is strongly normalising for λ_{comb}. That is to say that every reduction sequence of a typeable term terminates.

Theorem 2.7.1: Strong Normalisation

$$\Gamma; \cdot \vdash_w u : \sigma \implies u \Downarrow \tag{2.17}$$

$$\Gamma; \cdot \vdash_c c : \tau \implies c \Downarrow \tag{2.18}$$

Comment: The proof is done using logical relations and differs from a standard normalisation proof in two key ways. Firstly, we have two different typing judgements and two different contexts. Therefore every induction takes place jointly over circuit judgements and wire judgements. Secondly, we do not require both contexts to be empty. Instead, we only require the circuit context to be empty, as this is the only context which can introduce function types.

Proof Sketch: We define a predicate SN which holds for terms which reduce to a value and functions which preserve membership of SN upon application. Then it remains to show that every typeable term satisfies this predicate. We strengthen the induction hypothesis to include nonempty Δ contexts and require that every substitution of values for circuit variables results in SN being satisfied.

Proof on page 54

We will now define a symmetric, reflexive, transitive relation $\equiv$ on pairs of wire terms or pairs of circuit terms which satisfies the structural properties. We define $\equiv$ as the least relation satisfying these properties and the axioms in definition 2.7.1.

$$\frac{w_1 \equiv w_2}{\mathcal{W}_w[w_1] \equiv \mathcal{W}_w[w_2]}\ (1) \qquad \frac{w_1 \equiv w_2}{\mathcal{C}_w[w_1] \equiv \mathcal{C}_w[w_2]}\ (2) \qquad \frac{c_1 \equiv c_2}{\mathcal{W}_c[c_1] \equiv \mathcal{W}_c[c_2]}\ (3) \qquad \frac{c_1 \equiv c_2}{\mathcal{C}_c[c_1] \equiv \mathcal{C}_c[c_2]}\ (4)$$

These axioms define an equational theory of our language.

Definition 2.7.1: Equational Theory of λ_{comb}

$$\text{fst } (s, t) \equiv s \tag{2.25}$$

$$\text{snd } (s, t) \equiv t \tag{2.26}$$

$$(\text{fst } s, \text{snd } s) \equiv s \tag{2.27}$$

$$(\nu a.t)V \equiv t[V/a] \tag{2.28}$$

$$(\nu a.t)s \equiv t[s!a] \tag{2.29}$$

$$(\nu a.c\ a) \equiv c \qquad (a \notin \text{fn}(c)) \tag{2.30}$$

$$(\lambda x.t)c \equiv t[c/x] \tag{2.31}$$

$$(\lambda x.f\ x) \equiv f \qquad (x \notin \text{fv}(f)) \tag{2.32}$$

The set V is defined by $V := () \mid a \mid V_1 \times V_2$. The notation $t[s!a]$ in equation 2.29 is affine substitution. We require that s is free at most once in t and not within a function application.

We will refer to a special first-order fragment of λ_{comb} called $\lambda_{comb}*$ which does not contain lambda terms. This fragment corresponds closely with the original CD calculus. We define the fragment as the set of terms with no circuit variables or lambda abstractions.

Definition 2.7.2: Terms of $\lambda_{comb}*$

$$W := \ a \mid () \mid (W, W) \mid \text{fst } W \mid \text{snd } W \mid C\ W$$

$$C := \ \nu a.W \mid g \qquad (g \in \mathcal{G})$$

A key property to note about λ_{comb} and $\lambda_{comb}*$ is that adding function types does not add any expressivity for basic circuit types. In particular, any (non function) term we write using functions in λ_{comb} could have equivalently been written without function types in $\lambda_{comb}*$.

Theorem 2.7.2: Normal Forms

For every closed circuit term with typing derivation $\Gamma; \cdot \vdash_c c : \mathrm{Circ}(\sigma_1, \sigma_2)$, there exists an equivalent term $c' \equiv c$ such that c' is in $\lambda_{comb}*$.

Proof Sketch: We know that every term which is closed with respect to circuit variables can be normalised to a value with no redexes. Thus it can be expressed as a term with no function types.

Proof on page 56

Therefore, λ_{comb} really acts like a hardware description language with an expressive macro system which aids in the conciseness with which we can describe hardware. This models a common feature in real-life hardware description languages, namely the separation between *synthesis-time* computations and *run-time* behaviour within the hardware itself.

2.8 Aside on Nominal Sets

We are almost ready to give a semantic model of our language, but before moving on, we will take a brief detour to look at nominal sets which will be used in our categorical model. Nominal sets offer a clean way of sidestepping issues that commonly occur when defining alpha equivalence classes for term languages with binders. The key idea is to define everything in terms of permutations of *atoms* and the actions of those permutations on terms. The presentation here follows various sources from the literature [14, 28]. We start by recalling some simple definitions, beginning with the definition of a group.

Definition 2.8.1: Group

A group G is a quad $(A, \epsilon, \cdot, (-)^{-1})$ consisting of a set A together with a multiplication operation $(\cdot) : A \times A \to A$, an inverse operation $(-)^{-1}$ and an identity element ϵ satisfying for $a, b, c \in A$.

- (associativity) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- (identity) $\epsilon \cdot a = a \cdot \epsilon = a$
- (inverse) $a \cdot a^{-1} = a^{-1} \cdot a = \epsilon$

Now we will give a few further useful definitions. An action of a group G on a set X describes how group elements act on the set elements.

Definition 2.8.2: Action of a Group on a Set

The action of a group $G = (A, \epsilon, \cdot, (-)^{-1})$ on a set X is a function $* : A \times X \to X$ satisfying

- $a * (b * x) = (a \cdot b) * x$
- $\epsilon * x = x$

From this, we can define a notion of a G-set, which is a set equipped with a group action.

Definition 2.8.3: G-sets

For a given group G, we define a G-set as a pair $(X, *)$ of a set X and an action of G on X.

We are particularly interested in permutation groups consisting of bijective functions between a set and itself. Composition is defined in the obvious way $\pi_1 \cdot \pi_2(a) = \pi_1(\pi_2(a))$, and the identity element is the identity permutation $\pi_{\text{id}}(a) = a$, which leaves each element unchanged. Each permutation π has an inverse π^{-1}.

Definition 2.8.4: Finitary Permutation Group

For a set of identifiers $\mathbb{A}$, the group Perm $\mathbb{A}$ of finitary permutations on $\mathbb{A}$ consisting of permutations π_1, π_2 which only affect a finite number of elements. That is the set $\{a \in A : \pi(a) \neq a\}$ is finite.

Putting this all together, we can define a Perm $\mathbb{A}$-set. Such a set is really a pair $(X, *)$ such that for any finitary permutations π_1 and π_2, $*$ has the properties required by definition 2.8.2.

- $\pi_1 * (\pi_2 * x) = (\pi_1 \cdot \pi_2) * x$
- $\pi_{id} * x = x$

It is helpful to consider a few examples. The most straightforward action is just to interpret $*$ as function application.

Example 2.8.1: Trivial Perm $\mathbb{A}$-set

$\mathbb{A}$ is trivially a Perm $\mathbb{A}$-set with $\pi * x = \pi(x)$.

A more interesting example is to look at compound objects containing atoms from $\mathbb{A}$.

Example 2.8.2: Strings of $\mathbb{A}$ are a Perm $\mathbb{A}$-set

Consider the set of finite strings $\mathbb{A}^*$. We can define $\pi * a_0 a_1 ... a_n = \pi(a_0)\pi(a_1)...\pi(a_n)$.

We move on to introduce the idea of a support. Given a Perm $\mathbb{A}$-set $(X, *)$, a subset $S \subseteq X$ is said to be a support of x if the action of every permutation which leaves all elements of a unchanged also leaves x unchanged.

Definition 2.8.5: Support of a Set

The set A is a support of x if for each $\pi \in \text{perm}\ \mathbb{A}$

$$(\forall a \in A\ .\ \pi(a) = a) \implies \pi * x = x$$

We will use the notation: $\text{supp}\ x$ to refer to the least support if it exists.

Now using the idea of support sets, we define a nominal set.

Definition 2.8.6: Nominal Set

A nominal set A is a Perm $\mathbb{A}$-set where each $x \in A$ has a finite support.

We model wires and circuits with free variables as structure-preserving functions between nominal sets. These functions are known as equivariant functions (and are morphisms in the category of nominal sets).

Definition 2.8.7: Equivariant Function

A function $f : X \to Y$ is equivariant if $f(\pi *_X x) = \pi *_Y f(x)$

If we have two nominal sets A and B, we construct a nominal set $A \Rightarrow B$ of functions between A and B.

Definition 2.8.8: Nominal Set of Functions $A \Rightarrow B$

A set of functions $\{f : A \to B\}$ can be endowed with the structure a nominal set by taking the action $*$ to be given by $(\pi * f)(x) := \pi *_B f(\pi^{-1} *_A x)$.

It is straightforward to define cartesian product on nominal sets.

Definition 2.8.9: Cartesian Product

Given nominal sets $\langle A, *_A\rangle$ and $\langle B, *_B\rangle$, their product $A \times B$ forms a nominal set with action $*_{A\times B}$ given in the obvious way as $\pi *_{A\times B} \langle a, b\rangle = \langle \pi *_A a, \pi *_B b\rangle$. The support of the set is given by the union of the supports of the two sets $\text{supp}\ A \cup \text{supp}\ B$.

2.9 Categorical Semantics

We are now ready for the semantics of λ_{comb}. Instead of working abstractly with Markov categories, we will give a concrete semantic model using a PROPs (as an instance of Markov categories) and nominal sets. Therefore, we will assume we are working over a signature with a single atomic wire type $\mathcal{T} = \{*\}$. We choose this model as it reflects circuits over wires of binary values, and gates do not, in general, commute through copying (as in Figure 2.1). We start with a category WIRE = $(\mathbb{N}, +, 0)$ which is a PROP (Definition 2.4.1). Wire types will correspond to objects in this category. We will assume a set $\mathbb{A}$ of wire names. Each object $X \in$ WIRE has distinguished morphisms $\mathbf{copy}_X : X \to X + X$ and $\mathbf{delete}_X : X \to 0$. And for each gate $g \in \mathcal{G}$ with $\#g = \langle A, B\rangle$, there is a distinguished morphism $[\![g]\!] : A \to B$. This is a locally small category, so it has Hom sets. Wire types will correspond to objects in WIRE:

- $[\![\text{unit}]\!] = 0$
- $[\![*]\!] = 1$
- $[\![\sigma_1 \times \sigma_2]\!] = [\![\sigma_1]\!] + [\![\sigma_2]\!]$

Next, we interpret circuit types as nominal sets. So each wire type is a Perm $\mathbb{A}$-set with every element having finite support. The notation $\mathbb{A}^{\#n}$ indicates the set of tuples of n distinct atoms. In each case, we assume that we are quotienting by the addition of discarded atoms[1].

- $[\![\text{Circ}(\sigma_1, \sigma_2)]\!] = \langle \bigcup_{n\in\mathbb{N}} (\text{WIRE}([\![\sigma_1]\!] + n, [\![\sigma_2]\!]) \times \mathbb{A}^{\#n}), *\rangle$

  where we define the action as $\pi * \langle W, a_1 ... a_n\rangle := \langle W, \pi(a_1)...\pi(a_n)\rangle$. We will assume the action is implicit from now on when giving elements of the nominal set.

- $[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$

  where the function space is given the structure of a nominal set by definition 2.8.8.

Now we define the semantic interpretation of the typing contexts by simply taking the product. Note that the monoidal product is a sum in the PROP category WIRE and the cartesian product for nominal sets, representing circuit types, is as given in definition 2.8.9.

- $[\![a_1 : \sigma_1, ..., a_n : \sigma_n]\!] = \sum_{i=1}^{n} [\![\sigma_i]\!]$
- $[\![x_1 : \tau_1, ..., x_n : \tau_n]\!] = [\![\tau_1]\!] \times ... \times [\![\tau_n]\!]$

Finally, we interpret typing judgements as equivariant functions (definition 2.8.7) between nominal sets.

- $[\![\Gamma; \Delta \vdash_w u : \sigma]\!] : [\![\Delta]\!] \to [\![\text{Circ}(\Gamma, \sigma)]\!]$

---

[1] Formally we define an equivalence relation $\sim$ between $\langle m : (a+n) \to b, \mathbb{A}^n\rangle$ and $\langle m \otimes \mathbf{delete}_k : (a+n+k) \to b, \mathbb{A}^{n+k}\rangle$ and we take the codomain of the functions $[\![\text{Circ}(\sigma_1, \sigma_2)]\!]$ to be the set of equivalence classes modulo $\sim$.

