Timely Computation

CONAL ELLIOTT

This paper addresses the question "what is a digital circuit?" in relation to the fundamentally analog nature of actual (physical) circuits. A simple informal definition is given and then formalized in the proof assistant Agda. At the heart of this definition is the *timely* embedding of *discrete* information in temporally *continuous* signals. Once this embedding is defined (in constructive logic, i.e., type theory), it is extended in a generic fashion from one signal to many and from simple boolean operations (logic gates) to arbitrarily sophisticated sequential and parallel compositions, i.e., to computational circuits.

Rather than constructing circuits and *then* trying to prove their correctness, a *compositionally correct* methodology maintains specification, implementation, timing, and correctness proofs at every step. Compositionality of each aspect and of their combination is supported by a single, shared algebraic vocabulary and related by homomorphisms. After formally defining and proving these notions, a few key transformations are applied to reveal the *linearity* of circuit timing (over a suitable semiring), thus enabling practical, modular, and fully verified timing analysis as linear maps over higher-dimensional time intervals.

An emphasis throughout the paper is simplicity and generality of specification, minimizing circuit-specific definitions and proofs while highlighting a broadly applicable methodology of scalable, compositionally correct engineering through simple denotations and homomorphisms.

CCS Concepts: • **Hardware** → **Timing analysis**; **Functional verification**; • **Software and its engineering** → *Automated static analysis*; • **Theory of computation** → **Logic and verification**; *Type theory*; **Program verification**; *Categorical semantics*.

Additional Key Words and Phrases: compositional correctness; digital design; linear algebra

ACM Reference Format:
Conal Elliott. 2023. Timely Computation. *Proc. ACM Program. Lang.* 7, ICFP, Article 219 (August 2023), 25 pages.
https://doi.org/10.1145/3607861

1 INTRODUCTION

If you use or program computers, you are probably comfortable with a phenomenon that—upon closer inspection and reflection—appears rather odd and magical. The very nature of an electronic computer differs profoundly from what we use it for. A computer is an *analog* electronic circuit, and so consumes and produces collections of *electrical signals*, which are continuous in time and in value (voltage). In contrast, the calculations we use computers to accomplish consume and produce discrete information.[1]

Even more fundamentally, electronic circuits are *concrete*, while calculations are *abstract*. The inner workings of a circuit (like a program text) can be inspected, analyzed, and rearranged. In contrast, a calculation is a *function* (in the mathematical sense), so it can only be applied to inputs to generate outputs. While we can examine two circuits (or programs) with the same input/output

[1]Even when this discrete information serves to *approximate* some other continuous, physical information, we don't ask our computational representations to *be* continuous.

Author's address: conal@conal.net.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/8-ART219
https://doi.org/10.1145/3607861

$$\begin{array}{ccc} \Phi^o\ \tilde{A} & \xrightarrow{\quad \Phi^m\ \tilde{f} \quad} & \Phi^o\ \tilde{B} \\ \Big\downarrow h & \circlearrowleft & \Big\downarrow k \\ \varphi^o\ A & \xrightarrow{\quad \varphi^m\ f \quad} & \varphi^o\ B \end{array}$$

Fig. 1. Compositionally correct computing

behavior and notice differences in their implementations and resource use (e.g., time, area, and power consumption), functions hide such differences.[2]

Given these profound differences, what do we mean when we say that a circuit implements a function on discrete data? In other words, what is a *digital circuit*? Any definition must explain the transition both from continuous to discrete and from concrete to abstract. Considering the practical and philosophical importance of digital computing, it is worth taking care to formulate a definition that satisfies a few important criteria:

- *Precise*, so that we will know when we have successfully satisfied the definition.
- *Simple*, since the definition itself is the one facet of investigation that cannot be objectively verified.
- *Useful*, so that satisfying the definition helps us solve practical and challenging problems.
- *Formal*, so that proposed instances of the definition can be verified objectively, infallibly, and automatically.
- *Constructive*, so that definition instances have computational interpretations, i.e., yield *executable insight*.
- *Compositional*, so that correctness is integrated into constructions at every step—composing smaller correct digital computations into larger ones ones—enabling scalability.

This paper proposes the definition pictured in Figure 1, formally expressed and verified in Agda [Norell 2008; Bove et al. 2009] in later sections of this paper, and summarized as follows:[3]

A "digital circuit" is an analog circuit that respects discrete meanings.

A few remarks about Figure 1:

- $\tilde{A}$ and A concretely represent (multi-)signals and values, as interpreted by Φ^o and φ^o respectively ("o" for "objects"); and likewise for $\tilde{B}$ and B.
- $\tilde{f}$ and f concretely represent functions over signals and values, as interpreted by Φ^m and φ^m respectively ("m" for "morphisms").
- h and k extract discrete meanings from continuous mathematical signals.
- $\circlearrowleft$ proves a commutativity condition saying that the two paths from $\Phi^o\ \tilde{A}$ to $\varphi^o\ B$ are extensionally equal, i.e., $\varphi^m\ f \circ h \doteq k \circ \Phi^m\ \tilde{f}$.

The sense in which the representation $\tilde{f}$ in Figure 1 (eventually a circuit) "respects discrete meanings" is that $\Phi^m\ \tilde{f}$ maps semantically equivalent inputs to semantically equivalent outputs, i.e., for

[2]Machine *implementations* of functions, on the other hand, can be profiled for time and space use. Optimization (manual or automated) is about embracing both sides of this distinction: improve operational efficiency while leaving denotation unchanged. Although the code and proofs below won't manipulate the actual physical circuits they describe, they will manipulate a representation sufficiently tangible to extract instructions for fabricating physical circuits.

[3]As captioned, Figure 1 describes compositional correctness much more generally. What makes it relevant to digital circuits in particular is that the targets of Φ^m and φ^m in this paper will be signal and value functions respectively. The *compositional* aspect of Figure 1 will be explained in Section 9.

any input signals $\tilde{u}$ and $\tilde{v}$, if $h\ \tilde{u} \equiv h\ \tilde{v}$, then $k\ (\Phi^m\ \tilde{f}\ \tilde{u}) \equiv k\ (\Phi^m\ \tilde{f}\ \tilde{v})$. This conclusion follows directly from the premise and commutativity: $k\ (\Phi^m\ \tilde{f}\ \tilde{u}) \equiv \varphi^m\ f\ (h\ \tilde{u}) \equiv \varphi^m\ f\ (h\ \tilde{v}) \equiv k\ (\Phi^m\ \tilde{f}\ \tilde{v})$.

This paper explores the logical and practical implications of this definition of digital circuits. Figure 1 will accompany us throughout our journey as we clarify how to encode data in signals, how to constrain analog computations so that those embeddings exist, and how to synthesize physical circuits together with timing information sufficient to guarantee their correct execution. Because compositionality is crucial for constructing sophisticated computational artifacts, and homomorphism is a powerful principle for defining and proving correctness, *elementary* category theory provides a suitable and consistent language throughout.

This paper makes the following contributions:

- A specification and method for correct hardware design rooted in the temporally continuous nature of actual hardware rather than the common assumption of a discrete, clocked model. The shift to continuous time enables addressing subtleties of pin-to-pin delays within gates and across sequential and parallel compositions, leading to elegantly specified and formally verified automatic timing analysis for complex circuits. One such subtlety is computing correctly in the presence of the timing glitches that inevitably arise from unequal signal propagation delays within circuits.
- Identification and definition of *signal stability* (temporary constancy of discrete interpretation) as the key logical ingredient in defining digital computation as a disciplined use of analog computation. An important choice here is formulating stability in terms of *time intervals*.
- A compositional and constructively logical notion of functions over constrained values (similar to refinement types) and their use to extend stability from individual signals and gates to multi-signals and circuits. With stability, the constructive interpretation of this logical notion performs timing analysis as functions on time multi-intervals and constructs correctness proofs for the results.
- Illustration of a general methodology for compositionally correct engineering using a domain-independent construction known as a “comma category”. Each comma morphism encapsulates representations of the data and operations involved, their mathematical interpretations, specification *and* implementation of a computation, the relationship between specification and implementation data types, and correctness proofs. Crucially, this recipe forms a cartesian category and so is highly composable in the same language as each ingredient.
- Extraction of timing via small alterations to the mentioned interpretations (functors).
- Specialization to *static* (data-independent) timing to enable correctly clocked, globally synchronous circuit implementations, via a simple shift to one of the functors that defines the comma category.
- Observation and proof that timing analysis is *linear* in a space of multi-dimensional time intervals, leading to much more practical and flexible timing analysis than with unconstrained timing functions.

This paper is a fully proved literate Agda program[4], so all of the syntax-colored, indented code you will see below is proof-checked before type-setting.

2 PURE ANALOG COMPUTATION

Since digital computation is a disciplined use and discrete interpretation of analog computation, let’s start with analog computing. To simplify the exposition, we’ll use *semi-analog* signals: continuous in time and discrete in value. Extending to fully analog signals appears not to raise significant

[4] There are no postulates in the category theory library [Elliott 2023] or in code specific to this paper, and all modules are compiled with the `--safe` and `--without-K` pragmas.

$$\mathsf{analog}_0 : \mathbb{T}^{\,0} \to (\mathbb{B}^{\,0} \to \mathbb{B}) \to (\mathbb{S}^{\,0} \to \mathbb{S})$$
$$\mathsf{analog}_0\ \mathsf{tt}\ h\ \mathsf{tt} = \lambda\ t \to h\ \mathsf{tt}$$

$$\mathsf{analog}_1 : \mathbb{T} \to (\mathbb{B} \to \mathbb{B}) \to (\mathbb{S} \to \mathbb{S})$$
$$\mathsf{analog}_1\ \delta\ h\ \tilde{x} = \lambda\ t \to h\ (\tilde{x}\ (t - \delta))$$

$$\mathsf{analog}_2 : \mathbb{T}^{\,2} \to (\mathbb{B}^{\,2} \to \mathbb{B}) \to (\mathbb{S}^{\,2} \to \mathbb{S})$$
$$\mathsf{analog}_2\ (\delta_1\ ,\ \delta_2)\ h\ (\tilde{x}_1\ ,\ \tilde{x}_2) = \lambda\ t \to h\ (\tilde{x}_1\ (t - \delta_1)\ ,\ \tilde{x}_2\ (t - \delta_2))$$

Fig. 2. Analog logic gate templates

challenges, as described in Section 16. A signal ($\mathbb{S}$) is thus a function from continuous time ($\mathbb{T}$, measured in nanoseconds and represented as a rational number) to booleans ($\mathbb{B}$):

$$\mathbb{T} : \mathsf{Set} \qquad\qquad \mathbb{S} : \mathsf{Set}$$
$$\mathbb{T} = \mathbb{Q} \qquad\qquad \mathbb{S} = \mathbb{T} \to \mathbb{B}$$

A (semi-)analog computation is a transformation on "multi-signals", i.e., aggregates (nested products) of signals.

