# Colimit-Based Composition of High-Level Computing Devices

Damian Arellanes
Lancaster University
damian.arellanes@lancaster.ac.uk

Models of High-level Computation (MHCs) provide effective means to describe complex real-world computing systems because they offer formal foundations for the specification of interacting computing devices, as opposed to describing individual ones, which has been the focus of classical models such as Turing machines or the lambda calculus itself. Despite numerous proposals over the past half century, there is still no canonical MHC akin to Turing machines for (compositionally) reasoning about computation in the large. One of the major drawbacks of state- and data-oriented MHCs is that they extensively neglect control flow, a well-known semantic property that defines computation order. Only control-oriented MHCs treat control explicitly at the expense of ignoring data flow or assuming that data follows control. Mixing data and control within the same framework leads to inefficient methods for formal analysis and verification. To address this, the computon model has recently emerged as a category-theoretic MHC that separates data and control and makes control explicit by supporting composition operators characterised as finite colimit constructions. Such constructions allow the formation of sequential, parallel, branching and iterative computing devices. Unfortunately, the computon model is still a generic reference rather than a concrete realisation. In this paper, we provide a variation of it to enable functional computing devices, introduce a new branching operator, discuss how to define synchronous parallelising out of sequencing and asynchronous parallelising, describe concrete operational semantics for computon execution and provide the first implementation of the model. The implementation yields an open-source programming environment that realises the underlying categorical semantics with partial type-level guarantees. This tool is publicly available for building complex computing devices with a high degree of structural correctness by construction.

## 1 Introduction

The Church-Turing thesis states that a function is computable if there is an effective procedure able to produce its values. Such a thesis has been successful for reasoning about computation in the small where a single abstract device realises a well-defined effective procedure. Over the past half century, there has been a collective attempt to move from the small to the large by defining models of computation capable of describing not the behaviour of isolated computing devices, but interactions among a collection of them. This paradigm shift, referred to as Models of High-Level Computation (MHCs) [2], has become increasingly relevant due to the need of describing complex computations in real-world domains (e.g., distributed systems). To date, there is no standard MHC akin to Turing machines for constructing or reasoning about computation in the large, despite the wide variety of MHCs that have emerged over time, e.g., Kahn Process Networks [17], Hierarchical State Machines (HSMs) [15], ONets [7] and, more recently, algebras over operads of wiring diagrams [27].

In the context of MHCs, interaction is the causal effect of composition, an inductive mechanism for gluing together computing devices into more complex ones known as composites. For example, HSMs compose state machines by nesting, ONets compose Petri nets by identifying outputs with inputs via pushout constructions and Kahn networks combine processes by connecting them through (possibly unbounded) FIFO channels. In general, composition can be realised through the combination of states, control flow, data flow or any combination thereof. Although control lies at the heart of computation,

To appear in EPTCS.

because it is ever present in any (low- or even high-level) computing device, it is striking that such a dimension has been treated implicitly by some classes of MHCs [2]. For example, in both HSMs and ONets, control is implicit in the activation of state transitions and, in Kahn networks, in data exchanges. Making control explicit in an MHC leads to formal reasoning of computation order, facilitating the verification of common computational properties such as reachability or termination [1].

The computon model [4, 3] provides categorical semantics to formally compose high-level computing devices by control flow in an incremental and bottom-up manner. For this, it provides composition operators, with behaviour characterised as finite colimit constructions, that define explicit control flow for the activation of computing devices in some well-defined order. Data flow is considered, albeit it is a second-class dimension governed by control. Particularly, there are separate operators for forming sequential, (synchronous and asynchronous) parallel, branching and iterative computing devices. As their internal structure is only accessible through an interface, devices are modular black boxes with explicit separation of concerns. Separating data from control has proven relevant for reasoning about such dimensions independently [10]. For example, one can verify termination by analysing control flow only or data reachability without inspecting control flow at all [1]. Beyond verification, separating concerns has been effective for model transformation purposes [4].

Although the separation of control and data is a distinctive property of the computon model, such a model is still a generic formalism that serves as a reference rather than a concrete realisation dictating how devices should compute. Accordingly, it leaves the interpretation of computing devices open by avoiding concrete implementations and giving operational semantics at a higher-level of abstraction via P/T Petri nets. Moreover, the operators it provides do not offer sufficient flexibility for modelling expressive decision-making structures.

In this paper, we move from the abstract to the particular by endowing computons with structural relations between inputs and outputs to enable functional computing devices, and introduce a new operator for branching. We also show that the original operator for forming synchronous parallel composites can be defined out of sequencing and asynchronous parallelising, and study additional algebraic properties in terms of identity, a law that was not originally analysed in [4, 3]. The original definition of the sequential operator is modified to satisfy this law.