- $\llbracket \Gamma;\Delta \vdash_c c : \tau \rrbracket : \mathbb{A}^{\#\llbracket\Gamma\rrbracket} \times \llbracket\Delta\rrbracket \to \llbracket\tau\rrbracket$

Concretely, we give the semantics inductively on the structure of the typing derivation, starting with the wire typing judgements. We also define a function which breaks a typing context containing (possibly compound) wire types into a sequence of fresh individual wire atoms.

$$\mathrm{atoms}(a_1 : \sigma_1, ..., a_n : \sigma_n) := a_{1,1}, a_{1,2}, ..., a_{1,\llbracket\sigma_1\rrbracket}, ..., a_{n,1}, a_{n,2}, ..., a_{n,\llbracket\sigma_n\rrbracket}$$

- $\llbracket \Gamma, a : \sigma, \Gamma'; \Delta \vdash_w a : \sigma \rrbracket(d) = \langle \mathbf{delete}_{\llbracket\Gamma\rrbracket} \otimes \mathbf{id}_{\llbracket\sigma\rrbracket} \otimes \mathbf{delete}_{\llbracket\Gamma'\rrbracket}, \langle\rangle\rangle$
- $\llbracket \Gamma;\Delta \vdash_w c^{\mathrm{Circ}(\sigma_1,\sigma_2)}\ t_1^{\sigma} : \sigma_2 \rrbracket(d) = \langle m_2 \circ (m_1 \otimes \mathbf{id}_{\llbracket\Gamma\rrbracket} \otimes \mathbf{id}_{|s_2|})$

  $\circ\, (\mathbf{id}_{\llbracket\Gamma\rrbracket} \otimes \mathbf{swap}_{\llbracket\Gamma\rrbracket,|s_1|} \otimes \mathbf{id}_{|s_2|})$

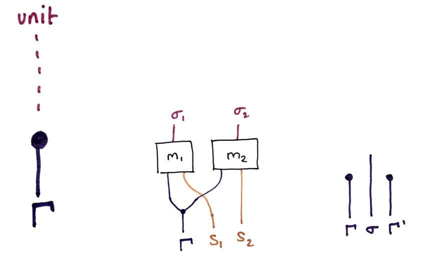
  $\circ\, (\mathbf{copy}_{\llbracket\Gamma\rrbracket} \otimes \mathbf{id}_{|s_1|} \otimes \mathbf{id}_{|s_2|}), s_1 +\!\!+\ s_2\rangle$

  where $\langle m_1, s_1\rangle = \llbracket \Gamma;\Delta \vdash_w t : \sigma \rrbracket(d)$

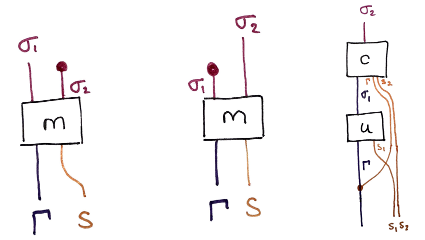
  and $\langle m_2, \mathrm{atoms}(\Gamma), s_2\rangle = \llbracket \Gamma;\Delta \vdash_c c \rrbracket(\mathrm{atoms}(\Gamma), d)$
- $\llbracket \Gamma;\Delta \vdash_w () : \mathrm{unit} \rrbracket(d) = \langle \mathbf{delete}_{\llbracket\Gamma\rrbracket}, \langle\rangle\rangle$
- $\llbracket \Gamma;\Delta \vdash_w \mathrm{fst}\ s^{\sigma_1\times\sigma_2} : \sigma_1 \rrbracket(d) = \langle (\mathbf{id}_{\llbracket\sigma_1\rrbracket} \otimes \mathbf{delete}_{\llbracket\sigma_2\rrbracket}) \circ m, s\rangle$
  where $\langle m, s\rangle = \llbracket \Gamma;\Delta \vdash_w s : \sigma_1 \times \sigma_2 \rrbracket(d)$
- $\llbracket \Gamma;\Delta \vdash_w \mathrm{snd}\ s^{\sigma_1\times\sigma_2} : \sigma_2 \rrbracket(d) = \langle (\mathbf{delete}_{\llbracket\sigma_1\rrbracket} \otimes \mathbf{id}_{\llbracket\sigma_2\rrbracket}) \circ m, s\rangle$
  where $\langle m, s\rangle = \llbracket \Gamma;\Delta \vdash_w (u, v) : \sigma_1 \times \sigma_2 \rrbracket(d)$
- $\llbracket \Gamma;\Delta \vdash_w s : \sigma_1 \times \sigma_2 \rrbracket(d) =$
  $\langle (m_1 \otimes m_2) \circ (\mathbf{id}_{\llbracket\sigma_1\rrbracket} \otimes \mathbf{swap}_{\llbracket\Gamma\rrbracket,|s_1|} \otimes \mathbf{id}_{|s_2|}) \circ (\mathbf{copy}_{\llbracket\Gamma\rrbracket} \otimes \mathbf{id}_{|s_1|} \otimes \mathbf{id}_{|s_2|}), s_1 +\!\!+\ s_2\rangle$
  where $\langle m_1, s_1\rangle = \llbracket \Gamma;\Delta \vdash_w t : \sigma \rrbracket(d)$ and $\langle m_2, s_2\rangle = \llbracket \Gamma;\Delta \vdash_c c \rrbracket(d)$

Circuit typing judgements are given by

1. $\llbracket \Gamma; x_1 : \tau_1, ..., x_i : \tau_i, ..., x_n : \tau_n \vdash_c x_i : \tau_i \rrbracket(s, d) = \pi_i\ d$
2. $\llbracket \Gamma;\Delta \vdash_c g : \mathrm{Circ}(\sigma_1, \sigma_2) \rrbracket(s, d) = \langle \llbracket g \rrbracket, \langle\rangle\rangle$
3. $\llbracket \Gamma;\Delta \vdash_c \nu a^{\sigma_1}.u^{\sigma_2} : \mathrm{Circ}(\sigma_1, \sigma_2) \rrbracket(s, d) = \langle m \circ (\mathbf{swap}_{\llbracket\sigma_1\rrbracket,\llbracket\Gamma\rrbracket} \otimes \mathbf{id}_{|s'|}), s +\!\!+\ s'\rangle$
   where $\langle m, s'\rangle = \llbracket \Gamma, a : \sigma_1;\Delta \vdash_w u : \sigma_2 \rrbracket(d)$
4. $\llbracket \Gamma;\Delta \vdash_c \lambda x^{\tau_1}.c^{\tau_2} : \tau_1 \to \tau_2 \rrbracket(s, d) = x' \mapsto \llbracket \Gamma;\Delta, x : \tau_1 \vdash_c c : \tau_2 \rrbracket(s, d, x')$
5. $\llbracket \Gamma;\Delta \vdash_c f^{\tau_1\to\tau_2}\ c^{\tau_1} : \tau_2 \rrbracket(s, d) = h(z)$
   where $h = \llbracket \Gamma;\Delta \vdash_c f : \tau_1 \to \tau_2 \rrbracket(s, d)$ and $z = \llbracket \Gamma;\Delta \vdash_c c : \tau_1 \rrbracket(s, d)$



(a) TYP-UNIT (b) TYP-PROD (c) TYP-NAME



(d) TYP-PROJ1 (e) TYP-PROJ2 (f) TYP-CIRC-APP

Figure 2.15: String diagram semantics for wires.

The categorical notation involves a lot of plumbing of morphisms which somewhat obscures the key ideas. It can also be helpful to think of the semantics diagrammatically as in Figures 2.15 and 2.16.

Theorem 2.9.1: Correctness of Nominal Sets and Equivariant Functions

For each wire judgement $\Gamma; \Delta \vdash_w u$, $[\![\Gamma; \Delta \vdash_w u]\!]$ is an equivariant function and for each judgement $\Gamma; \Delta \vdash_c c$, $[\![\Gamma; \Delta \vdash_c c]\!]$ is an equivarient function. And for each each type τ, the interpretation $[\![\tau]\!]$ is a nominal set.

Proof on page 56

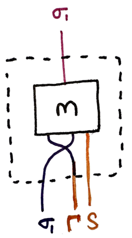


Figure 2.16: String diagram semantics for circuit abstraction.

We finish by giving a soundness result for the equational theory. In particular, we show that if any two terms are equal using the equaltional rules in definition 2.7.1, then it must be the case that they have the same semantic interpretation. This opens the door for equational reasoning about circuits described in our term language and gives us guarantees about our implementation.

Theorem 2.9.2: Soundness

The semantic interpretation respects the equational theory.

$$\Gamma; \Delta \vdash_w u_1 \equiv u_2 \implies [\![\Gamma; \Delta \vdash_w u_1 : \sigma]\!] = [\![\Gamma; \Delta \vdash_w u_2 : \sigma]\!]$$

$$\Gamma; \Delta \vdash_c c_1 \equiv c_2 \implies [\![\Gamma; \Delta \vdash_c c_1 : \tau]\!] = [\![\Gamma; \Delta \vdash_c c_2 : \tau]\!]$$

Proof on page 56

Chapter 3

Synchronous Circuits

We are now ready to move from combinational to synchronous circuits. These circuits consist of a global clock and components which make synchronised changes to their state based on the clock signal. We will start by discussing the later modality (Section 3.2) and will take a look at streaming computations more broadly. Then (in Section 3.3) we introduce a modification of λ_{comb} with a new constructor to enable synchronous behaviour.

3.1 Synchronous and Streaming Programming Languages

The use of functional programming techniques to describe sequential streams has been explored at length. The idea is known as Functional Reactive Programming (FRP) and synchronous versions have been implemented in languages such as Estrel [6], Lustre [9], and Lucid Synchrone [29]. Although there are very strong links between synchronous circuits and streaming reactive programs, the emphasis has mainly been on embedded software rather than hardware applications.

One of the significant contributions of this direction of research has been the use of modal types to guard feedback and ensure productive definitions. We will consider a type system with a type constructor $\bullet$ known as the later modality. The later modality was originally introduced by Nakano [26] to enforce productive function definitions but will correspond to delay-guarded feedback at the circuit level.

3.2 What is the Later Modality?

The later modality is all about guardedness. A term has a guarded type if it occurs in a place that ensures some *progress* has been made before the term is used. If all recursive occurrences of a term are guarded, our recursive definitions are guaranteed to be productive. We will use the notation $\bullet\sigma$ to express types guarded by the later modality.

Terms of type $\bullet\sigma$ are more general than those of type σ. If we think of $\bullet\sigma$ as a term available later and σ as a term available now, it makes sense that we could use the value available now at a later point in time, but not vice versa. We formalise this intuition by defining a subtyping relation specifying which types are more general than others. We define this relation ($\preceq \subseteq \textbf{Typ} \times \textbf{Typ}$) to be the least set satisfying the rules in Figure 3.1. Now each term is actually associated with a whole family of different valid types.

$$\frac{}{\sigma \preceq \sigma}\ (\preceq\text{-}\textsc{Refl}) \qquad \frac{\sigma \preceq \tau \quad \tau \preceq \gamma}{\sigma \preceq \gamma}\ (\preceq\text{-}\textsc{Trans}) \qquad \frac{}{\sigma \preceq \bullet\sigma}\ (\preceq\text{-}\textsc{Later})$$

$$\frac{\sigma_1 \preceq \sigma_2 \quad \tau_1 \preceq \tau_2}{(\sigma_2 \to \tau_1) \preceq (\sigma_1 \to \tau_2)}\ (\preceq\text{-}\textsc{Func}) \qquad \frac{\sigma \preceq \tau}{\bullet\sigma \preceq \bullet\tau}\ (\preceq\text{-}\textsc{Delay}) \qquad \frac{\sigma_1 \preceq \sigma_1' \quad \sigma_2 \preceq \sigma_2'}{\sigma_1 \times \sigma_2 \preceq \sigma_1' \times \sigma_2'}\ (\preceq\text{-}\textsc{Prod})$$

$$\frac{}{(\bullet\sigma \to \bullet\tau) \preceq \bullet(\sigma \to \tau)}\ (\preceq\text{-}\textsc{Dist1}) \qquad \frac{}{\bullet(\sigma \to \tau) \preceq (\bullet\sigma \to \bullet\tau)}\ (\preceq\text{-}\textsc{Dist2})$$

$$\frac{}{\bullet(\sigma_1 \times \sigma_2) \preceq \bullet\sigma_1 \times \bullet\sigma_2}\ (\preceq\text{-}\textsc{ProdDist1}) \qquad \frac{}{\bullet\sigma_1 \times \bullet\sigma_2 \preceq \bullet(\sigma_1 \times \sigma_2)}\ (\preceq\text{-}\textsc{ProdDist2})$$

Figure 3.1: Subtyping rules.

3.2.1 Streams

We will take a short pause from circuits and will think about synchronous programming more generally. Synchronous programming revolves around a global clock which induces each term to correspond to a co-inductive stream of values. We will give some informal examples to build intuition for the more formal presentation of the later modality in the next section. For simplicity, we will work with the set $\mathbb{N}^\omega$ of infinite streams of natural numbers for now.

We start with functions which operate pointwise on streams. For example, given two streams $a, b \in \mathbb{N}^\omega$, we lift addition to the level of streams by considering $(a +_\omega b)_i := a_i + b_i$. Next, we introduce a special construct $\rhd$ (pronounced followed-by). This will append a value to the front of a stream, shifting the whole stream down by one step.

Definition 3.2.1: Followed By Operation

We define the binary operation $\rhd : \mathbb{N} \times \mathbb{N}^\omega \to \mathbb{N}^\omega$ by $(n \rhd a)_i = \begin{cases} n, & i = 0 \\ a_{i-1}, & i > 0 \end{cases}$

Consider a very simple example with two constant streams.

Example 3.2.1: Combination of Constant Streams

$0 \rhd 1^\omega = 0, 1, 1, 1, \ldots$

Now we might use this notation to give a simple recursive definition.

Example 3.2.2: Counting Stream Definition

$x = 0 \rhd (x +_\omega 1)$

This definition is well-founded because it has a unique solution $x = 0, 1, 2, 3, \ldots$. Similarly we can even define the Fibonacci sequence.

Example 3.2.3: Fibonacci Sequence

$\mathrm{fib} = 0 \rhd (\mathrm{fib} +_\omega (1 \rhd \mathrm{fib}))$

Which, if we expand using the definition of the $\rhd$ operation, gives us the expected result $\mathrm{fib} = 0, 1, 1, 2, 3, 5, \ldots$. In both these cases, the recursive call was *guarded* by appearing on the right of the $\rhd$ construct. If we instead considered a stream defined by an unguarded recursive call, we end up with an invalid definition.