We will work with a handful of primitive signal functions (analog logic gates), each having one output and either one or two inputs. Playing loose with terminology, we can also consider false and true to be gates with no inputs (though they are really implemented as connections to ground or high voltage). We can thus define all analog gates in terms of templates that describe zero-, one-, or two-input gates, as shown in Figure 2.[5,6] Each analog gate template takes a delay between each input pin and the single output pin along with a function on booleans. The generated output signal sampled at time t equals the function applied to the input signals sampled *earlier* than t by the given delays. For exposition, assume the following roughly realistic delays (in nanoseconds) for our logic gates:[7]

$$\delta\text{-false} = \mathsf{tt} \qquad\qquad \delta\text{-nand} = \mathsf{dup}\ (1\ /\ 5)$$
$$\delta\text{-true} = \mathsf{tt} \qquad\qquad \delta\text{-nor} = \mathsf{dup}\ (1\ /\ 5)$$
$$\delta\text{-not} = 1\ /\ 10 \qquad\qquad \delta\text{-xor} = \mathsf{dup}\ (1\ /\ 4)$$

For instance, an analog *nand* (negated *and*) gate:

$$\mathsf{nand}^{\mathsf{A}} : \mathbb{S}^{\,2} \to \mathbb{S}$$
$$\mathsf{nand}^{\mathsf{A}} = \mathsf{analog}_2\ \delta\text{-nand}\ \mathsf{nand}$$

3 TIME AND DIGITAL COMPUTATION

Continuous time is an intrinsic aspect of analog computing and has simple semantics as we've just seen. In *digital* computation, however, time plays a subtle role that often leads to confusion and costly mistakes.

[5] I've introduced two definitions to help readability: $A^{\,0} = \top$, and $A^{\,2} = A \times A$.

[6] As a simplification, the definitions in Figure 2 deterministically assign a value to every point in time, while actual circuits involve unstable and unpredictable periods. The logical analysis described in the rest of this paper (correctly) avoids making any promises about these problematic periods.

[7] Here, tt is the sole element of the unit (nullary product) type $\top$, and dup : a → a × a (duplication).

Because we compute via physical processes, we generally accept that results of a computation become available strictly *after* the inputs do.[8] For this reason, we mustn't compute so *eagerly* that we try consuming the output from a computation before the input to that computation is ready and before the computation itself has had time to finish. Just as we shouldn't consume information prematurely (before it's ripened), we also must avoid consuming it postmaturely (after it's rotted). To make matters even trickier, different paths through a circuit involve different delays, leading inevitably into the hazardous territory of "timing glitches". Much of the fun and cleverness of computational engineering lies in navigating skillfully between too soon and too late, as well as managing timing hazards.

Like hardware, imperative programming is tricky because programmers are exposed to the difficulty of staying in this Goldilocks zone—neither too eager nor too reluctant, but just right. Concurrency makes matters much worse, inviting programmers to create bugs in the form of nondeterminism, deadlock, and timing glitches. These bugs are of the insidious form known as "Heisenbugs", because they often change or even disappear when observed, only to return when one stops observing.

In contrast, functional programming shifts the burden of managing temporal consequences of computing's physicality from the use of a language to its implementation. *Nonstrict* functional programming makes this shift more thoroughly than strict functional programming by enabling useful decomposition of operations that require infinite compute time into simpler operations of the same nature [Hughes 1989].

An obvious conclusion is that we should always program in (nonstrict) purely functional languages. There's a big fly in that ointment, however. Although correct functional programming is much easier than correct hardware design, compiled software usually runs much less efficiently than custom hardware. There is an enormous gulf between the language's simple model of computation as pure functions and the specific nature of the physical substrate (with billions of transistors operating and communicating in parallel), let alone the analog nature of electricity as used to *continuously* charge and discharge capacitors.[9,10]

Hardware designers cope with timing glitches by requiring input signals to have enough of the right sort of stability (temporary constancy of boolean interpretation) to guarantee that correct results can always be found at predictable times. Even without glitches, any useful circuit must produce correct values not just momentarily but throughout an *interval* of time, for a simple and practical reason: physical measuring devices (including you and me) are incapable of paying attention at *exactly* the right moment and assimilating the information without at least a tiny amount of persistence.

[8]This simple statement about computation has some important exceptions, including constant and other non-hyperstrict functions. Moreover, sequential, digital computation is often described in general in terms of causal stream functions, with the understanding that the transformed streams are (and must be) evaluated lazily rather than eagerly.

[9]Imperative programming makes efficient mapping from program to hardware much more difficult by denying two fundamental aspects of the physical substrate:

- Computation occurs in parallel—not just within and among silicon chips but in the universe as a whole. Performance depends crucially on this physical parallelism, and the gap between sequential and parallel performance is expanding.
- Imperative programming is about *discrete* state change, so it has an ambient notion of "time" (usually chosen to model time-varying quantities) that is discrete rather than continuous. In contrast, physical time is effectively *continuous* at anywhere near the resolution we or our machines can inspect.

[10]The abstraction of discretely clocked computation can be implemented correctly only by considering what goes on during the time intervals *between* clock pulses—specifically moving electrons through wires, capacitors, and transistors. For instance, faster clocks generate faster results, but correctness can easily get lost in the rush due to unequal propagation delays. Conversely, correctness imposes constraints on clock rate (for a given circuit design), so a *precise* understanding of the semantics of timing can guide hardware engineers toward *temporally optimal* correct implementations.

We might consider limiting *all* signals to have stable intervals, but stability is needed only by the digital paradigm and so should not be imposed on the underlying analog notions. Instead, we can limit our own attention to the stable subset of signals and the signal functions that preserve stability.

This coping strategy of signal stability suggests a first definition of Φ: map from stable signals and stability-preserving transformations to plain signals and their transformations—simply by forgetting stability. We will formulate this choice in Section 7, after a few preliminaries.

4 THE COMMON VOCABULARY

A common vocabulary suffices to describe analog circuits, timing, discrete value computations, and correctness conditions, all related succinctly by homomorphisms. That vocabulary comes from cartesian categories together with a small collection of logic (boolean) operations. The reason for this choice is that category theory has proved to be an extraordinarily versatile and mathematically well-behaved foundation for composition in general and for computational notions in particular. Like other algebraic abstractions, category theory has a simple and useful notion of homomorphism (precise analogy) that naturally serves to define correctness.[11]

Categories provide the identity transformation ("morphism") and sequential composition:[12,13]

```
record Category {obj : Set o} (_⇾_ : obj → obj → Set ℓ) : Set (o ⊔ ℓ) where
  infixr 9 _∘_
  field
    id  : a ⇾ a
    _∘_ : (b ⇾ c) → (a ⇾ b) → (a ⇾ c)
open Category { … } public
```

This class has two primary parameters: the type of objects (*obj*), and the object-indexed type of morphisms (_⇾_), with the objects to be inferred automatically, as signified by the curly braces.[14]

Each *cartesian* category needs nullary and binary product objects, specified through a class parametrized by objects and not morphisms (to help automatic instance inference):

```
record Products (obj : Set o) : Set o where
  infixr 2 _×_
  field
    ⊤   : obj
    _×_ : obj → obj → obj
open Products { … } public
```

[11] As John Baez put it, "Every sufficiently good analogy is yearning to become a functor."
[12] The open line indicates that the record type acts as a type class with instances to be inferred automatically at their uses.
[13] You may be wondering why these classes omit laws. The category theory library used in this paper splits operations and laws into separate classes, because it is often possible and convenient to prove laws automatically via a homomorphism (functor) from the newly defined category to an existing *lawful* category, assuming that morphism equivalence in the new category is defined as equivalence modulo the chosen functor. This functor is often the primary denotation of the source category in terms of target category, in which case homomorphisms and denotational equivalence guarantee that we have no abstraction leak. Although the notion of functor is more often defined as between two *lawful* categories, it needn't be, since the functor laws involve only the operations and not the laws of a category. (The same reasoning holds for other algebraic abstractions and their associated notions of homomorphism.)
[14] The implicit parameters o and ℓ are universe levels, used in Agda to avoid logical inconsistencies.