Our ultimate goal is to offer a universal framework to formally reason about high-level computing devices in a compositional manner via categorical semantics. As the proposed framework is sufficiently general, we envision multiple theoretical extensions and tools built out of it. In this paper, we take the first step towards a tool that bridges theory and practice by implementing the computon model in a functional programming language with dependent types. The implementation yields an open-source programming environment, intended for describing high-level computations via colimit constructions. To the best of our knowledge, this is the first implementation of the computon model.

The rest of the paper is structured as follows. Section 2 presents the semantics of a category of computons and their morphisms. Section 3 describes colimit constructions that capture the behaviour of control-driven composition operators for sequencing, parallelising and branching. Section 4 defines concrete operational semantics for computon execution. Section 5 and appendices B and C present an implementation of the computon model. Section 6 discusses related work. Section 7 outlines the conclusions and future work. Appendix A presents the proofs of some results, omitted for space.

## 2 Categories of Computons and Computon Morphisms

A computon is intuitively a bipartite graph where nodes are ports or computation units, connected through edges that represent control flow or data flow. A port is a buffer for storing a datum or a control signal so

it has a type associated to it. To abstract away from concrete types, a finite nonempty set $\underline{n} := [0,n) \cap \mathbb{N}$ is used, capturing the essence of a fixed type system with $n \geq 1$ possible port types. In any choice, 0 represents the type of control signals whereas the other numbers are used for other data types. Having $\underline{1} := \{0\}$ as the minimal choice entails that control ports are always present whereas data ports are optional, i.e., computation is necessarily driven by control signals (cf. Fig. 1c). Although computation units are not mandatory, as shown in Fig. 1a, they always have control ports attached when they exist. In general, Fig. 1 shows diverse computon examples over the set $\underline{7}$.

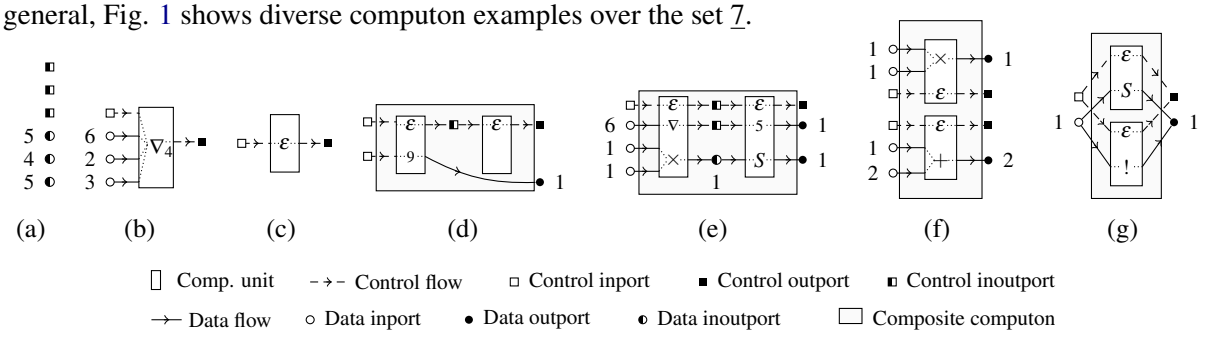



Figure 1: Examples of computons over the set $\underline{7}$ of types and the set $\{\nabla_4, \varepsilon, 9, \nabla, \times, 5, S, +, !\}$ of computing devices which, for clarity, are not displayed as strings but as symbols expressing behaviour. Particularly, the devices in this example correspond to programs for discarding 4 values ($\nabla_4$), echoing a single control signal ($\varepsilon$), producing the constant 9, discarding a single value ($\nabla$), computing binary multiplication ($\times$), producing the constant 5, computing the successor of a number ($S$), adding two numbers ($+$) and computing the factorial of a number ($!$). Computons abstract away from concrete types in order to provide a general implementation-independent framework, e.g., 1 can express the type of integers, 2 the type of floats and so on. In Sect. 4, we describe operational semantics to map natural numbers to concrete types from a fixed type system. Unlike data ports, control ports are never 0-labelled for clarity.