Example 3.2.4: Bad Stream

$x = x +_\omega (0 \rhd 1^\omega)$

To make things easier to reason about, we will replace recursion with a fixed point operator. For instance our Fibonacci sequence from before would be written as follows.

Example 3.2.5: Fibonacci Sequence Using Fixedpoint

$\mathrm{fib} = \mathrm{fix}\ \lambda x.(0 \rhd (x +_\omega (1 \rhd x)))$

The key to only allowing well-founded recursion is to replace the fixed point function

$$\mathrm{fix} : (\mathbb{N}^\omega \to \mathbb{N}^\omega) \to \mathbb{N}^\omega$$

with a guarded fixed point function

$$\mathrm{fix}_\bullet : (\bullet\mathbb{N}^\omega \to \mathbb{N}^\omega) \to \mathbb{N}^\omega$$

This fixed point operator requires that the input function is able to “remove” the later modality from its input type. In order to allow such definitions, we must change the type of the $\triangleright$ operation, which is how we make progress in a recursive definition, to allow the removal of the later modality from its second argument

$$\triangleright : \mathbb{N} \times \bullet\mathbb{N}^{\omega} \to \mathbb{N}^{\omega}$$

3.2.2 Synthetic Guarded Domain Theory

The later modality offers insight into well-founded recursion for streams. In the circuit case, instead of avoiding ill-founded recursion, we would like to avoid purely combinational cycles. As such, we want to ensure that all feedback loops are guarded by a delay element such as a buffer or register. Unfortunately, the informal presentation in terms of sequences is not sufficient for the general case. Instead, previous work has primarily focused on giving analytic semantics to guarded reactive programming by interpreting streams as ultrametric spaces [18]. This is rather indirect and difficult to work with. We will use the topos of trees [7], which offers a somewhat more direct interpretation.

Unlike the previous categorical model, the topos of trees is behavioural. Multiple circuits implement the same behaviour on streams, so this model validates equalities that do not generally hold when taking a structural approach to modelling hardware. However, the topos of trees does form a Markov category, and so the ideas from Section 2.4 still apply.

3.2.3 Background on the Topos of Trees

Each type α will be associated with a family of sets A_i indexed by $i \in \mathbb{N}^+$ and a family of restriction functions $r_i : A_{i+1} \to A_i$. We will think of A_i as being the set of execution traces after i clock cycles and the restriction functions r as extracting the immediate prefixes of the traces.

$$\begin{tikzcd}A_1 & A_2 \arrow[l, "r_1"'] & A_3 \arrow[l, "r_2"'] & \ldots \arrow[l, "r_3"']\end{tikzcd}$$

To take a concrete example, the type of streams of integers $\mathbb{Z}$ can be represented by taking the set A_n to be the n-tuple approximations of an infinite stream of integers.

$$\begin{tikzcd}\mathbb{Z} & \mathbb{Z}^2 \arrow[l, "z_1"'] & \mathbb{Z}^3 \arrow[l, "z_2"'] & \ldots \arrow[l, "z_3"']\end{tikzcd}$$

Where we define the restriction functions in the obvious way.

$$z_i(\langle x_1, ..., x_{i-1}, x_i\rangle) = \langle x_1, ..., x_{i-1}\rangle$$

The later modality transforms a type σ to a type $\bullet\sigma$ of streams delayed by one timestep. The initial set is replaced with the singleton set, so the first restriction function is uniquely determined.

$$\begin{tikzcd}
\alpha : & A_1 & A_2 \arrow[l, "r_1"'] & A_3 \arrow[l, "r_2"'] & ... \arrow[l, "r_3"'] \\
\bullet\alpha : & \{\star\} & A_1 \arrow[l, "r_\star"'] & A_2 \arrow[l, "r_1"'] & ... \arrow[l, "r_2"']
\end{tikzcd}$$

There is also a special void type 0 given by the empty sets and trivial restrictions.

$$\begin{tikzcd}
0 : & \{\} & \{\} \arrow[l, "t"'] & \{\} \arrow[l, "t"'] & ... \arrow[l, "t"']
\end{tikzcd}$$

$$\begin{tikzcd}
\bullet 0 : & \{\star\} & \{\} \arrow[l, "t"'] & \{\} \arrow[l, "t"'] & ... \arrow[l, "t"']
\end{tikzcd}$$

We now consider the space of functions between streams. Importantly, we only allow causal functions for which the output at a given timestep never depends on the input at future timesteps. Therefore, functions between streams will be given as a sequence of functions between the partial results, and we will require that taking functions on the partial results commutes with taking restrictions according to the following diagram.

$$\begin{tikzcd}
A_1 \arrow[d, "f_1"] & A_2 \arrow[l, "r_1"'] \arrow[d, "f_2"] & A_3 \arrow[l, "r_2"'] \arrow[d, "f_3"] & \arrow[l, "r_3"'] \\
B_1 & B_2 \arrow[l, "r'_1"'] & B_3 \arrow[l, "r'_2"'] & \arrow[l, "r'_3"']
\end{tikzcd}$$

3.2.4 Category Theoretic Perspective

It is possible to consider this from a categorical perspective. We will use the ordinal ω to signify the order category of the positive natural numbers.

$$\begin{tikzcd}
1 \arrow[r, "\leq"] & 2 \arrow[r, "\leq"] & 3 \arrow[r, "\leq"] & ...
\end{tikzcd}$$

The central idea of synthetic guarded domain theory is to consider the topos $\mathcal{S}$ of presheaves on ω. Concretely, the objects of $\mathcal{S}$ are functors $X : \omega^{\mathrm{op}} \to \mathbf{Set}$ (where ω^{op} denotes the category ω with all morphisms reversed). The morphisms of $\mathcal{S}$ are exactly the families of restriction functions between sets. Then the later modality $\bullet(-)$ is really an endofunctor in the category $\mathcal{S}$.

The category $\mathcal{S}$ is a Markov category with the cartesian product $\times$ as the monoidal product. We have operations $\mathbf{copy}_X : X \to X \otimes X$ given by $\mathbf{copy}_{X\,i}(X(i)) = X(i) \times X(i)$ and $\mathbf{delete}_X : X \to 1$ given $\mathbf{delete}_{X\,i}(X(i)) = \{*\}$.

The topos of trees category is cartesian closed, meaning we can create exponential objects B^A representing morphisms from $A \to B$. If we return to the picture from above, we see that the morphism is described by a sequence of components.

$$\begin{tikzcd}
A_1 \arrow[d, "f_1"] & A_2 \arrow[l, "r_1"'] \arrow[d, "f_2"] & A_3 \arrow[l, "r_2"'] \arrow[d, "f_3"] & {} \arrow[l, "r_3"'] \\
B_1 & B_2 \arrow[l, "r'_1"'] & B_3 \arrow[l, "r'_2"'] & {} \arrow[l, "r'_3"']
\end{tikzcd}$$

The exponential object is then given by a sequence of partial streams of functions.

$$B^A(k) = \mathrm{Hom}_{\mathcal{S}}(\bullet^k(0) \times A, B)$$

Here we use the iterated application of the endofunctor $\bullet(-)$ to create a partial chain

$$1 \leftarrow 1 \leftarrow \ldots \leftarrow 1 \leftarrow 0 \leftarrow 0 \leftarrow \ldots$$

such that taking the product with it results in a partial stream of functions of length k. We then define a special family of morphisms $\mathbf{eval}_{A,B} : B^A \times A \to B$ which apply the exponential object. This is given by $\mathbf{eval}_{A,B\,i}(f, A(i)) = f(A(i))$. There is also a natural isomorphism **curry** between $\mathrm{Hom}(A \times B, C)$ and $\mathrm{Hom}(A, C^B)$.

A key result is that there are a family of fixed point morphisms $\mathbf{fix}_A : A^{\bullet A} \to A$ which compute the fixed points of guarded functions [7].

3.3 Later in Time

We use the later modality to introduce synchronous circuits. We first introduce the *followed-by* construct to λ_{comb} (Definition 2.5.1) together with a guarded fixed point operator to give a new term language

λ_{sync}.

Definition 3.3.1: Terms of λ_{sync}

$$W := \; a \mid () \mid (W, W) \mid \mathsf{fst}\ W \mid \mathsf{snd}\ W \mid C\ W \mid W \rhd W$$

$$C := \; x \mid \nu a.W \mid \lambda x.C \mid C\ C \mid g\ C \mid \mathsf{fix}_\sigma\ C \qquad\qquad (g \in \mathcal{G})$$

At the hardware level, we think of $\rhd$ as a buffer which is initialised to a particular value. We will represent this diagrammatically using a box with the initial value entering as a wire from the top as shown in Figure 3.2.

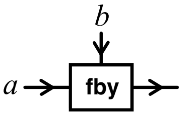


Figure 3.2: The term $a \rhd b$ represented diagrammatically.

We also need to introduce suitable typing rules to extend the type system appropriately.

$$\frac{\Gamma; \Delta \vdash_w u : \sigma \quad \Gamma; \Delta \vdash_w v : \bullet\sigma}{\Gamma; \Delta \vdash_w u \rhd v : \sigma}\ (\textsc{typ-fby}) \qquad \frac{\Gamma; \Delta \vdash_c c : \mathsf{Circ}(\bullet\sigma, \sigma)}{\Gamma; \Delta \vdash_w \mathsf{fix}_\sigma\ c : \sigma}\ (\textsc{typ-fix})$$

$$\frac{\Gamma; \Delta \vdash_w u : \sigma_1 \quad \sigma_2 \preceq \sigma_1}{\Gamma; \Delta \vdash_w u : \sigma_2}\ (\textsc{typ-subtyp})$$

Figure 3.3: New typing rules.

As an example using the new constructs, we can construct a circuit which produces the alternating stream $0, 1, 0, 1, \ldots$. Assuming there is a way to generate the constant 0[1], we take the fixed point of a circuit which delays and inverts its input: $\mathsf{fix}_*\ \nu a.(0 \rhd \textbf{not}\ a)$. The corresponding circuit diagram is given in Figure 3.4.

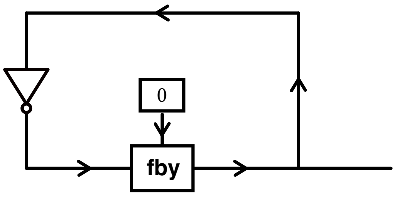


Figure 3.4: A simple toggle circuit.

We can consider some more interesting examples of combinational circuits such as a Fibonacci sequence generator based on the example 3.2.3 which is given by $\mathsf{fix}_{*^4}\ (\nu a.0 \rhd \mathsf{add}(a, 1 \rhd a))$. A full example of

---

[1] which we will implement by introducting a function type $\mathsf{false} : \mathsf{Circ}(\mathsf{unit}, *)$ and then writing $\mathsf{false}\ ()$ for 0.

this circuit is given as a case study (Section 5.2.5). The corresponding diagram for this circuit is shown in Figure 3.5.

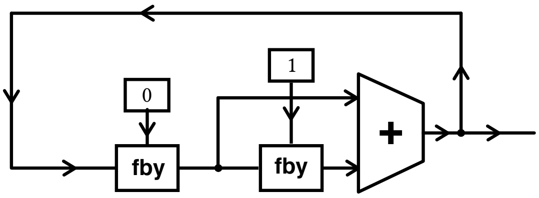


Figure 3.5: Fibbonaci sequence circuit.

3.4 Guarded Semantics

We will associate the base type $*$ with a bitstream where $\mathbb{B} = \{0,1\}$. Each type corresponds to an object in $\mathcal{S}$. Note that morphism in a functor category like $\mathcal{S}$ are just natural transformations, so have a component (which is a function, as it is a morphism in **Set**) at each timestep $n \in \mathbb{N}^+$.

- $\llbracket * \rrbracket = \ \mathbb{B} \xleftarrow{b_1} \mathbb{B}^2 \xleftarrow{b_2} \mathbb{B}^3 \xleftarrow{b_3} ...$
- $\llbracket \text{unit} \rrbracket = \bullet^{\infty}(0) = \ \{*\} \xleftarrow{!} \{*\} \xleftarrow{!} \{*\} \xleftarrow{!} ...$
- $\llbracket \sigma_1 \times \sigma_2 \rrbracket = \llbracket \sigma_1 \rrbracket \times \llbracket \sigma_2 \rrbracket$
- $\llbracket \text{Circ}(\sigma_1, \sigma_2) \rrbracket = \llbracket \sigma_1 \rrbracket \Rightarrow \llbracket \sigma_2 \rrbracket$
- $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \Rightarrow \llbracket \tau_2 \rrbracket$
- $\llbracket \bullet \sigma \rrbracket = \bullet(\llbracket \sigma \rrbracket)$

We extend the semantic interpretation to typing contexts.

- $\llbracket a_1 : \sigma_1, ..., a_n : \sigma_n \rrbracket = \llbracket \sigma_1 \rrbracket \times ... \times \llbracket \sigma_n \rrbracket$
- $\llbracket x_1 : \tau_1, ..., x_n : \tau_n \rrbracket = \llbracket \tau_1 \rrbracket \times ... \times \llbracket \tau_n \rrbracket$

Typing judgements then correspond to morphisms in $\mathcal{S}$ taking the typing contexts to the assigned type.