The cartesian morphisms produce the nullary product, combine two morphisms with the same domain, and project pair components (plus several utility operations not shown here):[15]

```
record Cartesian {obj : Set o} ⦃ _ : Products obj ⦄
          (_⇾′_ : obj → obj → Set ℓ) ⦃ _ : Category _⇾′_ ⦄
    : Set (o ⊔ ℓ) where
  private infix 0 _⇾_; _⇾_ = _⇾′_
  infixr 7 _▵_
  field
    !     : a ⇾ ⊤
    _▵_  : (a ⇾ c) → (a ⇾ d) → (a ⇾ c × d)
    exl   : a × b ⇾ a
    exr   : a × b ⇾ b

  infixr 7 _⊗_
  _⊗_ : (a ⇾ c) → (b ⇾ d) → (a × b ⇾ c × d)
  f ⊗ g = (f ∘ exl) ▵ (g ∘ exr)
open Cartesian ⦃ … ⦄ public
```

Finally, we have generalized booleans and a few hardware-oriented operations (“gates”):

```
record Boolean {o} (obj : Set o) : Set o where
  field
    𝔹 : obj
open Boolean ⦃ … ⦄ public


record Logic {o} {obj : Set o} ⦃ _ : Products obj ⦄ ⦃ _ : Boolean obj ⦄
             {ℓ} (_⇾′_ : obj → obj → Set ℓ) : Set (o ⊔ ℓ) where
  private infix 0 _⇾_; _⇾_ = _⇾′_
  field
    false true : ⊤ ⇾ 𝔹
    not : 𝔹 ⇾ 𝔹
    nand nor xor : 𝔹 × 𝔹 ⇾ 𝔹
  and or : ⦃ _ : Category _⇾_ ⦄ → 𝔹 × 𝔹 ⇾ 𝔹
  and = not ∘ nand
  or  = not ∘ nor
open Logic ⦃ … ⦄ public
```

5 CONSTRAINING TYPES

Since digital computation depends on signal function *stability* (to be defined precisely in Section 7), let’s take a look at how to constrain types and the functions that operate on them. A natural

[15] An alternative formulation is to interpose *symmetric monoidal* categories [Baez and Stay 2010] between Category and Cartesian. In that case the tensoring operation ⊗ becomes a method rather than a singly defined operation (along with reassociating nested pairs and swapping pair elements). I’ve chosen to keep the categorical class hierarchy simpler in this paper.

definition of sets of values of some type A in type theory is as a *type-level predicate* on A, which is a function from A to propositions (types).[16]

Since we will want to aggregate constrained signals and compose constrained signal functions, a cartesian category is in order. In this category, an object is a type and a predicate constraining it, while a morphism is a function and a requirement that it map constrained inputs to constrained outputs:[17,18]

```
record PRED : Set (suc (m ⊔ ℓm)) where
  constructor pred
  field
    {ty} : Set m
    P : ty → Set ℓm
```

```
record _⇒_ (𝒜 ℬ : PRED) : Set (m ⊔ ℓm) where
  constructor mk⇒ ; open PRED
  field
    {f} : ty 𝒜 → ty ℬ
    imp : ∀ {u} → P 𝒜 u → P ℬ (f u)
```

If we think of predicates as type *subsets*, then nullary and binary cartesian products correspond, respectively, to the universal set over the unit type and cartesian products of sets:[19]

```
products : Products PRED
products = record
  { ⊤ = pred {⊤} λ { tt → ⊤ }
  ; _×_ = λ (pred {A} P) (pred {B} Q) → pred {A × B} λ (u , v) → P u × Q v }
```

For each of the operations of a cartesian category, the (implicit) value functions and (explicit) predicate functions are both defined by that same operation on proofs, giving the whole category a very elegant definition:

```
category : Category _⇒_
category = record
  { id  = mk⇒ id
  ; _∘_ = λ (mk⇒ g) (mk⇒ f) →
            mk⇒ (g ∘ f) }
```

```
cartesian : Cartesian _⇒_
cartesian = record
  { !   = mk⇒ !
  ; _▵_ = λ (mk⇒ f) (mk⇒ g) → mk⇒ (f ▵ g)
  ; exl = mk⇒ exl ; exr = mk⇒ exr }
```

This category takes care to separate types from predicates and functions from proofs. A simple functor recombines the parts, returning us to the usual category of functions:[20]

---

[16] Crucially, the choice of Set rather than Bool means that predicates needn't be decidable, and indeed the predicates we will define are mostly undecidable. Another example of the power of this choice is correct parsing of languages defined as predicates on strings [Elliott 2021],

[17] This notion of constrained types and constraint-respecting functions closely resembles refinement types [Freeman and Pfenning 1991]. This categorical formulation ensures that the input *proof* cannot influence the output *values* by keeping the value function and proofs separate, rather than injecting proof irrelevance into a function between existential types [Lovas and Pfenning 2009].

[18] In the definition of _⇒_, the "open PRED" enables conveniently accessing the ty and P fields of the domain and codomain PRED objects 𝒜 and ℬ. Since P is function-valued, it takes an additional argument beyond a PRED object.

[19] The Agda standard library provides all of the fundamentals used here [Agda Team 2022, Relation.Unary]:

- In PRED, the type of P can be written "Pred A ℓm".
- In _⇒_, the type of imp can be written "(P 𝒜 ⟨→⟩ P ℬ) f".
- In ⊤, the argument to pred can be written "U".
- In _×_, the argument to the result pred can be written "P ⟨×⟩ Q".

In order to make the explanation more self-contained here, I've not used these names.

[20] The _⇛_ here is the type of functors, i.e., morphisms in the category of categories. The constructor cat for the objects of that category takes the type of category objects as an implicit first argument (inferred here) and category morphisms as an explicit argument (here _⇒_ and _→_).

```
exists : cat _⇒_ ⇛ cat _→_
exists = mk⇛ (λ (pred P) → ∃ P) (λ (mk⇒ {f} imp) (x , Px) → f x , imp Px)
```

6 TIME INTERVALS

Time intervals are specified by lower and upper bounds, which may be finite or infinite. Rather than defining intervals all at once, we can form them as the intersection of two semi-infinite intervals, one given by a lower bound and another by an upper bound—having implicit upper and lower bounds of +∞ and -∞, respectively. This factoring corresponds to the notions of "setup" and "hold" timing analyses in hardware design, as well as coinciding with the useful "max-plus" and "min-plus" ("tropical") semirings [Golan 2005; Dolan 2013; Wilding 2015]. The semiring structure is more than just a pretty abstraction and proves crucial in extracting useful and flexible timing analyses, as demonstrated in Section 12.

The max-plus semiring is represented as an infinite or finite lower bound:

```
infix 9 _↑
data 𝕀↑ : Set where
  -∞ : 𝕀↑
  _↑ : 𝕋 → 𝕀↑

infix 4 _∈_
_∈_ : 𝕋 → 𝕀↑ → Set
t ∈ -∞ = ⊤
t ∈ x ↑ = x ≤ t
```

The semiring operations are usually referred to as "max" and "plus", but I'll call them "∩" and "✶" instead:

```
infixl 6 _∩_
_∩_ : Op₂ 𝕀↑
-∞ ∩ -∞ = -∞
-∞ ∩ y ↑ = y ↑
x ↑ ∩ -∞ = x ↑
x ↑ ∩ y ↑ = (x ⊔ y) ↑

infixl 7 _✶_
_✶_ : Op₂ 𝕀↑
-∞ ✶ -∞ = -∞
-∞ ✶ y ↑ = -∞
x ↑ ✶ -∞ = -∞
x ↑ ✶ y ↑ = (x + y) ↑
```

The reason for these names is that we can interpret a lower bound l as representing the *set* of all times greater than or equal to l (via ∈). The max and plus operations then correctly implement intersection and convolution of sets. The convolution of two sets A and B is the set of all values $x + y$ such that $x \in A$ and $y \in B$.[21] The max-plus semiring uses ∩ and ✶ as "addition" and "multiplication", and their identities -∞ and 0 ↑ as "zero" and "one". The semiring law proofs can be found in the source code for this paper. Because convolution is commutative, we have a *commutative* semiring, and we need this fact as well in Section 12.

The min-plus semiring is defined analogously:

```
infix 9 _↓
data 𝕀↓ : Set where
  _↓ : 𝕋 → 𝕀↓
  +∞ : 𝕀↓

infix 4 _∈_
_∈_ : 𝕋 → 𝕀↓ → Set
t ∈ x ↓ = t ≤ x
t ∈ +∞ = ⊤
```

[21] This operation (also called "Minkowski sum"), the usual notion of convolution over time and space, and language concatenation are all special cases of generalized convolution [Elliott 2019, 2021].

```
infixl 6 _∩_
_∩_ : Op₂ 𝕀↓
x ↓ ∩ y ↓ = (x ⊓ y) ↓
x ↓ ∩ +∞ = x ↓
+∞ ∩ y ↓ = y ↓
+∞ ∩ +∞ = +∞
```

```
infixl 7 _∗_
_∗_ : Op₂ 𝕀↓
x ↓ ∗ y ↓ = (x + y) ↓
x ↓ ∗ +∞ = +∞
+∞ ∗ y ↓ = +∞
+∞ ∗ +∞ = +∞
```

A doubly bounded interval is represented simply as a pair of singly bounded intervals, with the set denotation being the intersection of the two set interpretations. Importing the max-plus and min-plus modules as "↑" and "↓", respectively, we have

```
𝕀 : Set
𝕀 = 𝕀↑ × 𝕀↓
```

```
infix 4 _∈_
_∈_ : 𝕋 → 𝕀 → Set
t ∈ (l , u) = t ↑.∈ l × t ↓.∈ u
```

Since 𝕀↑ and 𝕀↓ are semirings, so is 𝕀, thanks to the general direct product construction. The semiring operations are thus equivalent to the following:

```
U : 𝕀
U = -∞ , +∞
```

```
infixl 6 _∩_
_∩_ : Op₂ 𝕀
(l₁ , u₁) ∩ (l₂ , u₂) = l₁ ↑.∩ l₂ , u₁ ↓.∩ u₂
```

```
{_} : 𝕋 → 𝕀
{ x } = x ↑ , x ↓
```

```
infixl 7 _∗_
_∗_ : Op₂ 𝕀
(l₁ , u₁) ∗ (l₂ , u₂) = l₁ ↑.∗ l₂ , u₁ ↓.∗ u₂
```

7 TIME, SIGNALS, AND STABILITY

Stability is a predicate on signals saying that the signal is constant throughout some interval:[22]

```
stable : Pred 𝕊 0ℓ
stable x̃ = ∃₂ λ (x̂ : 𝕀) (x : 𝔹) → ∀ {t : 𝕋} → t ∈ x̂ → x̃ t ≡ x
```

Reading this definition from left to right as a predicate on signals $\tilde{x}$: there exist an interval $\hat{x}$ and a value x such that for all times t in $\hat{x}$, the signal $\tilde{x}$ at time t equals x.[23]

Booleans in our category of stable multi-signals are then stable uni-signals:

```
boolean : Boolean PRED
boolean = record { 𝔹 = pred stable }
```

There's a subtlety here. With the definition of stable above, *every* signal is stable, because we can choose the interval $\hat{x}$ to be empty, thereby satisfying the property vacuously. What's the point, then? Because Agda's logic is *constructive*, proofs have useful computational content. A stable

[22]For a fully analog signal, we would say that the continuous signal has a constant *discrete interpretation* throughout the interval. In moving between low and high voltages, continuous signals pass through voltages that have no discrete interpretation. Digital computations must avoid paying attention to those non-semantic values in order to achieve correctness. The details are simple and shown in Section 16.

[23]Alternatively, we could eliminate x and say that $\tilde{x}\ t_1 \equiv \tilde{x}\ t_2$ whenever $t_1 \in \hat{x}$ and $t_2 \in \hat{x}$. Using the definition of stable above, however, leads to simpler compositionality and proofs.

$$\mathsf{stable}{\Rightarrow}_0 : \mathbb{T}^{\,0} \to (\mathbb{B}^{\,0} \to \mathbb{B}) \to (\mathbb{B}^{\,0} \Rightarrow \mathbb{B})$$
$$\mathsf{stable}{\Rightarrow}_0\ \delta\ h = \mathsf{mk}{\Rightarrow}\ \{f = \mathsf{analog}_0\ \delta\ h\}\ \lambda\ \{\ \mathsf{tt} \to \delta\ \hat{\S}^0\ \mathsf{tt}\ ,\ h\ \mathsf{tt}\ ,\ \lambda\ _ \to \mathsf{refl}\ \}$$

$$\mathsf{stable}{\Rightarrow}_1 : \mathbb{T} \to (\mathbb{B} \to \mathbb{B}) \to (\mathbb{B} \Rightarrow \mathbb{B})$$
$$\mathsf{stable}{\Rightarrow}_1\ \delta\ h = \mathsf{mk}{\Rightarrow}\ \{f = \mathsf{analog}_1\ \delta\ h\}\ \lambda\ (\hat{x}\ ,\ x\ ,\ P) \to \delta\ \hat{\S}\ \hat{x}\ ,\ h\ x\ ,\ \mathsf{cong}\ h \circ P \circ \in\!\ast$$

$$\mathsf{stable}{\Rightarrow}_2 : \mathbb{T}^{\,2} \to (\mathbb{B}^{\,2} \to \mathbb{B}) \to (\mathbb{B}^{\,2} \Rightarrow \mathbb{B})$$
$$\mathsf{stable}{\Rightarrow}_2\ \delta\ h = \mathsf{mk}{\Rightarrow}\ \{f = \mathsf{analog}_2\ \delta\ h\}\ \lambda\ ((\hat{x}_1\ ,\ x_1\ ,\ P_1)\ ,\ (\hat{x}_2\ ,\ x_2\ ,\ P_2)) \to$$
$$\quad \delta\ \hat{\S}^2\ (\hat{x}_1\ ,\ \hat{x}_2)\ ,\ h\ (x_1\ ,\ x_2)\ ,\ \mathsf{cong}_2{}'\ h \circ (P_1 \circ \in\!\ast \otimes P_2 \circ \in\!\ast) \circ \in\!\cap^e$$

Fig. 3. Stable logic templates

signal is more than a signal and an unsubstantiated claim of stability; it includes a *constructed* (computed) proof of that claim.

Existentially quantified types are dependent products, and universally quantified properties are dependent function types [Wadler 2015]. The quantifier $\exists_2$ is defined as nested existential quantification. Proofs (inhabitants) of stable $\tilde{x}$ are thus triples $(x\ ,\ \hat{x}\ ,\ p)$, where p is a curried function taking a time t (to be inferred) and a proof of $t \in \hat{x}$, and yielding a proof of $\tilde{x}\ t \equiv x$. (When an output interval is too small to be useful (including empty), then one can either supply a larger input interval or redesign the computation.)

Stability proofs thus contain the timing information needed for digital computation and proofs of its correctness. This definition of stable enables extracting a single bit (x). These extractions are combined with others by the generic Products PRED instance defined in Section 5. These multi-bit extractions contribute to the semantic functions h and k in Figure 1 via a broadly useful construction to be described in Section 9.

8 ANALOG GATES

Each *stable* analog logic primitive is a regular analog logic primitive whose use is constrained by stability (via PRED from Section 5). Templates for zero-, one-, and two-input stable gates are defined in Figure 3, where

$$_\hat{\S}^0_ : \mathbb{T}^{\,0} \to \mathbb{I}^{\,0} \to \mathbb{I} \qquad _\hat{\S}_ : \mathbb{T} \to \mathbb{I} \to \mathbb{I} \qquad _\hat{\S}^2_ : \mathbb{T}^{\,2} \to \mathbb{I}^{\,2} \to \mathbb{I}$$
$$\mathsf{tt}\ \hat{\S}^0\ \mathsf{tt} = \mathsf{U} \qquad \delta\ \hat{\S}\ \hat{x} = \{\ \delta\ \} \ast \hat{x} \qquad (\delta_1\ ,\ \delta_2)\ \hat{\S}^2\ (\hat{x}_1\ ,\ \hat{x}_2) = \delta_1\ \hat{\S}\ \hat{x}_1 \cap \delta_2\ \hat{\S}\ \hat{x}_2$$

and we've used two interval lemmas:

$$\in\!\ast : t \in \{\ \delta\ \} \ast p \to t - \delta \in p \qquad \in\!\cap^e : \forall\ \{t\} \to t \in p \cap q \to t \in p \times t \in q$$

Some remarks:

- Each $\mathsf{stable}{\Rightarrow}_i$ constructs a morphism in the PRED category (Section 5), using interval operations (Section 6) and the stable-signal definition of booleans (Section 7).
- Zero-input gates yield constant signals, which are perpetually stable (valid over the universal interval U).
- One-input gates apply their given function f to their discrete input *values* x, with output validity corresponding to the input validity but delayed by δ (expressed as convolution with a singleton interval).
- Two-input gates produce a signal that is valid over the *intersection* of delayed versions of the input valid intervals. This intersection is necessary for correctness (and thus for proof), because validity of the output depends on *simultaneous* validity of the delayed inputs.

- In each $\text{stable}{\Rightarrow}_i$ definition, the analog signal function could be inferred correctly and automatically from correctness proofs but is given explicitly for clarity.

Our stable analog gates simply instantiate the templates in Figure 3:

```
logic : Logic _⇒_
logic = record { false = stable⇒₀ δ-false false
               ; true  = stable⇒₀ δ-false true
               ; not   = stable⇒₁ δ-not   not
               ; nand  = stable⇒₂ δ-nand  nand
               ; nor   = stable⇒₂ δ-nor   nor
               ; xor   = stable⇒₂ δ-xor   xor }
```

In this section, we have defined gates that transform *stable* analog signals. Next, we will want to incorporate those stabilized analog gates into digital computations as shown in Figure 1 and compose the resulting digital gates into digital circuits of arbitrary sophistication, maintaining correctness at every step. We'll address composition next and then return to *digital* gates in Section 10.

9 COMPOSITIONALLY CORRECT ENGINEERING

Sophisticated engineering artifacts are not usually built monolithically but rather by *composing* simpler artifacts. The necessity for this practice stems from our limited ability to manage complexity [Dijkstra 1972]. Quality of the results depends crucially on the sort of composition being done. To ensure reliability, composition must satisfy the following criteria:

- Every component (primitive and composite) must clearly specify the goal it promises to fulfill.
- The complexity of each specification must remain within human ability to understand dependably and accurately, regardless of the amount of composition happening. Since the *operational* complexity (detailed steps of execution) of composite computations is the *sum* of the operational complexities of its components, specifications must have a fundamentally different nature from the operational recipes that promise to satisfy them. In particular, for specifications (clarity of goal) to be scalably beneficial, they must be *elegant*.[24]
- Composition must *preserve* reliability, i.e., compositions are as reliable as each component. Otherwise, likelihood of error grows at every composition step. Consequently, even if we are aiming at various approximations of correctness—such as *nearly*, *probably*, or *usually* correct—we can only scale up if we have *perfect* reliability at each stage. Otherwise, repeated composition drives such approximations to their opposites: *grossly*, *probably*, and *usually* incorrect. Formal, machine-checked proof is the only way I know of to preserve reliability in the face of repeated composition.[25]

Compositionally correct computing as pictured in in Figure 1 is a standard construction in category theory, namely a *comma category* [Mac Lane 1998; Riehl 2016]. The two functors (Φ and φ) are parameters of this category and so will be held fixed across all parts of a composition (though we will evolve variations of Φ and φ for increasingly powerful analysis and circuit synthesis).

[24]Note that the motivation for elegance here is practical, not aesthetic. While somewhat subjective, elegance can be formulated as simplicity of specification, but we must define what concepts are allowed to be assumed in such a specification. As Murray Gell-Mann [2009] expressed this criterion, "A theory appears beautiful or elegant [...] when it's simple; in other words when it can be expressed very concisely in terms of mathematics that we've already learned for some other reasons".

[25]This level of reliability does not come for free. Rather, the formal proof requirement forces engineers to discover and fix mistakes before composing (when it's relatively easy to do so) rather than after (when it's more difficult).

These functors map from each of two categories C_1 and C_2 to another category C_0 in which the commutative diagram lives.