- $\llbracket \Gamma, a : \sigma, \Gamma'; \Delta \vdash_w a : \sigma \rrbracket = \textbf{delete}_{\llbracket \Gamma \rrbracket} \otimes \textbf{id}_{\llbracket \sigma \rrbracket} \otimes \textbf{delete}_{\llbracket \Gamma' \rrbracket} \otimes \textbf{delete}_{\llbracket \Delta \rrbracket}$
- $\llbracket \Gamma; \Delta \vdash_w () : \text{unit} \rrbracket : \textbf{delete}_{\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket}$
- $\llbracket \Gamma; \Delta \vdash_w (u, v) : \sigma_1 \times \sigma_2 \rrbracket = \textbf{copy}_{\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket}; (\llbracket \Gamma; \Delta \vdash_w u : \sigma_1 \rrbracket \times \llbracket \Gamma; \Delta \vdash_w v : \sigma_2 \rrbracket)$
- $\llbracket \Gamma; \Delta \vdash_w \text{fst } u^{\sigma_1 \times \sigma_2} : \sigma_1 \rrbracket = \llbracket \Gamma; \Delta \vdash_w u : \sigma \rrbracket; \pi_1$
- $\llbracket \Gamma; \Delta \vdash_w \text{snd } u^{\sigma_1 \times \sigma_2} : \sigma_2 \rrbracket = \llbracket \Gamma; \Delta \vdash_w u : \sigma \rrbracket; \pi_2$
- $\llbracket \Gamma; \Delta \vdash_w u \rhd v : \sigma \rrbracket(i) = \llbracket \Gamma; \Delta \vdash_w u : \sigma \rrbracket(1) \times \llbracket \Gamma; \Delta \vdash_w \bullet v : \sigma \rrbracket(i)$[2]
- $\llbracket \Gamma; \Delta \vdash_w \text{fix } c : \sigma \rrbracket = \textbf{fix}_{\llbracket \sigma \rrbracket} \circ \llbracket \Gamma; \Delta \vdash_c c : \text{Circ}(\bullet \sigma, \sigma) : \sigma \rrbracket$
- $\llbracket \Gamma; \Delta \vdash_w c\, u : \sigma_2 \rrbracket = \textbf{copy}_{\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket}; \llbracket \Gamma; \Delta \vdash_c c : \text{Circ}(\sigma_1, \sigma_2) \rrbracket) \times \llbracket \Gamma; \Delta \vdash_w u : \sigma_1 \rrbracket; \textbf{eval}_{\llbracket \sigma_1 \rrbracket, \llbracket \sigma_2 \rrbracket}$
- $\llbracket \Gamma; \Delta \vdash_c g : \text{Circ}(\sigma_1, \sigma_2) \rrbracket = \llbracket g \rrbracket$
- $\llbracket \Gamma; \Delta \vdash_c f\ c : \tau_2 \rrbracket = \textbf{copy}_{\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket}; \llbracket \Gamma; \Delta \vdash_c f : \tau_1 \rightarrow \tau_2 \rrbracket \times \llbracket \Gamma; \Delta \vdash_w u : \tau_1 \rrbracket; \textbf{eval}_{\llbracket \tau_1 \rrbracket, \llbracket \tau_2 \rrbracket}$
- $\llbracket \Gamma; \Delta, x : \tau, \Delta' \vdash_c x : \tau \rrbracket = \textbf{delete}_{\llbracket \Gamma \rrbracket} \otimes \textbf{delete}_{\llbracket \Delta \rrbracket} \otimes \textbf{id}_{\llbracket \tau \rrbracket} \otimes \textbf{delete}_{\llbracket \Delta' \rrbracket}$

---

[2] This only works because the followed by construct is restricted to wire types. The objects in $\mathcal{S}$ are not always partial streams although this is the case for the interpretations of wire terms.

- $[\![\Gamma, a : \sigma_1; \Delta \vdash_c \nu a.t : \mathrm{Circ}(\sigma_1, \sigma_2)]\!] = \mathbf{curry}(\mathbf{id}_{[\![\Gamma]\!]} \otimes \mathbf{swap}_{[\![\sigma]\!],[\![\Delta]\!]}; [\![\Gamma, a : \sigma_1; \Delta \vdash_w t : \sigma_2)]\!])$
- $[\![\Gamma; \Delta \vdash_c \lambda x.c : \tau_1 \to \tau_2]\!] = \mathbf{curry}([\![\Gamma; \Delta, x : \tau_1 \vdash_c \lambda x.c : \tau_2]\!])$

Chapter 4

Implementation

In this Chapter, we discuss details of the implementation of a compiler for the term language. The implementation draws heavily from the equational theory and semantics introduced in Section 2.5. In particular, there are strong similarities between the treatment of atoms in Section 2.9 and the extraction of individual wire names during code generation stage of the compiler. We make use of Theorems 2.7.1 and 2.9.1 to justify the correctness of the normalisation step of our compilation procedure.

The compiler was written in Haskell and is able to generate synthesisable Verilog code from source files in the term language. We will give a range of examples including the source and generated output to demonstrate the utility of this approach to circuit design. We discuss unification-based type inference and give an algorithm for performing type inference on our term language (Algorithm 1 in Section 4.4). We also discuss how Verilog code is generated and give an algorithm for the code generation step (in Section 4.6).

4.1 Overview

The compiler is constructed in a modular way and performs a sequence of transformations on the source code. A high-level breakdown of the different stages of compilation is given in Figure 4.1. The module structure is given in Figure 4.2. We represent the abstract syntax using recursion schemes [22] which allows generic traversals to be defined over a range of annotated tree types. We successively annotate variables in the syntax tree with type information and then trees of Verilog wires which represent that variable in the output file.

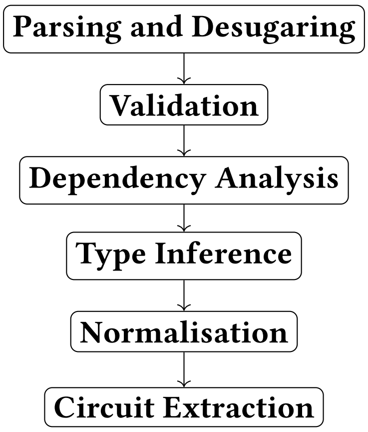


Figure 4.1: High-level structure of the compiler.

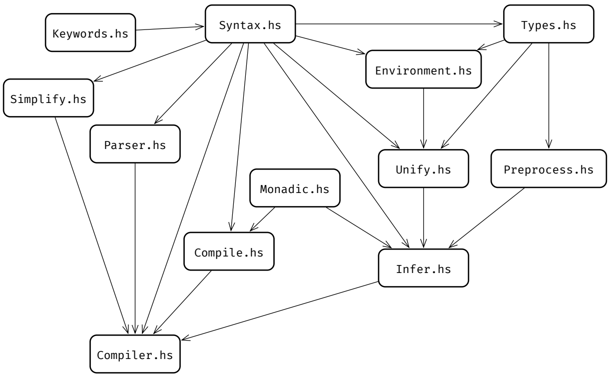
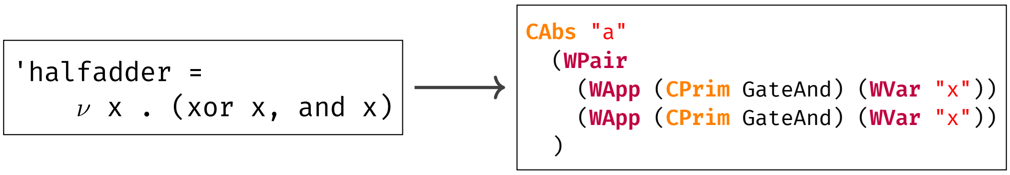


Figure 4.2: Module structure of the compiler.

4.2 Parsing, Desugaring and Validation

The source program is first parsed into the internal representation. At this stage, syntactic sugar is eliminated and fresh variables are generated where required by the desugaring. An example of the concrete and abstract syntax is given in Figure 4.3.



```
'halfadder =
    ν x . (xor x, and x)
```

```
CAbs "a"
  (WPair
    (WApp (CPrim GateAnd) (WVar "x"))
    (WApp (CPrim GateAnd) (WVar "x"))
  )
```

Figure 4.3: Parsing.

In order to allow for the generation of new variables during desugaring, monadic parsing takes place on top of a global state monad containing the necessary information for fresh name generation. Before moving on, the program is then validated to ensure that all free variables are defined and no variables are multiply defined. The main function is identified and various other correctness checks are performed to ensure the program has the correct structure.

4.3 Dependency Analysis

As the language has commutative semantics, similarly to Haskell, it is not necessarily the case that definitions are given in the order they are used. Therefore, we have to do some preprocessing work before type inference is done. This involves calculating the program's dependency graph, which is defined as follows.

Definition 4.3.1: Dependency Graph of a Program

The dependency graph is a directed graph G where each vertex is a top-level definition in the program, and there is an edge e from vertex u to vertex v precisely when the variable bound by the definition u occurs freely inside the body of the definition v.

In particular, recursive definitions correspond to vertices in G with a self-loop, and sets of mutually recursive definitions correspond to strongly connected components in G.

Now, given the original dependency graph G, we construct a new graph G' of the strongly connected components of G. The vertices in this graph are *sets* of vertices in G and there is an edge between vertices S_1, S_2 in G' precisely when there exist $v \in S_1$ and $u \in S_2$ such that the edge (v, u) is in G (for $S_1 \neq S_2$). More concretely, the edges in G' correspond to the dependencies between sets of mutually recursive definitions in the program.

It is easy to see that the graph G' forms a DAG, as the union of vertices in any cycle would form an even larger SCC in G. Hence we can perform a topological sort on the graph G'. Recursive definitions are replaced with explicit fixed points, and we perform type inference on the groups of definitions, solving constraints after each group and updating the global environment. This trick was originally described by Simon Peyton Jones [27].

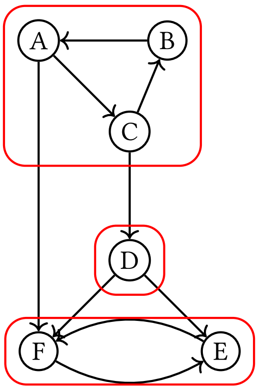


Figure 4.4: An example of a dependency graph with SCCs outlined in red.

4.4 Type Inference

A significant impediment to the adoption of more powerful type systems in hardware design is the additional burden they impose on the designer. Type inference reduces the need for type annotations and has been a major contributor to the success of strongly-typed languages such as ML [24] and Haskell [21]. We implement a unification-based type inference algorithm modelled on Hindley Milner's famous Algorithm W [23].

We start by defining new type inference rules using a constrained judgment. We write

$$\Gamma; \Delta \vdash_w u : \sigma \mid C$$

to indicate that wire term u has type σ in contexts Γ and Δ subject to the constraints C. Here C is a set of constraints of the form $\langle \sigma_1, \sigma_2 \rangle$ which require that the type equality $\sigma_1 \simeq \sigma_2$ holds. The new rules are given in Figures 4.5 and 4.6. In some cases, we require that a type variable be fresh, meaning that it does not appear in the types or constraints of the premises. We use the same notation for circuit terms as well.

$$\frac{}{\Gamma, a : \sigma, \Gamma'; \Delta \vdash_w a : \sigma \mid \emptyset}\ (\textsc{inf-name}) \qquad \frac{}{\Gamma; \Delta \vdash_w () : \text{unit} \mid \emptyset}\ (\textsc{inf-unit})$$

$$\frac{\Gamma; \Delta \vdash_w u : \sigma_1 \mid C_1 \quad \Gamma; \Delta \vdash_w v : \sigma_2 \mid C_2}{\Gamma; \Delta \vdash_w (u, v) : \sigma_1 \times \sigma_2 \mid C_1 \cup C_2}\ (\textsc{inf-prod})$$

$$\frac{\Gamma; \Delta \vdash_w s : \sigma_1 \quad \Gamma; \Delta \vdash_c c : \tau}{\Gamma; \Delta \vdash_w c\ s : \sigma_2 \mid \{\langle \tau, \text{Circ}(\sigma_1, \sigma_2) \rangle\} \cup C_1 \cup C_2}\ (\textsc{inf-circ-app})$$

$$\frac{\Gamma; \Delta \vdash_w s : \sigma_1 \mid C}{\Gamma; \Delta \vdash_w \text{fst}\ s : \sigma_2 \mid \{\langle \sigma_1, \sigma_2 \times \alpha \rangle\} \cup C \quad (\alpha\ \text{fresh})}\ (\textsc{inf-proj-1})$$

$$\frac{\Gamma; \Delta \vdash_w s : \sigma_1 \mid C}{\Gamma; \Delta \vdash_w \text{snd}\ s : \sigma_2 \mid \{\langle \sigma_1, \alpha \times \sigma_2 \rangle\} \cup C \quad (\alpha\ \text{fresh})}\ (\textsc{inf-proj-2})$$

Figure 4.5: Wire type inference rules.

$$\frac{}{\Gamma; \Delta, x : \tau, \Delta' \vdash_c x : \tau \mid \emptyset}\ (\textsc{inf-var}) \qquad \frac{g \in \mathcal{G} \quad \#g = \langle \sigma_1, \sigma_2 \rangle}{\Gamma; \Delta \vdash_c g : \text{Circ}(\sigma_1, \sigma_2) \mid \emptyset}\ (\textsc{inf-gate})$$

$$\frac{\Gamma, a : \sigma_1; \Delta \vdash_w u : \sigma_2 \mid C}{\Gamma; \Delta \vdash_c \nu a.u : \text{Circ}(\sigma_1, \sigma_2) \mid C}\ (\textsc{inf-circ-abs}) \qquad \frac{\Gamma; \Delta, x : \tau_1 \vdash_c c : \tau_2 \mid C}{\Gamma; \Delta \vdash_c \lambda x.c : \tau_1 \to \tau_2 \mid C}\ (\textsc{inf-func-abs})$$

$$\frac{\Gamma; \Delta \vdash_c c_1 : \tau_1 \mid C_1 \quad \Gamma; \Delta \vdash_c c_2 : \tau_2 \mid C_2}{\Gamma; \Delta \vdash_c c_1\ c_2 : \tau_3 \mid \{\langle \tau_1, \tau_2 \to \tau_3 \rangle\} \cup C_1 \cup C_2}\ (\textsc{inf-func-app})$$

Figure 4.6: Circuit type inference rules.