If we divide up the constellation pictured in Figure 1 in just the right way, we can see that it forms a cartesian category. This fact enables us to construct correct artifacts from smaller correct artifacts in a flexible, familiar, and well-behaved way. The domain and codomain *objects* for a comma morphism are the C_0 morphisms h and k, each bundled with the C_1 and C_2 objects that map *to* the domain and codomain of those morphisms via functors Φ and φ (renamed to "Φ_1" and "Φ_2" for the general construction below). The morphisms in the constructed category comprise the rest of the constellation: the morphisms $\tilde{f}$ and f (from C_1 and C_2, respectively) and the commutativity proof $\circlearrowleft$ (in C_0). These objects and morphisms are tidily expressed in an elegant, modern dependently typed language like Agda:[26]

```
record COM : Set (ℓᵒ ⊔ ℓᵐ) where
  constructor com
  field
    { τ₁ } : obj₁
    { τ₂ } : obj₂
    h : Φ₁ᵒ τ₁ ⇾₀ Φ₂ᵒ τ₂
```

```
record _↬_ (a b : COM) : Set (ℓᵐ ⊔ q₀) where
  constructor mk↬ ; open COM
  field
    f₁ : τ₁ a ⇾₁ τ₁ b
    f₂ : τ₂ a ⇾₂ τ₂ b
    ⟲ : h b ∘ Φ₁ᵐ f₁ ≈ Φ₂ᵐ f₂ ∘ h a
```

The compositional machinery of correct engineering lives in the definitions of the operations for constructing and combining morphisms in a comma category. To see how this general machinery works, consider the following two diagrams:

$$\begin{array}{ccc} \Phi_1^o\,\rho_1 & \xrightarrow{\Phi_1^m\,f_1} & \Phi_1^o\,\sigma_1 \\ {\scriptstyle h}\downarrow & \circlearrowleft^f & \downarrow{\scriptstyle k} \\ \Phi_2^o\,\rho_2 & \xrightarrow{\Phi_2^m\,f_2} & \Phi_2^o\,\sigma_2 \end{array} \qquad \begin{array}{ccc} \Phi_1^o\,\sigma_1 & \xrightarrow{\Phi_1^m\,g_1} & \Phi_1^o\,\tau_1 \\ {\scriptstyle k}\downarrow & \circlearrowleft^g & \downarrow{\scriptstyle i} \\ \Phi_2^o\,\sigma_2 & \xrightarrow{\Phi_2^m\,g_2} & \Phi_2^o\,\tau_2 \end{array}$$

Combine sequentially, erase the common edge, and apply functoriality:[27]

$$\begin{array}{ccccc} \Phi_1^o\,\rho_1 & \xrightarrow{\Phi_1^m\,f_1} & \Phi_1^o\,\sigma_1 & \xrightarrow{\Phi_1^m\,g_1} & \Phi_1^o\,\tau_1 \\ {\scriptstyle h}\downarrow & \circlearrowleft^f & \downarrow{\scriptstyle k} & \circlearrowleft^g & \downarrow{\scriptstyle i} \\ \Phi_2^o\,\rho_2 & \xrightarrow{\Phi_2^m\,f_2} & \Phi_2^o\,\sigma_2 & \xrightarrow{\Phi_2^m\,g_2} & \Phi_2^o\,\tau_2 \end{array} \qquad \begin{array}{ccc} \Phi_1^o\,\rho_1 & \xrightarrow{\Phi_1^m\,(g_1\circ f_1)} & \Phi_1^o\,\tau_1 \\ {\scriptstyle h}\downarrow & \circlearrowleft & \downarrow{\scriptstyle i} \\ \Phi_2^o\,\rho_2 & \xrightarrow{\Phi_2^m\,(g_2\circ f_2)} & \Phi_2^o\,\tau_2 \end{array}$$

Commutativity of the composite diagram follows from commutativity of the component diagrams and functoriality.

[26] I don't expect you to digest the details here, but rather get a sense of how simple the definitions are. In case you are interested, here are a few details:

- obj_i is the type of objects in category C_i.
- $\rightarrowtriangle_i$ is the type of morphisms in C_i.
- Φ_i^o and Φ_i^m are the object and morphism aspects of the functor Φ_i.
- $\approx$ is morphism equivalence in the category C_0.

[27] Functoriality gives us $\Phi_i^m\ g_i \circ \Phi_i^m\ f_i \approx \Phi_i^m\ (g_i \circ f_i)$.

Likewise, consider the following digital computations:

$$\begin{array}{ccc} \Phi^o_1\ \rho_1 & \xrightarrow{\Phi^m_1\ f_1} & \Phi^o_1\ \sigma_1 \\ {\scriptstyle h}\downarrow & \circlearrowleft^f & \downarrow{\scriptstyle k} \\ \Phi^o_2\ \rho_2 & \xrightarrow{\Phi^m_2\ f_2} & \Phi^o_2\ \sigma_2 \end{array} \qquad \begin{array}{ccc} \Phi^o_1\ \tau_1 & \xrightarrow{\Phi^m_1\ g_1} & \Phi^o_1\ \upsilon_1 \\ {\scriptstyle i}\downarrow & \circlearrowleft^g & \downarrow{\scriptstyle j} \\ \Phi^o_2\ \tau_2 & \xrightarrow{\Phi^m_2\ g_2} & \Phi^o_2\ \upsilon_2 \end{array}$$

Combine in parallel (“tensoring”), and apply monoidal functoriality:

$$\begin{array}{ccc} \Phi^o_1\ (\rho_1 \times \tau_1) & \xrightarrow{\Phi^m_1\ (f_1 \otimes g_1)} & \Phi^o_1\ (\sigma_1 \times \upsilon_1) \\ {\scriptstyle h \otimes i}\downarrow & \circlearrowleft & \downarrow{\scriptstyle k \otimes j} \\ \Phi^o_2\ (\rho_2 \times \tau_2) & \xrightarrow{\Phi^m_2\ (f_2 \otimes g_2)} & \Phi^o_2\ (\sigma_2 \times \upsilon_2) \end{array}$$

Again, commutativity of the composite diagram follows from commutativity of the component diagrams and monoidal functoriality.

We now have the primary operations of a monoidal category. To complete the operations of a cartesian category, we need the identity, the terminal morphism (discarding), pair projections, and duplication.[28] The comma version of each operation follows the pattern illustrated above for sequential and parallel composition: apply that same operation to the C_1 and C_2 morphisms, combine the comma objects (vertical edges) in simple and inevitable ways, and prove commutativity via (cartesian) functoriality and commutativity of the component diagrams. The *only* vocabulary and machinery we need that is specific to digital computation is our handful of Logic operations (Section 4), to which we turn our attention next.

10 DIGITAL GATES

Let’s now briefly review where we are and where we’re going. Section 7 defined signal stability and embedded that property in the PRED category of Section 5 to yield a cartesian category of stable multi-signals and their stability-preserving transformations. Section 2 defined a standard collection of analog logic primitives, later used in Section 8 to define *stable* versions of those primitives, tracking the relationship between input and output timing needed for correct digital use. Section 9 presented a general framework for compositionally correct engineering as a standard construction from category theory that neatly and compositionally embodies our guiding specification (pictured in Figure 1) and. We will now fill in the missing piece in this story, which is how to wrap the stable analog logic primitives from Section 8 so that the resulting “digital gates” can be composed as in Section 9 to form correct, sophisticated digital computations.

So far I have hinted at the nature of the mappings Φ and φ in Figure 1 without nailing them down. The general compositionally correct machinery of Section 9 only requires Φ and φ to be cartesian functors. To address digital composition, we’ll need to define these functors. For now let’s take Φ to be exists from Section 5 and φ to be the identity functor:

```
open Comma exists id
```

[28] For reversible computation, add reassociation and swap instead.

Recall from Section 9 that the domain and codomain objects of a comma morphism are the morphisms h and k from Figure 1 together with the objects that the functors Φ and φ map to the domain and codomain of h and k. Comma categories come equipped with a notion of categorical products that takes care of combining smaller value extractors into larger ones. The last remaining piece is the definition of boolean comma objects, which simply extract a single boolean value from a stable signal:

```
boolean : Boolean COM
boolean = record { 𝔹 = com {τ₁ = 𝔹} {τ₂ = 𝔹} λ (x̃ , x̂ , x , P) → x }
```

The parameter choice of $\tau_1 = \mathbb{B}$ is resolved to come from the Boolean PRED instance defined in Section 7, i.e., pred stable, while $\tau_2 = \mathbb{B}$ comes from the category of types and functions and so refers to boolean values. This pattern of object choice is typical for comma constructions and plays out with nullary and binary products as well.

Finally, the digital logic primitives for our comma category are defined quite simply. For each primitive logic operation, the digital (comma) version is defined via the same operation in the source categories C_1 (stable signal functions) and C_2 (functions) and a trivial correctness (commutativity) proof (reflexivity):

```
logic : Logic _↬_
logic = record { false = mk↬ false false λ _ → refl
               ; true  = mk↬ true  true  λ _ → refl
               ; not   = mk↬ not   not   λ _ → refl
               ; nand  = mk↬ nand  nand  λ _ → refl
               ; nor   = mk↬ nor   nor   λ _ → refl
               ; xor   = mk↬ xor   xor   λ _ → refl }
```

11 EXTRACTING TIMING INFORMATION

We've now accomplished our goal of formulating digital computation as in Figure 1. Anything we can express via logic gates and the language of cartesian categories (including sequential and parallel composition, projection, duplication, and forgetting) yields a digital computation, including the underlying analog computation, the corresponding discrete value function being implemented, correctness (commutativity) proof, and suitable value extractors. Although hardware designers don't typically use this vocabulary, they could. Alternatively, one could automatically translate from a typed lambda-calculus to categorical language [Elliott 2017].

There is more information that we can mine from our constructions, besides knowing what analog computation to run and what discrete function it implements correctly. We'll also want to know exactly how output timing depends on input timing and value. Fortunately, we can do so easily. Use the same functors and alter the Boolean COM instance in Section 10 to keep timing as well as values:

```
open Comma exists id

boolean : Boolean COM
boolean = record { 𝔹 = com {τ₁ = 𝔹} {τ₂ = 𝕀 × 𝔹} λ (x̃ , x̂ , x , P) → x̂ , x }
```

Here, we've changed τ_2 from $\mathbb{B}$ to $\mathbb{I} \times \mathbb{B}$, and changed the extractor to yield the timing $\hat{x}$ in addition to value x. The Logic instance adapts without much fuss:[29]

```
logic : Logic _↬_
logic = record { false = mk↬ false (δ-false §̂⁰_ ▵ false )               λ _ → refl
               ; true  = mk↬ true  (δ-true  §̂⁰_ ▵ true  )               λ _ → refl
               ; not   = mk↬ not   (δ-not   §̂_  ⊗ not   )               λ _ → refl
               ; nand  = mk↬ nand ( (δ-nand §̂²_ ⊗ nand ) ∘ transpose ) λ _ → refl
               ; nor   = mk↬ nor  ( (δ-nor  §̂²_ ⊗ nor  ) ∘ transpose ) λ _ → refl
               ; xor   = mk↬ xor  ( (δ-xor  §̂²_ ⊗ xor  ) ∘ transpose ) λ _ → refl }
```

With these changes, the extractors h and k and the bottom morphism $\varphi\ f$ in Figure 1 now produce and transform values and timing together.

Although we have made timing and its transformation available in the data of the comma objects and morphisms, we have lost the important fact that the output *value* depends *only* on the input value and not on its *timing*. Computation timing often depends on input value, especially for software (e.g., the number of loop iterations or depth of recursion is often input-dependent). Hardware designers, however, are highly motivated to design most aspects of a computation to take a fixed amount of time, independent of input values. When they're able to do so, they can use simple and efficient synchronization mechanisms, particularly a shared clock running at a constant frequency.

As a further refinement, therefore, let's consider how we might extract both values and timing while guaranteeing that each is independent of each other. For an arbitrary function $f : A_1 \times A_2 \to B_1 \times B_2$, there is a simple, general, and precise way say that the first output (B_1) depends only the first input (A_1) and the second output (B_2) depends only the second input (A_2), namely that there are functions $f_1 : A_1 \to B_1$ and $f_2 : A_2 \to B_2$ such that $f \doteq f_1 \otimes f_2$. In other words, f results from $\otimes$. Since Figure 1 already involves generating our value or value-and-timing function via a functor φ, perhaps there's a way to get φ (currently the identity) to apply $\otimes$ for us. A tricky point is that $\otimes$ takes a *pair* of functions while functors take just one morphism. Luckily, we can again appeal to a simple standard construction from category theory. The *product* of two categories $\mathcal{A}$ and $\mathcal{B}$ is a category in which an object comprises a pair of objects (one from $\mathcal{A}$ and one from $\mathcal{B}$), and a morphism comprises a pair of morphisms (again one from $\mathcal{A}$ and one from $\mathcal{B}$). Every categorical operation treats these object and morphism pairs with exactly the independence that we want for values and timing.

To apply this idea to digital circuit construction, we can make minor changes to the Boolean and Logic instances above. The most important change is to φ, which now maps from a product category:

```
open Comma exists unsquare
```

The functor unsquare maps the pair of objects (types) A_1 and A_2 to the single object $A_1 \times A_2$ and the pair of morphisms (functions) f_1 and f_2 to the single morphism $f_1 \otimes f_2$:

[29] Product transposition in cartesian (or monoidal) categories has the following signature:

```
transpose : (a × b) × (c × d) ⇾ (a × c) × (b × d)
```

```
unsquare : cat _⇾²_ ⇛ cat _⇾_
unsquare = mk⇛ (λ (A , B) → A × B) (λ (f , g) → f ⊗ g)
```

The other changes are subtler and follow inevitably from the functor change. Thanks to unsquare, the resulting definitions are tidier than before we disentangled values and timings:

```
boolean : Boolean COM
boolean = record { 𝔹 = com {τ₁ = 𝔹} {τ₂ = 𝕀 , 𝔹} λ (x̃ , x̂ , x , P) → x̂ , x }

logic : Logic _↬_
logic = record { false  = mk↬ false  (δ-false  §̂⁰_ , false ) λ _ → refl
               ; true   = mk↬ true   (δ-true   §̂⁰_ , true  ) λ _ → refl
               ; not    = mk↬ not    (δ-not    §̂_  , not   ) λ _ → refl
               ; nand   = mk↬ nand   (δ-nand   §̂²_ , nand  ) λ _ → refl
               ; nor    = mk↬ nor    (δ-nor    §̂²_ , nor   ) λ _ → refl
               ; xor    = mk↬ xor    (δ-xor    §̂²_ , xor   ) λ _ → refl }
```

The commutative diagram itself is exactly the same as the previous version[30], but now the data in the comma objects and morphisms manifest the independence of value and timing, leaving the functor unsquare to forget that independence. What makes this change useful is that the construction guarantees independence *and* gives us access to the independent value and timing functions. We can now verify (often automatically) that the function coincides with our intentions, and we can start exploring how to represent the data-independent timing function in a form that allows us to choose a static clock frequency at which the analog circuit provably calculates the desired function.

12 PRACTICAL (AND CORRECT) TIMING ANALYSIS

Now that we have separated timing computation from value and signal computation, we can run these computations independently. Doing so is not as useful as we might like, however, for a few reasons:

- Timing calculations are functions over "timings", i.e., aggregates (nested pairs) of intervals. If we feed input timing $\hat{u}$ into a timing function to yield output timing $\hat{v}$, then the correctness (commutativity) property guarantees that if we hold input steady during the interval $\hat{u}$, then the output will be held steady and correct during the interval $\hat{v}$. If we want to operate the circuit again with different input timing, we will have to recalculate output timing.
- If any component interval of an output (multi-)timing is empty, we can try again with larger input timing intervals, but we're hunting in the dark—guessing what combination of changes to the input timing might help.
- We can only calculate timings forward, i.e., from input to output. It is also useful to compute minimal requirements on input timing required to ensure meeting output timing goals.

Each of these practical limitations results from using *functions* to relate input and output timings, in that functions are by nature black boxes and so cannot be analyzed, optimized, inverted, etc. We are thus motivated to investigate whether the particular timing functions generated by our comma construction have properties enabling the sort of analysis we want to perform. The only operations we are using to construct those functions are from the following interfaces:

[30]Since the commutative diagram involves only the target categories of Φ and φ, it remains unchanged whenever we change only the source category of Φ or φ.

- Logic:
  - Zero-input: the constant function yielding the universal interval U (from -∞ to +∞).
  - One-input: convolution with a constant interval.
  - Two-input: convolutions with constant intervals, followed by intersection.
- Category:
  - The identity function.
  - Sequential composition.
- Cartesian:
  - Discarding information, yielding the zero-dimensional multi-interval (tt).
  - Parallel composition with input duplication.
  - Left and right projections.

There is one more important source of algebraic regularity we can exploit: *intervals form a commutative semiring* (Section 6). By combining intervals into higher-dimensional aggregates, we are forming a (left) *semimodule*, which is like a vector space but with the requirement on scalars relaxed from a field to a semiring (not requiring multiplicative or additive inverses, which intervals do not have). Could we be so lucky that timings are always semimodule homomorphisms, i.e., *linear maps* in our semimodule of higher-dimensional intervals? Let's see:

- Zero in this semiring is U, so a zero-input gate timing is a constant-zero function—the only kind of constant linear function.
- Multiplication is convolution, so a one-input gate timing is multiplying by a constant—exactly the linear scalar-to-scalar functions.
- Addition is intersection, so a two-input gate timing is $\lambda\,(x\,,\,y) \rightarrow f_1\;x + f_2\;y$ for linear functions f_1 and f_2—exactly characterizing linear functions from products.
- Linear functions form a cartesian category, i.e., linearity is closed under the Category and Cartesian operations.

Yes, we are lucky indeed. *Timing is linear.*

Linear functions form not just a cartesian category but a *biproduct* category, meaning that they have categorical coproducts as well as products and that the two coincide. Biproducts give linear algebra its essential nature, including the ability to decompose morphisms (linear functions) along the codomain as well as domain [Macedo and Oliveira 2013]. This double-sided decomposition together with the fact that linear functions between scalars are exactly multiplications by a constant give rise to the representation we call "matrices", though in a more compositional manner [Santos and Oliveira 2020]. In particular, the form of two-input gate timing ($\lambda\,(x\,,\,y) \rightarrow f_1\;x + f_2\;y$) is the cocartesian operation sometimes called "join" and written "$[\,f_1\,,\,f_2\,]$" or "$f_1 \triangledown f_2$", being dual to "fork", which is sometime written "$\langle\,f_1\,,f_2\,\rangle$" or "$f_1 \vartriangle f_2$" (as in this paper, following Gibbons [2002]). Likewise, zero-input gates are the initial morphism, often written "¡"—being dual to the terminal morphism, written "!". These compositional "matrices" provide the static (input-independent) timing analysis we want. Each matrix entry is an interval that expresses precisely how the timing of one output bit depends on the timing of one input bit.

We can formally prove and then exploit linearity of timing in the same way we proved and used *separability* of timing and value computations in Section 11, by augmenting the lower functor φ in Figure 1. Just as separability is expressed through a functor that *combines* two independent functions into one function—forgetting its separability—we can express linearity through a functor that consumes linear functions and forgets their linearity.

This new functor maps from the biproduct category of linear maps over some semiring to the category of functions. The source objects are semimodules over a fixed semiring (the parameter

of the category), and the morphisms are linear functions. The forgetful functor maps each semimodule to its carrier type (forgetting the operations and corresponding proofs of the semimodule laws) and each linear function to a function (forgetting the linearity proof).