We continue by defining the notion of a type substitution function S from type variables to types. We extend the function to apply to composite types as follows.

$$S(t) := t \qquad \text{for a constant type } t \in \mathcal{T} \tag{4.1}$$

$$S(\alpha) := S(\alpha) \tag{4.2}$$

$$S(\sigma_1 \times \sigma_2) := S(\sigma_1) \times S(\sigma_2) \tag{4.3}$$

$$S(\text{Circ}(\sigma_1, \sigma_2)) := \text{Circ}(S(\sigma_1), S(\sigma_2)) \tag{4.4}$$

$$S(\sigma_1 \to \sigma_2) := S(\sigma_1) \to S(\sigma_2) \tag{4.5}$$

We say that a given type substitution S *unifies* a constraint $\langle \sigma_1, \sigma_2 \rangle$ if the function satisfies $S(\sigma_1) = S(\sigma_2)$. And we say that a substitution S unifies a set of constraints C if the function unifies each constraint in C. The following two theorems establish the correctness of the unification-based approach.

Theorem 4.4.1: Substitution Preserves Typing Judgement

(1) If $\Gamma; \Delta \vdash_w u : \sigma$ and S is a substitution, then we must have $S(\Gamma); S(\Delta) \vdash_w u : S(\sigma)$.

(2) If $\Gamma; \Delta \vdash_c c : \tau$ and S is a substitution, then we must have $S(\Gamma); S(\Delta) \vdash_c c : S(\tau)$.

Proof: By induction on the typing derivation.

Theorem 4.4.2: Unification of Constraints

(1) If $\Gamma; \Delta \vdash_w u : \sigma \mid C$ and S is a unifier for C, then we must have $S(\Gamma); S(\Delta) \vdash_w u : S(\sigma)$.

(2) If $\Gamma; \Delta \vdash_c c : \tau \mid C$ and S is a unifier for C, then we must have $S(\Gamma); S(\Delta) \vdash_c c : S(\tau)$.

Proof: Using the previous theorem and by induction on the typing derivation.

Using this idea, we can reconstruct the type of a term by traversing the structure of the term and generating constraints as required by the new typing rules. Our algorithm for type inference consists of two parts.

1. Find type subject to a set of constraints C.
2. Find a unifier S for C.

The first part can be achieved recursively based on the rules given in 4.6. The second part is also achieved recursively and builds the substitution up gradually by solving one constraint at a time as shown in Algorithm 1.

Algorithm 1 Unification algorithm.

```
match $C$ with
    case $\emptyset$ return identity
    case $\{\langle T, T\rangle\} \cup C'$ return UNIFY($C'$)
    case $\{\langle \tau_1 \to \tau_2, \tau_1' \to \tau_2'\rangle\} \cup C'$ return UNIFY($\{\langle \tau_1, \tau_1'\rangle, \langle \tau_2', \tau_2'\rangle\} \cup C'$)
    case $\{\langle \sigma_1 \times \sigma_2, \sigma_1' \times \sigma_2'\rangle\} \cup C'$ return UNIFY($\{\langle \sigma_1, \sigma_1'\rangle, \langle \sigma_2, \sigma_2'\rangle\} \cup C'$)
    case $\{\langle \mathrm{Circ}(\sigma_1, \sigma_2), \mathrm{Circ}(\sigma_1', \sigma_2')\rangle\} \cup C'$ return UNIFY($\{\langle \sigma_1, \sigma_1'\rangle, \langle \sigma_2, \sigma_2'\rangle\} \cup C'$)
    case $\{\langle \alpha, T\rangle\}$ and $\alpha \notin$ free-vars($T$) return UNIFY($C'[T/\alpha]$) $\circ[T/\alpha]$
    case $\{\langle T, \alpha\rangle\}$ and $\alpha \notin$ free-vars($T$) return UNIFY($C'[T/\alpha]$) $\circ[T/\alpha]$
    case _ return FAIL("Not Unifiable")
```

The unification algorithm works by breaking each constraint into strictly smaller constraints or by directly solving constraints where possible. As the total complexity of the set of constraints is strictly decreasing with every iteration, the algorithm always terminates. Each solved nontrivial constraint results in another substitution being added to the unifier. We can see by induction that the resulting function is a unifier for the whole set of constraints.

4.5 Normalisation

Now we repeatedly β-reduce to remove all remaining functions. By Theorems 2.7.1 and 2.7.2 from Section 2.5 we know that this normalisation step is correct and terminating. We also know by Theorem 2.9.2 that it does not change the semantics of our circuit.

4.6 Verilog Extraction

The final part of the puzzle involves extracting Verilog from the internal representation. An important difference between our language and the final representation is that we allow wires to have internal structure and might write either **and** x or **and** (x_1, y_1). So each composite wire must be decomposed into a tree of constituent wire parts, in a similar way to how the semantic interpretation of circuit terms required a sequence of atoms, given by $\text{atoms}(\Gamma)$, as an input in Section 2.9. The rough outline of our procedure is as follows

1. Assign a tree of names to each variable based on its type.
2. Generate new output wires for each atomic gate application.
3. Treat `true` and `false` as special cases of circuits generating binary values.
4. Create non-blocking assignments for the initial and subsequent subexpressions of each $\rhd$ construct.
5. Identify which names should be abstracted as wires and which should be abstracted as registers.
6. Calculate the input and output list.
7. Determine if the circuit is purely combinational or requires a clock.

The generated Verilog can then be simulated using any standard hardware toolchain.

Chapter 5

Evaluation of the Compiler

5.1 Results

The compiler (described in Chapter 4) has been tested on a variety of different example files, including both synchronous and purely combinational circuits. In each case, the generated Verilog circuit was simulated using yosys and a circuit diagram was extracted. A table of the test circuits is given in Table 5.1. In this section, we give the source code and the output circuit for each example. The generated Verilog and intermediate representation after different stages of compilation are given in the Appendix (section 7.2).

Circuit	Gates	Latches
Blink Circuit	1	1
Full Adder	5	0
Ripple-Carry Adder	16	0
Binary Counter Circuit	5	4
Fibonacci Sequence Generator	16	8

Table 5.1: Table of example circuits.

5.2 Examples

5.2.1 Blink Circuit

The first example is a simple circuit which generates an alternating sequence of bits on a single output wire.

Source code:

```
-- alternating output --
blink = false () ▷ not blink

-- main module --
'main () = blink
```

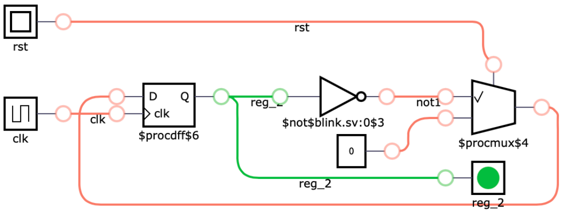


Figure 5.1: Yosys output circuit diagram for blink circuit.

5.2.2 Full Adder

The next example is the full adder circuit from Section 2.5.

Source code:

```
-- half adder --
'ha a = (xor a, and a)

-- full adder --
'fa (a,b,c) =
     let (sa, ca) = 'ha (a, b)
     in let (sb, cb) = 'ha (sa, c)
     in (sb, or(ca,cb))

-- main module --
'main = 'fa
```

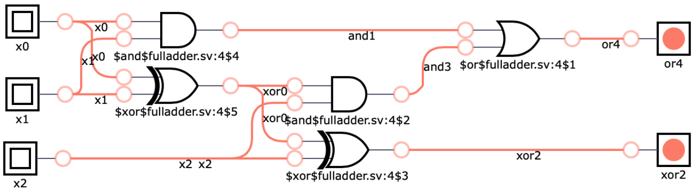


Figure 5.2: Yosys output circuit diagram for full adder circuit.

5.2.3 Ripple-Carry Adder

We chain together 4 full adder circuits to make a 4-bit ripple-carry adder. Note that the final two xor gates are combined into a single 3-input gate in the final circuit, saving a gate.

Source code (in addition to previous definitions):

```
-- 4 bit ripple-carry adder --
'ripple (x,y) =
     let (xa, xb, xc, xd) = x in
     let (ya, yb, yc, yd) = y in
     let (sa, ca) = 'ha (xa, ya) in
     let (sb, cb) = 'fa (xb, yb, ca) in
     let (sc, cc) = 'fa (xc, yc, cb) in
     let (sd, cd) = 'fa (xd, yd, cc) in
     (sa, sb, sc, sd)

-- main module --
'main = 'ripple
```

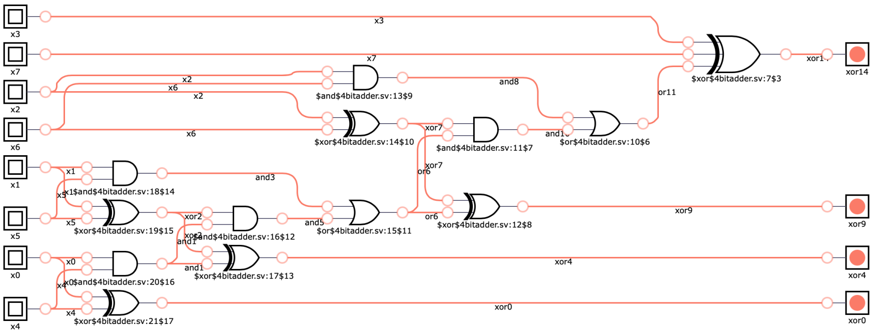


Figure 5.3: Yosys output circuit diagram for 4-bit adder circuit.

5.2.4 4-bit Binary Counter

Using the 4-bit adder, we can generate a counter. The full power of the adder is not required in this case as we are only ever incrementing the previous value. The generated Verilog code includes gates whose input is fixed, but these are optimised away when the Verilog is synthesised. The 4-bit binary constants 0 and 1 have to be defined and we use big-endian bit ordering as this is the format the adder uses.

Source code (in addition to previous definitions):

```
-- constants
zero = (false (), false (), false (), false ())
one = (true (), false (), false (), false ())

-- 4-bit counter --
count = zero ▷ 'ripple (one, count)

-- main module --
'main () = count
```

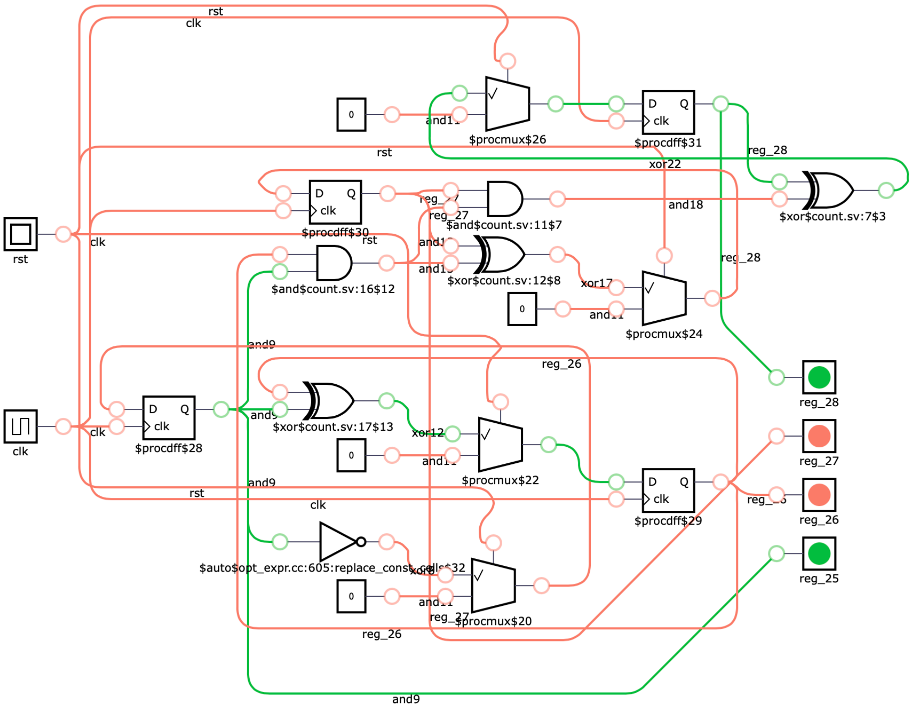


Figure 5.4: Yosys output circuit diagram for binary counter circuit.