The linear category is a subcategory of regular functions built on a broadly useful notion of *compositional properties* of morphisms. Here, the property is linearity, and compositionality is with respect to the operations of Category, Cartesian, and Cocartesian. To give the flavor of this category, linearity is defined as follows:[31]

```
record linear (f^M : M →^M N) : Set (ℓr ⊔ r) where
  private f = f^M •_
  field
    cong : u ≈ v → f u ≈ f v
    0-H  : f 0 ≈ 0
    +-H  : f (u + v) ≈ f u + f v
    ·-H  : f (s · u) ≈ s · f u
```

These four properties of a semimodule function f^M are (a) mapping equivalent arguments to equivalent results ("congruence"), (b) mapping the zero vector to the zero vector, (c) distributing over vector addition, and (d) distributing over vector scaling. Here "vector" refers to (bi)product-aggregated scalars. Each law relates an operation interpreted in one semimodule to the same operation in the other. The compositionality proofs needed to form this sort of subcategory work out rather simply, e.g.,[32]

```
cat^P : Category^P linear
cat^P = record
  { id^P = λ {M} → let instance _ = M in
      record { cong = id ; 0-H = refl ; +-H = refl ; ·-H = refl }
  ; _∘^P_ = λ {M N P} g° f° → let instance _ = P; _ = f°; _ = g° in
      record { cong = cong ∘ cong
             ; 0-H  = cong 0-H ⨾ 0-H
             ; +-H  = cong +-H ⨾ +-H
             ; ·-H  = cong ·-H ⨾ ·-H }
  }

cart^P : Cartesian^P linear
cart^P = record
  { !^P = _ -- inferred
  ; _△^P_ = λ f° g° → let instance _ = f° ; _ = g° in
      record { cong = cong △ cong
             ; 0-H  = 0-H , 0-H
```

[31]There are really two subcategories here. In the first subcategory, called "$\longrightarrow^{\mathrm{M}}$" here, the objects are semimodules, and the morphisms are arbitrary functions over semimodules. The linear predicate is defined over these morphisms, and so can assume the existence of semimodule operations ≈, **0**, +, and · for each semimodule (M and N here). The application operator used in the definition of linear is defined by dropping a constructor and applying the underlying function: $\mathsf{mk}^{\mathrm{M}}\, f \bullet u = f\ u$. In the second category—defined via the compositionality of linearity—the objects are still semimodules, but the morphisms are restricted to *linear* functions.

[32]The infix operator "⨾" refers to transitivity of morphism equivalence.

```
          ; +-H   = +-H , +-H
          ; ·-H   = ·-H , ·-H }
  ; exlᴾ = λ {M N} → let instance _ = M in
      record { cong = exl ; 0-H = refl ; +-H = refl ; ·-H = refl }
  ; exrᴾ = λ {M N} → let instance _ = N in
      record { cong = exr ; 0-H = refl ; +-H = refl ; ·-H = refl }
  }
```

The other proofs (in the paper's source code) are for Cocartesian and for scaling. Only the last of these operations needs commutativity of multiplication (convolution in the case of intervals). The category of linear functions over semimodules is defined via a standard recipe in terms of compositional properties (here, linearity). This recipe also provides the forgetful functor from linear functions to regular functions as needed to adjust φ.

13 EXAMPLES

Let's now see how correct, linear timing analysis plays out in practice. As a trivial example, consider a solitary exclusive-or gate, yielding a timing matrix with one row (for one output) and two columns (for two inputs):

```
_ : matrix 2 1 (timing xor) ≡ (1 // 4 , 1 // 4)
_ = refl
```

Some explanation:

- The utility function timing extracts (linear) timing functions from the second (separated) representation in Section 11:

  ```
  timing : ∀ {a b : COM} → (a ↬ b) → (_ → _)
  timing (mkᵐ _ (f , _) _) = f
  ```

- The matrix function exploits the linearity of the timing function to extract a *matrix* of intervals, given domain and codomain dimensions and a timing function.
- The utility operator _//_ generates a singleton interval for a given fraction:

  ```
  infix 7 _//_
  _//_ : (n : ℤ) (d : ℕ) { _ : NonZero d } → 𝕀
  n // d = { n / d }
  ```

- The right-hand side of the equality was filled in by the Agda type-checker (as needed to make the trivial reflexivity proof valid) and then tidied manually.

As a somewhat more interesting example, consider an *and* "gate", defined as not ∘ nand (Section 4):

```
_ : matrix 2 1 (timing and) ≡ (3 // 10 , 3 // 10)
_ = refl
```

The delay of 3/10 comes from adding the gate delays of 1/5 for nand and 1/10 for not (Section 2). Timing for *or* gates works out the same as for *and* gates.

Next, let's define a *half adder*, which adds two bits to yield a two-bit binary number.[33] The standard recipe is to combine *exclusive or* to compute the less significant bit together with *and* to compute the more significant bit:

```
ha : 𝔹 × 𝔹 ↬ 𝔹 × 𝔹
ha = xor △ and
```

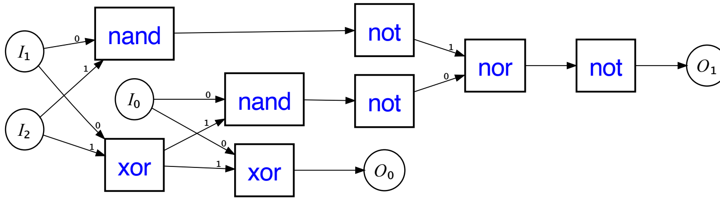
The corresponding 2-by-2 timing matrix corresponds to vertically stacking the timings for xor and and shown above:

```
_ : matrix 2 2 (timing ha) ≡ ( (1 // 4 , 1 // 4)
                             , (3 // 10 , 3 // 10) )
_ = refl
```

As a final example, consider a *full adder*, taking *two* addend bits and a "carry-in" and yielding a two-bit binary sum. The construction uses two half adders and an or gate:

```
fa : (𝔹 × 𝔹) × 𝔹 ↬ 𝔹 × 𝔹
fa = (id ⊗ or) ∘ assocʳ ∘ (ha ⊗ id) ∘ assocˡ ∘ (id ⊗ ha) ∘ assocʳ
```

This cryptic, point-free definition can be visualized graphically as follows (generated from the same formula in another category, as in Elliott [2017]), with boxes for primitive gates and circles for inputs and outputs:



Timing analysis:

```
_ : matrix (2 + 1) 2 (timing fa) ≡ ( ((1 // 4 , 1 // 2) , 1 // 2)
                                   , ((3 // 5 , ((17 / 20) ↑ , (3 / 5) ↓)) , ((17 / 20) ↑ , (3 / 5) ↓)) )
_ = refl
```

Note that each of the input bits I_1 and I_2 has two distinct paths of influence on the output bit O_1. Consequently, the corresponding timing matrix entries involve non-singleton intervals. Interestingly, the "lower" bound (from the MinPlus semiring) *exceeds* the "upper" bound (from the MaxPlus semiring), i.e., the interval's width is *negative* ($3/5 - 17/20 = -1/4$) as a result of intersecting the intervals 3 // 5 and 17 // 20. This information tells us that the second and third input bits must be stable for at least 1/4 ns in order for the second output bit to be stable for any (positive) amount of time. Recall that the linear timing function forms linear combinations, i.e., intersections of Minkowski sums of intervals (corresponding abstractly to dot products in the interval semiring defined in Section 6).

[33]The output can be thought of as one sum bit and one carry-out bit. Likewise, the input can be thought of as a *single* addend bit (thus "half") and a carry-in bit.

14 CIRCUITS

The bulk of our exploration has been about the continuous-to-discrete transition posed in Section 1. Let's now consider an aspect of the concrete-to-abstract transition. While we won't be able to bring the full physicality of circuits into the formal specification, we can bring some of their *tangibility*, i.e., susceptibility to analysis and manipulation, including the ability to synthesize physical artifacts. One tangible form is an inductive representation that directly captures the compositional vocabulary used to describe computations, namely the Category, Cartesian, and Logic operations. This vocabulary can then be interpreted into *any* type of morphisms ↠ that supports those three interfaces:[34]

```
data _‘→_ : obj → obj → Set o where         ⟦_⟧ : (a ‘→ b) → (a ↠ b)
  ‘id  : a ‘→ a                             ⟦ ‘id    ⟧ = id
  _‘∘_ : (b ‘→ c) → (a ‘→ b) → (a ‘→ c)     ⟦ g ‘∘ f ⟧ = ⟦ g ⟧ ∘ ⟦ f ⟧
  ‘!   : a ‘→ ⊤                             ⟦ ‘!     ⟧ = !
  _‘△_ : (a ‘→ c) → (a ‘→ d) → (a ‘→ c × d) ⟦ f ‘△ g ⟧ = ⟦ f ⟧ △ ⟦ g ⟧
  ‘exl : a × b ‘→ a                         ⟦ ‘exl   ⟧ = exl
  ‘exr : a × b ‘→ b                         ⟦ ‘exr   ⟧ = exr
  ‘false ‘true : ⊤ ‘→ 𝔹                     ⟦ ‘false ⟧ = false ; ⟦ ‘true ⟧ = true
  ‘not : 𝔹 ‘→ 𝔹                             ⟦ ‘not   ⟧ = not
  ‘nand ‘nor ‘xor : 𝔹 × 𝔹 ‘→ 𝔹              ⟦ ‘nand  ⟧ = nand ; ⟦ ‘nor ⟧ = nor ; ⟦ ‘xor ⟧ = xor
```

The semantic function ⟦_⟧ is homomorphic (over Category, Cartesian, and Logic) by construction, so the homomorphism proofs are all trivial. This ⟦_⟧ functor can then be pre-composed into Φ so that the comma construction contains the tangible representation, while the commutative diagram continues to relate the denotations.[35]

15 RELATED WORK

There has been a tremendous amount of fruitful research done in functional and relational modeling of hardware computation, including formal specification and verification [Sheeran 2005]. Some work has applied dependent types to describe hardware more precisely and to address correctness [Hanna and Daeche 1992; Flor et al. 2018]. I do not know of such work, however, that uses continuous time as a starting point and computes circuit timing with correctness proofs. Beginning with a temporally discrete model, either in terms of streams [Sheeran 1984; O'Donnell 1987; Matthews et al. 1998; Townsend et al. 2015; Zhai et al. 2015] or state and sequencing [Procter et al. 2015; Harrison et al. 2021], prevents us from even asking the essential question of when values can be correctly extracted from the underlying temporally continuous signals.

Ghica and Jung [2016]; Ghica et al. [2017] have also structured circuit descriptions in categorical terms, though in a discrete, syntactic, and operational style, using algebraic properties as specification rather than as lemmas that follow from a denotational specification (the stream or signal functions being computed). I used categories as well for specifying hardware (and other tasks), via a compiler plugin, automatically translating from Haskell into categorical combinators [Elliott 2017], but did not address timing or prove correctness.

[34] The module in which this "syntactic" (inductive) category is defined is parametrized by exactly such objects and morphisms.