5.2.5 Fibonacci Counter Circuit

A more interesting example is an implementation of the recursively defined Fibonacci sequence from example 3.2.3 in Section 3.2.1. This circuit makes two recursive calls and features a nested followed-by operator.

```
-- fibonacci sequence --
fib = zero ▷ 'ripple (fib, one ▷ fib)

-- main module --
'main () = fib
```

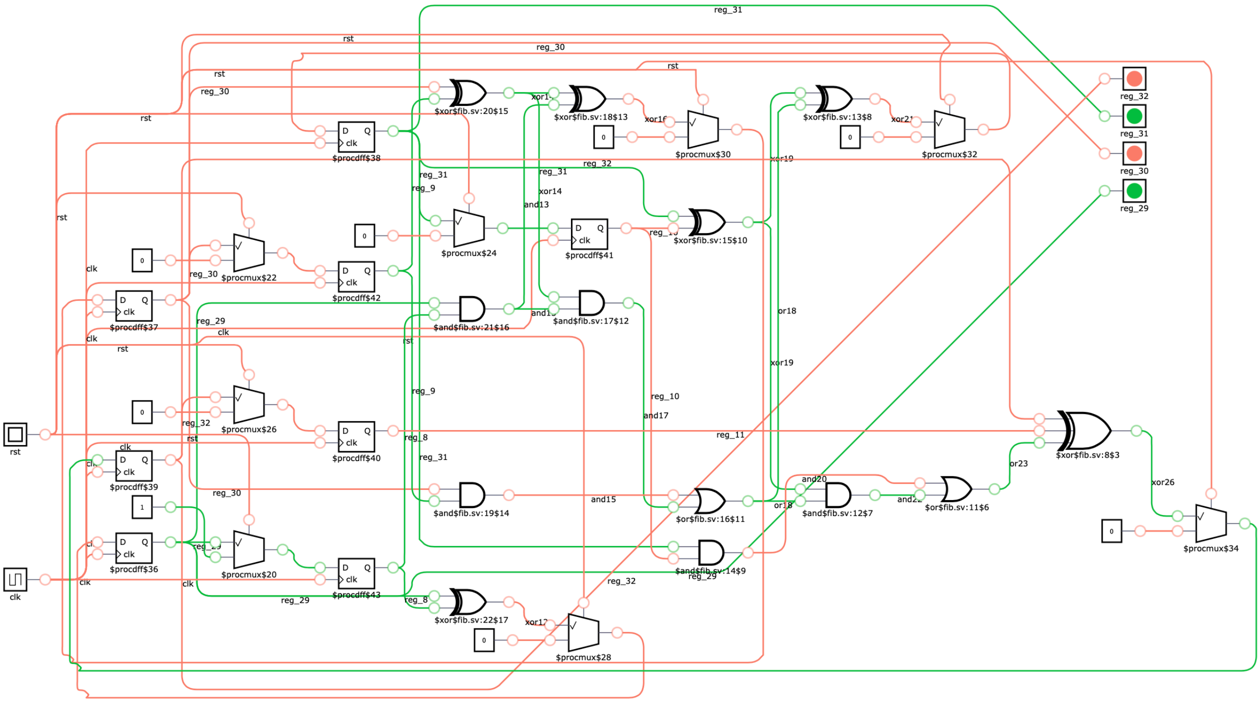


Figure 5.5: Yosys output circuit diagram for Fibonacci counter circuit.

Chapter 6

Conclusion

6.1 Summary

To summarise, we have taken a foundational look at hardware description through the lens of a new calculus (Section 2.5). Through this framework, we have formulated an equational theory and given a new categorical semantics to our language using nominal sets and equivariant functions (Section 2.9). This model has enabled a structural study of hardware described in an expressive, higher-order setting. We have also explored the potential of modal types to guard feedback in synchronous circuits (Chapter 3), which has allowed us to establish the correctness of hardware in a way that is not currently possible using mainstream languages like Verilog or VHDL. Additionally, we have implemented a compiler for our new calculus to offer a practical abstraction for the design of simple circuits (Chapter 4). Our examples (Chapter 5) have demonstrated the efficacy of this approach. Overall, we hope this work contributes towards a fuller picture of the semantics of hardware description.

6.2 Future Work

The application of methods from programming languages to hardware description continues to be an area in need of further research. One promising direction for future work is to extend the categorical semantics we presented to more expressive classes of languages. It would be particularly nice to extend the structural model of combinational circuits (Section 2.9) to the synchronous case while retaining the notions of guardedness provided by the topos of trees model.

A clear application for the work we have done is in the realm of hardware verification. Using the structural semantic model and equational theory (Chapter 2) it would be possible to verify the soundness of further abstractions, while using the behavioural model (Chapter 3) has applications in checking implementations against a desired reference behaviour, something that is currently cumbersome and usually done via model checking. Extending the semantics to more expressive type systems such as dependent types [2] or refinement types [12] has the potential to further increase the utility of denotational semantics and equational reasoning for hardware verification.

Chapter 7

Appendix

7.1 Proofs

Proof of Theorem 2.6.3 on page 19:

By induction on the typing derivation of the left-hand hypothesis.

- (Case TYP-NAME)
  - for equation 2.9, it is possible $u = a$ and so $\sigma_1 = \sigma_2$ by TYP-NAME. Then $u[s/a] = s$ and we already know $\Gamma; \Delta \vdash_w s : \sigma_1 = \sigma_2$ from the other hypothesis.
  - Otherwise, for equation 2.9 (or equation 2.11) it must be that $u[s/a] = u$ (or $u[d/x] = u$) and $u : \sigma_2$ must occur in Γ so $\Gamma; \Delta \vdash_w u : \sigma_2$ by TYP-NAME.
- (Case TYP-CIRC-APP) So $u = k\ v$ for some circuit term k and wire term v. For equation 2.9, we know $\Gamma, a : \sigma_1; \Delta \vdash_c k[s/a] : \mathrm{Circ}(\sigma_3, \sigma_2)$ (for some σ_3) and $\Gamma, a : \sigma_1; \Delta \vdash_w v[s/a] : \sigma_3$ by induction hypothesis. Then, by TYP-CIRC-APP, $\Gamma, a : \sigma_1; \Delta \vdash_w k[s/a]\ v[s/a] = (k\ v)[s/a] : \sigma_2$. The analogous reasoning for equation 2.11 gives $\Gamma; \Delta, x : \tau_1 \vdash_w (k\ v)[s/a] : \sigma$
- (Case TYP-VAR) We can apply the same reasoning as in the TYP-NAME case. For equation 2.12 we may have $c = x$ so $\tau_1 = \tau_2$ and we are done. Alternatively we know that $c[s/a] = c$ or $c[d/x] = c$ from which the result is direct.
- (Case TYP-FUNC-APP) Analogous to the TYP-CIRC-APP case.
- (Case TYP-CIRC-ABS) In this case $c = \nu b.u$ and $\tau = \mathrm{Circ}(\sigma_2, \sigma_3)$ (or $\tau_2 = \mathrm{Circ}(\sigma_2, \sigma_3)$). Considering first the case of equation 2.10, we know that $\Gamma, a : \sigma_1, b : \sigma_2; \Delta \vdash_w u : \sigma_3$. We can apply exchange (equation 2.1 from Theorem 2.6.1) to swap the order to $\Gamma, b : \sigma_2, a : \sigma_1; \Delta \vdash_w u : \sigma_3$. We can not apply the induction hypothesis yet as the righthand hypothesis

is in the wrong form, but we assume $b \notin \text{fn}(s)$ so we can apply weakening (equation 2.5 from Theorem 2.6.2) to the righthand hypothesis to get $\Gamma, b : \sigma_2; \Delta \vdash_w s : \sigma_1$ which we can then use to apply the induction hypothesis, giving $\Gamma, b : \sigma_2; \Delta \vdash_w u[s/a] : \sigma_3$. So by TYP-CIRC-ABS, we have $\Gamma; \Delta \vdash_w \nu b.u[s/a] = (\nu b.u)[s/a] : \text{Circ}(\sigma_2, \sigma_3)$.

In the case of equation 2.12, we know $\Gamma, b : \sigma_2; \Delta, x : \tau \vdash_w u : \sigma_3$. Then we just need apply another of the weakening equations (equation 2.6 from Theorem 2.6.2) to the righthand hypothesis to get $\Gamma, a : \sigma_1; \Delta \vdash_c d : \tau_1$ from which we apply the induction hypothesis to get $\Gamma; \Delta \vdash_c \nu a.u[d/x] = (\nu b.u)[d/x] : \text{Circ}(\sigma_2, \sigma_3)$ as required.

- (Case TYP-FUNC-ABS) Analogous to the TYP-CIRC-ABS case.
- The cases TYP-UNIT and TYP-GATE are trivial as no substitution takes place, and the remaining cases (TYP-PROD, TYP-PROJ-1 and TYP-PROJ-2) are straightforward from the fact that substitution carries through the product structure.

Proof of Theorem 2.7.1 on page 20:

We will prove normalisation using logical relations. We start by defining a predicate SN^Γ_τ for each circuit type τ and SN^Γ_σ for each wire type σ as follows.

$$\begin{aligned}
SN^\Gamma_\sigma(u) &:= \Gamma; \cdot \vdash_w u : \sigma \text{ and } u \Downarrow \\
SN^\Gamma_{\text{Circ}(\sigma_1,\sigma_2)}(c) &:= \Gamma; \cdot \vdash_c c : \text{Circ}(\sigma_1, \sigma_2) \text{ and } c \Downarrow \\
SN^\Gamma_{\tau_1 \to \tau_2}(f) &:= \Gamma; \cdot \vdash_c f : \tau_1 \to \tau_2 \text{ and } f \Downarrow \text{ and } \forall c. SN^\Gamma_{\tau_1}(c) \implies SN^\Gamma_{\tau_2}(f\ c)
\end{aligned}$$

The proof will be broken into two parts.

$$\begin{aligned}
\Gamma; \cdot \vdash_w u : \sigma &\overset{(1)}{\implies} SN^\Gamma_\sigma(u) \overset{(2)}{\implies} u \Downarrow \\
\Gamma; \cdot \vdash_c c : \tau &\overset{(1)}{\implies} SN^\Gamma_\tau(u) \overset{(2)}{\implies} c \Downarrow
\end{aligned}$$

The second part labelled (2) is immediate from the definition of the SN relation, so it remains to show the implications labelled (1). We will begin by proving a lemma.

Lemma 3. Closure of SN under forwards and backwards reduction

$$\Gamma; \cdot \vdash_w s : \sigma \text{ and } s \hookrightarrow u \quad \implies \quad SN^\Gamma_\sigma(s) \text{ iff } SN^\Gamma_\sigma(u) \tag{7.1}$$

$$\Gamma; \cdot \vdash_c c : \tau \ \text{ and } c \hookrightarrow d \quad \implies \quad SN^\Gamma_\tau(c) \text{ iff } SN^\Gamma_\tau(d) \tag{7.2}$$

The result for equation 7.1 is evident from the fact that $s \Downarrow$ iff $u \Downarrow$ which is all we are required to show. For equation 7.2, we proceed by induction on the structure of the type τ.

- Case 1: Here we again note that $s \Downarrow$ iff $u \Downarrow$ which gives us all that we require.
- Case 2: $\tau = \tau_1 \rightarrow \tau_2$ Consider any $\Gamma; \cdot \vdash_c c_1 : \tau_1$ satisfying $SN^{\Gamma}_{\tau_1}(c_1)$. Then we have $\Gamma; \cdot \vdash_c c\, c_1 : \tau_2$, where τ_2 satisfies equation 7.2 by the induction hypothesis. Using the reduction rule, we have $c\, c_1 \hookrightarrow d\, c_1$. This then gives $SN^{\Gamma}_{\tau_2}(d\, c_1)$. Likewise, assuming $SN^{\Gamma}_{\tau_2}(d\, c_1)$ it must be that $(d\, c_1) \Downarrow$ so $(c\, c_1) \Downarrow$ and $SN^{\Gamma}_{\tau_2}(c\, c_1)$.

Lemma 4. Validity of SN for well-typed terms

let $\Delta = x_1 : \tau_1, ..., x_m : \tau_m$.

let $\{c_j\}_{j=1}^{m}$ be closed circuit values with $SN_{\tau_j}(c_j)$ for each j.

let $\gamma = \{x_1 \mapsto c_1, ..., x_n \mapsto c_n\}$ be a finite substitution.

$$\Gamma; \Delta \vdash_w u : \sigma \implies SN^{\Gamma}_{\sigma}(\gamma(u)) \tag{7.3}$$

$$\Gamma; \Delta \vdash_c c : \tau \implies SN^{\Gamma}_{\tau}(\gamma(c)) \tag{7.4}$$

The proof is again by induction on the typing judgement.

- (Case TYP-NAME) Then $u = a_i = \gamma(a_i)$ so if $\Gamma; \Delta \vdash_w u : \sigma$ then $\Gamma; \cdot \vdash_w u : \sigma$ and $\Gamma; \cdot \vdash_w \gamma(u) : \sigma$ hence $SN^{\Gamma}_{\sigma}(\gamma(u)$.
- (Case TYP-VAR) Then $c = x_j$ and $\gamma(c) = v_j$ so $SN^{\Gamma}_{\sigma}(v_j)$ by assumption.
- (Case TYP-FUNC-APP) Then $c = f\ d$ with $\Gamma; \Delta \vdash_c f : \tau' \rightarrow \tau$ and $\Gamma; \Delta \vdash_c d : \tau'$, so by our induction hypothesis we may assume $SN^{\Gamma}_{\tau' \rightarrow \tau}(\gamma(f))$ and $SN^{\Gamma}_{\tau'}(\gamma(d))$. Using the definition of SN for function types, we have $SN^{\Gamma}_{\tau}(\gamma(f)\ \gamma(d))$ but clearly $\gamma(f)\ \gamma(d) = \gamma(f\ d) = \gamma(c)$ as required.
- (Case TYP-CIRC-APP) Then $u = c\, w$ so $\Gamma; \Delta \vdash_w w : \sigma'$ and $\Gamma; \Delta \vdash_c c : \sigma$ hence $SN^{\Gamma}_{\sigma'}(\gamma(w))$ and $SN^{\Gamma}_{\sigma}(\gamma(c))$ by induction hypothesis. Note that circuit application (unlike function application) never introduces new redexes, and so $(\gamma(c)\ \gamma(w)) \Downarrow$ and hence $(\gamma(c\, w)) \Downarrow$.
- (Case TYP-CIRC-ABS) Then $c = \nu b.s$ and $\tau = \mathrm{Circ}(\sigma', \sigma'')$. Hence $\Gamma, b : \sigma'; \Delta \vdash_w s : \sigma''$. By the induction hypothesis, we have $SN^{\Gamma, b:\sigma'}_{\sigma''}(\gamma(s))$ hence $\gamma(s) \Downarrow$. Then it is clear we have $(\nu b.s) \Downarrow$ and so $SN^{\Gamma}_{\tau}(\nu b.s)$.
- (Case TYP-FUNC-ABS) Then $c = \lambda y.d$ and $\tau = \tau' \rightarrow \tau''$. The premise of the TYP-FUNC-APP

rule gives us $\Gamma;\Delta, y : \tau' \vdash_c d : \tau''$. It is straightforward to see that as $d \Downarrow$ then $(\lambda y.d) \Downarrow$. Consider any circuit term $\Gamma; \cdot \vdash_c d' : \tau'$ satisfying $SN^{\Gamma}_{\tau'}(d')$. Suppose $d' \Downarrow d''$. We know by induction that $SN^{\Gamma}_{\tau''}(\gamma'(d))$ where γ' is the substitution γ extended with any $\{y \mapsto d''\}$. As we have $SN^{\Gamma}_{\tau''}(\gamma'(d))$ and $\gamma'(d) = \gamma(d[d''/y])$ and we have $(\lambda y.d)d' \hookrightarrow^* (\lambda y.d)d'' \hookrightarrow d[d''/y]$ then $SN^{\Gamma}_{\tau'}((\lambda y.d)d')$ and hence $SN^{\Gamma}_{\tau'}(c)$.

- The remaining cases are all structural and follow immediately from the induction hypothesis and definition of substitution.

As a corollary of Lemma 4 we get the required result.

$$\Gamma; \cdot \vdash_w u : \sigma \implies SN^{\Gamma}_{\sigma}(u) \tag{7.5}$$

$$\Gamma; \cdot \vdash_c c : \tau \implies SN^{\Gamma}_{\tau}(c) \tag{7.6}$$

This completes the proof

Proof of Theorem 2.7.2 on page 22:

Suppose we have $\Gamma; \cdot \vdash_c c : \mathrm{Circ}(\sigma_1, \sigma_2)$. Firstly, note that by theorem 2.6.1 there exists a term c' such that $c \Downarrow c'$ and c' has no redexes. From the definition, we have $\hookrightarrow \subset \equiv$, so $c \equiv c'$. Secondly, we know that each lambda abstraction introduces an arrow type via TYP-FUNC-ABS which can only be removed (by TYP-FUNC-APP) if it appears on the left of an abstraction. As there are no function abstractions on the left of applications in v, there can be no lambda abstractions in the term v. Finally, as the typing context Δ is empty, there can be no free circuit variables in v. Additionally, we know that there are no bound circuit variables (as there are no circuit abstractions), hence there can be no circuit variables (free or bound) in c'. Then we have that $c \equiv c'$ and c' is in $\lambda_{comb}*$.

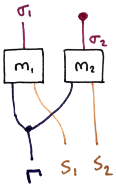
Proof of Theorem 2.9.1 on page 27:

This proof of the first part is done individually for each semantic definition. We note that in each case, the support of the input contains the support of the output. The second part is relatively straightforward, as we note that string of atoms for circuit types is finite, so simply taking the set of all atoms which occur makes it a finite support. The fact that the function spaces form a nominal set is by construction.

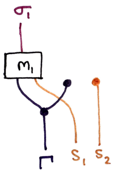
Proof of Theorem 2.9.2 on page 28:

Most of the equations can be proved diagrammatically using the comonoid laws from the Markov structure of the PROP category, and by the fact that we quotient by the addition of discarded free atoms.

$\llbracket \Gamma ; \Delta \vdash_w \mathrm{fst}\ (m_1, m_2) : \sigma_1 \rrbracket =$



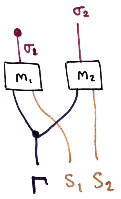
$=$



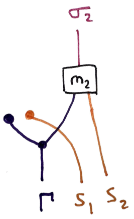
$=$

$= \llbracket \Gamma ; \Delta \vdash_w m_1 \rrbracket$

$\llbracket \Gamma ; \Delta \vdash_w \mathrm{snd}\ (m_1, m_2) : \sigma_2 \rrbracket(d) =$



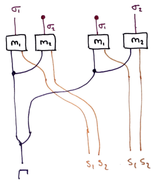
$=$



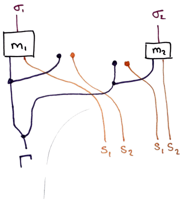
$=$

$= \llbracket \Gamma ; \Delta \vdash_w m_2 \rrbracket(d)$

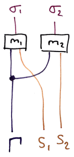
$\llbracket \Gamma ; \Delta \vdash_w (\mathrm{fst}\ s, \mathrm{snd}\ s) : \sigma_1 \times \sigma_2 \rrbracket(d) =$



$=$

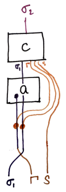


$=$



$= \llbracket \Gamma ; \Delta \vdash_w s : \sigma_1 \times \sigma_2 \rrbracket(d)$

$\llbracket \Gamma ; \Delta \vdash_c \nu a.ca : \mathrm{Circ}(\sigma_1, \sigma_2) \rrbracket(d) =$



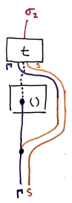
$=$

$=$

$= \llbracket \Gamma ; \Delta \vdash_w s : \sigma_1 \times \sigma_2 \rrbracket(d)$

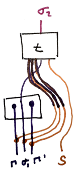
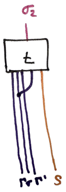
The V substitution proof is inductive on the structure of V. There are two base cases for names and unit and there is one inductive case for product types. The final step holds as all V commute through the copying map.

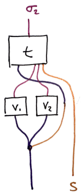
$\llbracket \Gamma ; \Delta \vdash_w (\nu a.t)() : \sigma_2 \rrbracket(d) =$



$=$

$\llbracket \Gamma, a : \mathrm{unit} ; \Delta \vdash_w t : \sigma_2 \rrbracket(d) = \llbracket \Gamma ; \Delta \vdash_w t[()/a] : \sigma_2 \rrbracket(d)$

$[\![\Gamma; \Delta \vdash_w (\nu a.t)b : \sigma_2]\!](d) =$  $=$  $= [\![\Gamma; \Delta \vdash_w t[b/a] : \sigma_2]\!](d)$

$[\![\Gamma; \Delta \vdash_w (\nu a.t)(v_1, v_2) : \sigma_2]\!](d) =$  $=$  $= [\![\Gamma; \Delta \vdash_w t[(v_1, v_2)/a] : \sigma_2]\!](d)$

The affine substitution case only applies when there are no higher-order functions, so the proof is very similar to that in [33].

$$[\![\Gamma; \Delta \vdash_w (\nu a.t)s]\!] = [\![\Gamma; \Delta \vdash_w t[s!a]]\!]$$

We use the standard, rather than diagrammatic, notation for the remaining cases.

$[\![\Gamma; \Delta \vdash_c (\lambda x.t)\ c : \tau_2]\!](s, d)$

$= h(z)$ (by circuit semantics equation 5)

where $h = [\![\Gamma; \Delta \vdash_c \lambda x.t : \tau_1 \to \tau_2]\!](s, d)$

and $z = [\![\Gamma; \Delta \vdash_c c : \tau_1]\!](s, d)$

$= h(z)$

where $h = x' \mapsto [\![\Gamma; \Delta, x : \tau_1 \vdash_c t : \tau_2]\!](s, d, x')$ (by circuit semantics equation 4)

and $z = [\![\Gamma; \Delta \vdash_c c : \tau_1]\!](s, d)$

$= [\![\Gamma; \Delta, x : \tau_1 \vdash_c t : \tau_2]\!](s, d, z)$

where $z = [\![\Gamma; \Delta \vdash_c c : \tau_1]\!](s, d)$

$= [\![\Gamma; \Delta \vdash_c t[c/x] : \tau_2]\!](s, d)$

$[\![\Gamma; \Delta \vdash_c \lambda x.f\ x : \tau_1 \to \tau_2]\!](s, d)$

$= x' \mapsto [\![\Gamma; \Delta, x : \tau_1 \vdash_c f\ x : \tau_2]\!](s, d, x')$ (by circuit semantics equation 4)

$= x' \mapsto h(z)$ (by circuit semantics equation 5)

where $h = [\![\Gamma; \Delta, x : \tau_1 \vdash_c f : \tau_1 \to \tau_2]\!](s, d, x')$

and $z = [\![\Gamma; \Delta, x : \tau_1 \vdash_c x : \tau_1]\!](s, d, x')$

$= x' \mapsto h(z)$

where $h = [\![\Gamma; \Delta \vdash_c f : \tau_1 \to \tau_2]\!](s, d)$ (as $x \notin \mathrm{fv}(f)$ and by discarding)

and $z = x'$ (by circuit semantics equation 1)

$= [\![\Gamma; \Delta \vdash_c f : \tau_1 \to \tau_2]\!](s, d)$

7.2 Generated Verilog

7.2.1 Blink Circuit

Source code:

```
-- alternating output --
blink = false () ▷ not blink

-- main module --
'main () = blink
```

After type inference and desugaring:

```
blink : bit
blink = (fix (νblink.((false ()) ▷ (not blink))))

'main : Circ(unit, bit)
'main = (ν_unit_pattern_.blink)
```

After definition unfolding:

```
'main : Circ(unit, bit)
'main = (ν_unit_pattern_.((νblink.blink) (fix (νblink.((false ()) ▷ (
   ↪ not blink))))))
```

After simplification:

```
'main : Circ(unit, bit)
'main = (ν_unit_pattern_.(fix (νblink.((false ()) ▷ (not blink)))))
```

Generated Verilog:

```
module main(reg_2, clk, rst);
    input clk, rst;
    output reg reg_2;
    wire not1;
    not gate_not1(not1, reg_2);
    always @ (posedge clk) begin
        reg_2 <= not1;
        if (rst == 1) begin
            reg_2 <= 1'b0;
        end
    end
endmodule
```

7.2.2 Full Adder

Source code:

```
-- half adder --
'ha a = (xor a, and a)

-- full adder --
'fa (a,b,c) =
     let (sa, ca) = 'ha (a, b)
     in let (sb, cb) = 'ha (sa, c)
     in (sb, or(ca,cb))

-- main module --
'main = 'fa
```

After type inference and desugaring:

```
'ha : Circ((bit, bit), (bit, bit))
'ha = (νa.((xor a), (and a)))

'fa : Circ((bit, (bit, bit)), (bit, bit))
'fa = (ν_t0.((νa.((νb.((νc.((ν_t1.((νsa.((νca.((ν_t2.((νsb.((νcb.(sb, (
   ↪ or (ca, cb)))) (π₂ _t2))) (π₁ _t2))) ('ha (sa, c)))) (π₂ _t1))) (
   ↪ π₁ _t1))) ('ha (a, b)))) (π₂ (π₂ _t0)))) (π₁ (π₂ _t0)))) (π₁ _t0))
   ↪ )

'main : Circ((bit, (bit, bit)), (bit, bit))
'main = (νx.('fa x))
```

After definition unfolding:

```
'main : Circ((bit, (bit, bit)), (bit, bit))
'main = (νx.((ν_t0.((νa.((νb.((νc.((ν_t1.((νsa.((νca.((ν_t2.((νsb.((νcb
   ↪ .(sb, (or (ca, cb)))) (π₂ _t2))) (π₁ _t2))) ((νa.((xor a), (and a
   ↪ ))) (sa, c)))) (π₂ _t1))) (π₁ _t1))) ((νa.((xor a), (and a))) (a,
   ↪  b)))) (π₂ (π₂ _t0)))) (π₁ (π₂ _t0)))) (π₁ _t0))) x))
```

After simplification:

```
'main : Circ((bit, (bit, bit)), (bit, bit))
'main = (νx.((ν_t0.((ν_t1.((ν_t2.((π₁ _t2), (or ((π₂ _t1), (π₂ _t2)))))
   ↪ ((νa.((xor a), (and a))) ((π₁ _t1), (π₂ (π₂ _t0)))))) ((νa.((xor
   ↪ a), (and a))) ((π₁ _t0), (π₁ (π₂ _t0)))))) x))
```

Generated Verilog:

```
module main(x0, x1, x2, xor2, or4);
    input x0, x1, x2;
    output wire xor2, or4;
    wire and3, and1, xor0;
    or gate_or4(or4, and1, and3);
    and gate_and3(and3, xor0, x2);
    xor gate_xor2(xor2, xor0, x2);
    and gate_and1(and1, x0, x1);
    xor gate_xor0(xor0, x0, x1);
endmodule
```

7.2.3 Ripple-Carry Adder

Source code (in addition to previous definitions):

```
-- 4 bit ripple-carry adder --
'ripple (x,y) =
     let (xa, xb, xc, xd) = x in
     let (ya, yb, yc, yd) = y in
     let (sa, ca) = 'ha (xa, ya) in
     let (sb, cb) = 'fa (xb, yb, ca) in
     let (sc, cc) = 'fa (xc, yc, cb) in
     let (sd, cd) = 'fa (xd, yd, cc) in
     (sa, sb, sc, sd)

-- main module --
'main = 'ripple
```