[35] Unfortunately, this functor encounters a technical difficulty. In the underlying category theory library, CAT (the category of categories) is parametrized by a universe level for objects and another for morphisms, and all functors (CAT morphisms) stay within those levels. The semantic function ⟦_⟧, however, moves between levels. More thought is needed either to loosen the library's restriction or to work more creatively within it.

The pioneering circuit retiming work of Leiserson and Saxe [1991] addressed "timely computation" head on, performing calculations similar to those suggested by the development above, but phrased as a graph (rather than semantic) problem that appears to be non-compositional and thus awkward to formalize and use. Sheeran [1988] systematized retiming compositionally in terms of relations in Ruby, though with discrete rather than continuous time.

16 CONCLUSIONS AND FUTURE WORK

Fundamentally, computational science and engineering—in their purer and more applied forms—are about *instructive analogy*. We have some organized physics (a "machine") in our hands and a question in our minds. By stimulating the physics ("inputs") and observing its reaction ("outputs"), we somehow learn about something *other than the physics*. Computation is thus the clever craft of reliably putting *doing* into the service of *knowing*. To apply this paradigm, we need an *analogy* between the two. When the analogy is valid (a homomorphism), so are its logical consequences. Our primary tool in computational science and engineering is thus defining valid analogies and reasoning from them correctly.

This paper has explored a precise analogy (functor) that relates analog circuits (computational hardware) to the abstraction we know as *digital computation*, expressed formally in the dependently typed proof assistant Agda, pictured in Figure 1 and summarized as follows: *A "digital circuit" is an analog circuit that respects discrete meanings.* The comma construction from category theory provides a *compositionally correct* methodology for designing and engineering correct circuits, i.e., ones in which correctness is specified and proved at every step of construction. A single vocabulary is used to describe analog circuits, timing, discrete value computations, and correctness conditions, all related succinctly by homomorphisms (functors).

At the heart of the relationship between the analog nature of hardware and the digital abstraction that disciplines its use is *timing* calculation, specified as a function between multi-dimensional intervals of time. When input signals are held steady throughout any (suitably shaped/typed) multi-interval, the outputs are proved to hold steady throughout the computed output multi-interval, and not only steady, but *correct* in the sense of agreeing with the result of applying a specified mathematical function to the held multi-bit value. While functions make for a simple specification of timing, they do not allow the sort of analysis that leads to efficiently clocked circuits. Fortunately (and the primary, specific technical result of this paper), these timing functions turn out to be *linear* over a semiring of intervals and hence can be represented as a (generalized, compositional) matrix and thus richly analyzed. Again, timing matrices speak the same (and an even somewhat richer) categorical language and are related to timing functions homomorphically. This pleasant and useful fact gives a new, compositional account of the "setup" and "hold" timing analyses commonly employed in circuit design, while formally proving correctness of those analyses and relating them intimately to the "tropical" (max-plus and min-plus) semirings.

This work suggests several useful extensions:

- Realistic circuits extract many bits per signal rather than just the single bit implied by stable (as defined in Section 7). The simple one-bit definition of stable can be easily extended to many, allocated sequentially in time. In this way *cyclic* sequential circuits and their semantics, timing analysis, and correctness can be accounted for. (Even without the multi-bit extension, the analysis given in this paper suffices to determine correct clocking of the inner loop of a typical cyclic sequential circuit, i.e., a state machine's combinational transition function.) Preliminary work suggests that a suitable semantic basis can be extrapolated from stream fixed points [Hutton and Jaskelioff 2011; Capretta et al. 2016], using the categorical vocabulary of a traced monoidal category.

- Richer examples of correct digital circuits, including sophisticated, efficient systolic designs, should help evaluate the linear timing analysis and correct design methodology in realistic use.
- Use *full-analog* signals, i.e., continuous in value (voltage) as well as in time:

  $\mathbb{S}' = \mathbb{T} \rightarrow \mathbb{Q}$

  Interpret especially low voltages as false and especially high voltages as true, with in-between voltages having no boolean interpretation, e.g.,

  $_\sim_ : \mathbb{Q} \rightarrow \mathbb{B} \rightarrow \mathsf{Set}$
  $x \sim \mathsf{f} = x \leq 1 / 5$
  $x \sim \mathsf{t} = x \geq 4 / 5$

  Then alter the definition of stable from Section 7 very slightly to use ~ in place of ≡:

  $\mathsf{stable}' : \mathsf{Pred}\ \mathbb{S}'\ 0\ell$
  $\mathsf{stable}'\ \tilde{x} = \exists_2\ \lambda\ (\hat{x} : \mathbb{I})\ (x : \mathbb{B}) \rightarrow \forall\ \{t : \mathbb{T}\} \rightarrow t \in \hat{x} \rightarrow \tilde{x}\ t \sim x$

- Although Section 11 specialized the specification to static (data-independent) timing, the general specification is essentially friendly to dynamic (data-dependent) timing, thus potentially supporting compositionally correct design of "asynchronous" (dynamically, locally synchronized) circuits, as well as hybrids of the two styles such as circuits with multiple clock domains. Such designs are notoriously difficult to get correct, and their analysis relies on continuous time.
- The specific way in which values and timings are computed in the static setting is exactly how derivatives are computed for affine (constant-derivative) functions. I expect that the more general setting of dynamic timing corresponds to generalized automatic differentiation, particularly in the setting of Fréchet derivatives, which are linear maps (often, but not necessarily, represented as Jacobian matrices) [Elliott 2018].
- Although logic gates implement signal *functions*, the gates themselves are implemented via assemblies of transistors, the semantics of which is *relational* rather than functional. Reformulating analog and digital computing to explain the construction of logic gates from their component transistor relations may enable more aggressively optimized design of correct computational hardware.

REFERENCES

Agda Team. 2022. The Agda standard Library (Version 2.0).

John Baez and Mike Stay. 2010. Physics, Topology, Logic and Computation: A Rosetta Stone. In *New Structures for Physics*. 95–172.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda — A functional language with dependent types. In *Theorem Proving in Higher Order Logics*.

Venanzio Capretta, Graham Hutton, and Mauro Jaskelioff. 2016. Contractive functions on infinite data structures. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*.

Edsger W. Dijkstra. 1972. The humble programmer. *Communications of the ACM* 15, 10 (1972). Turing award lecture.

Stephen Dolan. 2013. Fun with semirings: A functional pearl on the abuse of linear algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. 101–110.

Conal Elliott. 2017. Compiling to categories. *Proceedings of the ACM on Programming Languages* 1, ICFP (Sept. 2017).

Conal Elliott. 2018. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 4 (Sept. 2018), 29 pages.

Conal Elliott. 2019. Generalized convolution and efficient language recognition. *CoRR* abs/1903.10677 (2019).

Conal Elliott. 2021. Symbolic and automatic differentiation of languages. In *Proceedings of the ACM on Programming Languages (ICFP)*.

Conal Elliott. 2023. Felix: An Agda category theory library for denotational design.

João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2018. Pi-Ware: Hardware description and verification in Agda. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics)*.

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*.

Murray Gell-Mann. 2009. Beauty and Elegance in Physics. (2009). Talk given at University of Scranton.

Dan R Ghica, Achim Jung, and Aliaume Lopez. 2017. Diagrammatic semantics for digital circuits. In *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*.

Dan R. Ghica and Achim Jung. 2016. Categorical semantics of digital circuits. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*.

Jeremy Gibbons. 2002. Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Lecture Notes in Computer Science, Vol. 2297. Springer-Verlag.

Jonathan S. Golan. 2005. Some recent applications of semiring theory. In *International Conference on Algebra in Memory of Kostia Beidar*.

F. K. Hanna and Neil Daeche. 1992. Dependent types and formal synthesis. *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences* 339 (1992).

William L. Harrison, Chris Hathhorn, and Gerard Allwein. 2021. A mechanized semantic metalanguage for high level synthesis. In *23rd International Symposium on Principles and Practice of Declarative Programming*.

John Hughes. 1989. Why functional programming matters. *Computer Journal* 32, 2 (1989).

Graham Hutton and Mauro Jaskelioff. 2011. Representing contractive functions on streams. *Submitted to the Journal of Functional Programming* (2011).

Charles E. Leiserson and James B. Saxe. 1991. Retiming synchronous circuitry. *Algorithmica* 6, 1–6 (June 1991).

William Lovas and Frank Pfenning. 2009. Refinement types as proof irrelevance. In *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications (TLCA 2009)*.

Saunders Mac Lane. 1998. *Categories for the working mathematician, second edition.* Springer.

Hugo Daniel Macedo and José Nuno Oliveira. 2013. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming* 78, 11 (2013).

John Matthews, Byron Cook, and John Launchbury. 1998. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*. IEEE Computer Society.

Ulf Norell. 2008. Dependently typed programming in Agda. In *Revised Lectures of the Sixth International Spring School on Advanced Functional Programming (Lecture Notes in Computer Science)*.

John T. O'Donnell. 1987. Hardware description with recursion equations. In *IFIP 8th International Symposium on Hardware Description Languages and their Applications*.

Adam M. Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. 2015. Semantics driven hardware design, implementation, and verification with ReWire. *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM* (2015).

Emily Riehl. 2016. *Category Theory in Context*.

Armando Santos and José N. Oliveira. 2020. Type your matrices for great good: A Haskell library of typed matrices and applications (functional pearl). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*.

Mary Sheeran. 1984. MuFP, a Language for VLSI Design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*.

Mary Sheeran. 1988. Slowdown and retiming in Ruby. In *IFIP Workshop on The Fusion of Hardware Design and Verification*.

Mary Sheeran. 2005. Hardware design and functional programming: A perfect match. *JUCS - Journal of Universal Computer Science* 11, 7 (2005).

Richard Townsend, Martha A. Kim, and Stephen A. Edwards. 2015. *Hardware in Haskell: Implementing memories in a stream-based world*. Technical Report CUCS-017-15. Columbia University, Department of Computer Science.

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015).

David Wilding. 2015. *Linear Algebra Over Semirings*. Ph. D. Dissertation. University of Manchester.

Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. 2015. Hardware synthesis from a recursive functional language. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

Received 2023-03-01; accepted 2023-06-27