After type inference and desugaring:

```
'ripple : Circ( ((bit, (bit, (bit, bit))), (bit, (bit, (bit, bit))))
, (bit, (bit, (bit, bit))) )
'ripple = (ν_t3.((νx.((νy.((ν_t4.((νxa.((νxb.((νxc.((νxd.((ν_t5.((νya
   ↪ .((νyb.((νyc.((νyd.((ν_t6.((νsa.((νca.((ν_t7.((νsb.((νcb.((ν_t8
   ↪ .((νsc.((νcc.((ν_t9.((νsd.((νcd.(sa, (sb, (sc, sd)))) (π₂ _t9)))
   ↪ (π₁ _t9))) ('fa (xd, (yd, cc))))) (π₂ _t8))) (π₁ _t8))) ('fa (xc,
   ↪  (yc, cb))))) (π₂ _t7))) (π₁ _t7))) ('fa (xb, (yb, ca))))) (π₂
   ↪ _t6))) (π₁ _t6))) ('ha (xa, ya)))) (π₂ (π₂ (π₂ _t5))))) (π₁ (π₂ (π₂
   ↪  _t5))))) (π₁ (π₂ _t5)))) (π₁ _t5))) y)) (π₂ (π₂ (π₂ _t4))))) (π₁ (
   ↪ π₂ (π₂ _t4))))) (π₁ (π₂ _t4)))) (π₁ _t4))) x)) (π₂ _t3))) (π₁ _t3))
   ↪ )

'main : Circ( ((bit, (bit, (bit, bit))), (bit, (bit, (bit, bit)))), (
   ↪ bit, (bit, (bit, bit))) )
'main = (νx.('ripple x))
```

Generated Verilog:

```
module main(x0, x1, x2, x3, x4, x5, x6, x7, xor0, xor4, xor9, xor14);
    input x0, x1, x2, x3, x4, x5, x6, x7;
    output wire xor0, xor4, xor9, xor14;
    wire or16, and15, and13, xor12, or11, and10, and8, xor7, or6, and5,
       ↪  and3, xor2, and1;
    or gate_or16(or16, and13, and15);
    and gate_and15(and15, xor12, or11);
    xor gate_xor14(xor14, xor12, or11);
    and gate_and13(and13, x3, x7);
    xor gate_xor12(xor12, x3, x7);
    or gate_or11(or11, and8, and10);
    and gate_and10(and10, xor7, or6);
    xor gate_xor9(xor9, xor7, or6);
    and gate_and8(and8, x2, x6);
    xor gate_xor7(xor7, x2, x6);
    or gate_or6(or6, and3, and5);
    and gate_and5(and5, xor2, and1);
    xor gate_xor4(xor4, xor2, and1);
    and gate_and3(and3, x1, x5);
    xor gate_xor2(xor2, x1, x5);
    and gate_and1(and1, x0, x4);
    xor gate_xor0(xor0, x0, x4);
endmodule
```

7.2.4 4-bit Binary Counter

Source Code:

```
-- constants
zero = (false (), false (), false (), false ())
one = (true (), false (), false (), false ())

-- 4-bit counter --
count = zero ▷ 'ripple (one, count)

-- main module --
'main () = count
```

Generated Verilog:

```
module main(reg_25, reg_26, reg_27, reg_28, clk, rst);
    input clk, rst;
    output reg reg_25, reg_26, reg_27, reg_28;
    wire or24, and23, xor22, and21, xor20, or19, and18, xor17, and16,
       ↪ xor15, or14, and13, xor12, and11, xor10, and9, xor8;
    or gate_or24(or24, and21, and23);
    and gate_and23(and23, xor20, or19);
    xor gate_xor22(xor22, xor20, or19);
    and gate_and21(and21, 1'b0, reg_28);
    xor gate_xor20(xor20, 1'b0, reg_28);
    or gate_or19(or19, and16, and18);
    and gate_and18(and18, xor15, or14);
    xor gate_xor17(xor17, xor15, or14);
    and gate_and16(and16, 1'b0, reg_27);
    xor gate_xor15(xor15, 1'b0, reg_27);
    or gate_or14(or14, and11, and13);
    and gate_and13(and13, xor10, and9);
    xor gate_xor12(xor12, xor10, and9);
    and gate_and11(and11, 1'b0, reg_26);
    xor gate_xor10(xor10, 1'b0, reg_26);
    and gate_and9(and9, 1'b1, reg_25);
    xor gate_xor8(xor8, 1'b1, reg_25);
    always @ (posedge clk) begin
        reg_28 <= xor22;
        reg_27 <= xor17;
        reg_26 <= xor12;
        reg_25 <= xor8;
        if (rst == 1) begin
            reg_28 <= 1'b0;
            reg_27 <= 1'b0;
            reg_26 <= 1'b0;
            reg_25 <= 1'b0;
        end
    end
endmodule
```

7.2.5 Fibonacci Counter Circuit

Source Code:

```
-- fibonacci sequence --
fib = zero ▷ 'ripple (fib, one ▷ fib)

-- main module --
'main () = fib
```

Generated Verilog:

```
module main(reg_29, reg_30, reg_31, reg_32, clk, rst);
    input clk, rst;
    output reg reg_29, reg_30, reg_31, reg_32;
    reg reg_11, reg_10, reg_9, reg_8;
    wire or28, and27, xor26, and25, xor24, or23, and22, xor21, and20,
       ↪ xor19, or18, and17, xor16, and15, xor14, and13, xor12;
    or gate_or28(or28, and25, and27);
    and gate_and27(and27, xor24, or23);
    xor gate_xor26(xor26, xor24, or23);
    and gate_and25(and25, reg_32, reg_11);
    xor gate_xor24(xor24, reg_32, reg_11);
    or gate_or23(or23, and20, and22);
    and gate_and22(and22, xor19, or18);
    xor gate_xor21(xor21, xor19, or18);
    and gate_and20(and20, reg_31, reg_10);
    xor gate_xor19(xor19, reg_31, reg_10);
    or gate_or18(or18, and15, and17);
    and gate_and17(and17, xor14, and13);
    xor gate_xor16(xor16, xor14, and13);
    and gate_and15(and15, reg_30, reg_9);
    xor gate_xor14(xor14, reg_30, reg_9);
    and gate_and13(and13, reg_29, reg_8);
    xor gate_xor12(xor12, reg_29, reg_8);
    always @ (posedge clk) begin
        reg_32 <= xor26;
        reg_31 <= xor21;
        reg_30 <= xor16;
        reg_29 <= xor12;
        reg_11 <= reg_32;
        reg_10 <= reg_31;
        reg_9 <= reg_30;
        reg_8 <= reg_29;
        if (rst == 1) begin
            reg_32 <= 1'b0;
            reg_31 <= 1'b0;
            reg_30 <= 1'b0;
            reg_29 <= 1'b0;
            reg_11 <= 1'b0;
            reg_10 <= 1'b0;
            reg_9 <= 1'b0;
            reg_8 <= 1'b1;
        end
    end
endmodule
```

References

[1] Samson Abramsky and Bob Coecke. *A categorical semantics of quantum protocols.* 2007. arXiv: quant-ph/0402130 [quant-ph].

[2] Thorsten Altenkirch et al. “ΠΣ: Dependent Types without the Sugar”. In: *Functional and Logic Programming*. Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 40–55. ISBN: 978-3-642-12251-4.

[3] Arvind Arvind. “Bluespec: A language for hardware design, simulation, synthesis and verification Invited Talk.” In: Jan. 2003, pp. 249–.

[4] C. Baaij. *CλasH : from Haskell to hardware.* Dec. 2009. URL: http://essay.utwente.nl/59482/.

[5] Andrej Bauer and Matija Pretnar. “Programming with Algebraic Effects and Handlers”. In: *CoRR* abs/1203.1539 (2012). arXiv: 1203.1539. URL: http://arxiv.org/abs/1203.1539.

[6] Gérard Berry and Laurent Cosserat. “The ESTEREL synchronous programming language and its mathematical semantics”. In: *Seminar on Concurrency.* Ed. by Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 389–448. ISBN: 978-3-540-39593-5.

[7] Lars Birkedal et al. “First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees”. In: *2011 IEEE 26th Annual Symposium on Logic in Computer Science.* 2011, pp. 55–64. DOI: 10.1109/LICS.2011.16.

[8] Per Bjesse et al. “Lava: Hardware Design in Haskell”. In: *SIGPLAN Not.* 34.1 (Sept. 1998), pp. 174–184. ISSN: 0362-1340. DOI: 10.1145/291251.289440. URL: https://doi.org/10.1145/291251.289440.

[9] P. Caspi et al. “LUSTRE: A Declarative Language for Real-Time Programming”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL ’87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 178–188. ISBN: 0897912152. DOI: 10.1145/41625.41641. URL: https://doi.org/10.1145/41625.41641.

[10] Kenta Cho and Bart Jacobs. “Disintegration and Bayesian inversion via string diagrams”. In: *Mathematical Structures in Computer Science* 29.7 (Mar. 2019), pp. 938–971. DOI: 10.1017/s0960129518000488. URL: https://doi.org/10.1017%2Fs0960129518000488.

[11] Koen Claessen and David Sands. “Observable Sharing for Functional Circuit Description”. In: vol. 1742. Nov. 1999, pp. 78–78. ISBN: 978-3-540-66856-5. DOI: 10.1007/3-540-46674-6_7.

[12] Tim Freeman and Frank Pfenning. “Refinement Types for ML”. In: *SIGPLAN Not.* 26.6 (May 1991), pp. 268–277. ISSN: 0362-1340. DOI: 10.1145/113446.113468. URL: https://doi.org/10.1145/113446.113468.

[13] Tobias Fritz. “A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics”. In: *Advances in Mathematics* 370 (Aug. 2020), p. 107239. DOI: 10.1016/j.aim.2020.107239. URL: https://doi.org/10.1016%2Fj.aim.2020.107239.

[14] Murdoch Gabbay and Andrew Pitts. “A New Approach to Abstract Syntax with Variable Binding”. In: *Formal Asp. Comput.* 13 (July 2002), pp. 341–363. DOI: 10.1007/s001650200016.

[15] Dan R. Ghica. “Geometry of Synthesis: A Structured Approach to VLSI Design”. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’07. Nice, France: Association for Computing Machinery, 2007, pp. 363–375. ISBN: 1595935754. DOI: 10.1145/1190216.1190269. URL: https://doi.org/10.1145/1190216.1190269.

[16] Dan R. Ghica and Achim Jung. “Categorical semantics of digital circuits”. In: *2016 Formal Methods in Computer-Aided Design (FMCAD).* 2016, pp. 41–48. DOI: 10.1109/FMCAD.2016.7886659.

[17] Dan R. Ghica, George Kaye, and David Sprunger. *A compositional theory of digital circuits*. 2023. arXiv: 2201.10456 [cs.LO].

[18] Neelakantan R. Krishnaswami and Nick Benton. "Ultrametric Semantics of Reactive Programs". In: *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*. LICS '11. USA: IEEE Computer Society, 2011, pp. 257–266. ISBN: 9780769544120. DOI: 10.1109/LICS.2011.38. URL: https://doi.org/10.1109/LICS.2011.38.

[19] Tom Leinster. *Basic Category Theory*. 2016. arXiv: 1612.09375 [math.CT].

[20] J. M. Lucassen and D. K. Gifford. "Polymorphic Effect Systems". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 47–57. ISBN: 0897912527. DOI: 10.1145/73560.73564. URL: https://doi.org/10.1145/73560.73564.

[21] Simon Marlow et al. "Haskell 2010 language report". In: *Available online http://www. haskell. org/(May 2011)* (2010).

[22] Erik Meijer, Maarten Fokkinga, and Ross Paterson. "Functional programming with bananas, lenses, envelopes and barbed wire". In: *Functional Programming Languages and Computer Architecture*. Ed. by John Hughes. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 124–144. ISBN: 978-3-540-47599-6.

[23] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[24] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.

[25] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(91)90052-4. URL: https://www.sciencedirect.com/science/article/pii/0890540191900524.

[26] H. Nakano. "A modality for recursion". In: *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. 2000, pp. 255–266. DOI: 10.1109/LICS.2000.855774.

[27] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. USA: Prentice-Hall, Inc., 1987. ISBN: 013453333X.

[28] Andrew Pitts. *Nominal Sets and their Applications*. https://www.cl.cam.ac.uk/teaching/1314/L23/lectures/lecture-1.pdf. Accessed: 2023-05-09. 2013.

[29] Marc Pouzet. *Lucid Synchrone, version 3*. Research Report. Université Paris Sud Orsay ; Laboratoire de Recherche en Informatique [LRI], UMR 8623, Bâtiments 650-660, Université Paris-Sud, 91405 Orsay Cedex, Apr. 2006. URL: https://hal.science/hal-03090137.

[30] M Sheeran. *RUBY-a Language of Relations and Higher Order Functions*. Tech. rep. Technical report, Glasgow University, 1986.

[31] Mary Sheeran. "Hardware Design and Functional Programming: a Perfect Match". In: *JUCS - Journal of Universal Computer Science* 11.7 (2005), pp. 1135–1158. ISSN: 0948-695X. DOI: 10.3217/jucs-011-07-1135. eprint: https://doi.org/10.3217/jucs-011-07-1135. URL: https://doi.org/10.3217/jucs-011-07-1135.

[32] Sam Staton. "Instances of Computational Effects: An Algebraic Perspective". In: *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. 2013, pp. 519–519. DOI: 10.1109/LICS.2013.58.

[33] Dario Stein. "Structural Foundations for Probabilistic Programming Languages". PhD thesis. University of Oxford, 2021.

[34] Dario Stein and Sam Staton. *Compositional Semantics for Probabilistic Programs with Exact Conditioning*. 2021. arXiv: 2101.11351 [cs.PL].