Intuitively, ports are attached to units to indicate the type of values a unit can receive/produce. Rather than denoting a single computational behaviour, a unit is a collection of computing devices (i.e., programs) that receive at least one input and produce exactly one output. Particularly, the inputs of a device $d$ in a unit $u$ come from a subset of ports connected to $u$, whereas the output of $d$ is sent a port connected from $u$. The set of all possible computing devices is a nonempty finite set $B \subset \Sigma^*$, i.e., a device is a finite string over some finite alphabet $\Sigma$.[1] One of the particular properties is that $B$ always contains a device $\varepsilon$ that echoes a single control signal, as shown in Fig. 1c. As there evidently are multiple choices for $B$ and $\underline{n}$, one can form multiple computon categories. For a particular choice, objects are defined as follows.

**Definition 1** (Computon)**.** *A computon $\lambda$ is a 13-tuple $(U,P,I,O,\underline{n},B,\sigma,t,\tau,s,c,r,f)$ where:*

- *$U$ is a finite set of computation units,*
- *$P$ is a finite nonempty set of ports,*
- *$I$ is a finite set of inflows,*
- *$O$ is a finite set of outflows,*
- *$\sigma\colon O \twoheadrightarrow U$ is a surjective function that defines the source unit of each outflow,*
- *$t\colon O \to P$ is a function that specifies the target port of each outflow,*
- *$\tau\colon I \twoheadrightarrow U$ is a surjective function that specifies the target unit of each inflow,*
- *$s\colon I \to P$ is a function that specifies the source port of each inflow,*
- *$c\colon P \to \underline{n}$ is a function that assigns a type to each port,*

[1] Representing a computing device as a finite sequence of symbols over a finite alphabet is valid since it is well-known that every program (including computable functions) can be encoded thereby [23].

- $r\colon I \twoheadrightarrow O$ *is a surjective function relating inflows with outflows, and*
- $f\colon O \to B$ *is a function that attaches each outflow to a computing device*

*such that (i)* $0 \in \underline{n}$*, (ii)* $\varepsilon \in B$*, (iii)* $\tau \restriction_{(c \circ s)^{-1}(0)}$ *and* $\sigma \restriction_{(c \circ t)^{-1}(0)}$ *are surjective, (iv)* $\sigma \circ r = \tau$*, and (v) there are ports* $p \in P \setminus t(O)$ *and* $q \in P \setminus s(I)$ *with* $c(p) = 0 = c(q)$*.*

**Notation 1.** *A computon* $\lambda$ *with* $u \in U$ *and* $p \in P$ *has sets* $s(\tau^{-1}(u))$*,* $t(\sigma^{-1}(u))$*,* $\tau(s^{-1}(p))$ *and* $\sigma(t^{-1}(p))$ *written* $\bullet u$*,* $u\bullet$*,* $p\bullet$ *and* $\bullet p$*, respectively. To distinguish between computons, we use natural numbers to index their components. If a computon symbol has no subindex, its components have no subindex either.*

Rather than directly specifying a function $U \to B$, Definition 1 relies on a span $U \overset{\sigma}{\leftarrow} O \overset{f}{\to} B$ to generalise the functional relation given by the former, i.e., a unit can be related not to just one device, but to multiple ones. Having $U \to B$ alone is not sufficient to allow units produce different outputs consistently, as such a function does not encode any means to determine which values go to which outflows without imposing certain order, i.e., the only way of enabling multiple outputs is through a product type. By equipping computons with a function chain $I \overset{\mathrm{r}}{\twoheadrightarrow} O \overset{f}{\to} B$, we abstract away from particular orderings by offering a direct relationship between inflows and outflows that makes it possible to extract information flows coming from/into computing devices. Concretely, an outflow $o \in O$ takes data from the result of a device $f(o)$ which, in turn, operates on the input values from $r^{-1}(o)$. By the totality of $f$, it is guaranteed that each outflow reads information from exclusively one device.

To encapsulate behaviour within units, Restriction (iv) in Definition 1 enforces devices to read/produce data from/into ports attached to the unit they belong to. Restriction (iii) enforces units to have control ports connected to and from it, and Restrictions (i) and (ii) respectively specify that the control type and the signal echoing device $\varepsilon$ are ever present. Restriction (v) ensures the existence of ports where control flow starts and terminates, called control inports and control outports, respectively. The set $P^+$ of all control and data inports defines the input interface of a computon $\lambda$, whilst the set $P^-$ of all control and data outports give rise to the output interface.

**Definition 2** (Computon Interface)**.** *The interface of a computon* $\lambda$ *is a tuple* $(P^+, P^-)$ *where* $P^+$ *and* $P^-$ *are the sets* $P \setminus t(O)$ *and* $P \setminus s(I)$*, respectively.*

When there is a sequence of flows from every inport (or from any port with outflows) to some outport, we say that a computon is connected (see Definition 3 and Fig. 1e). By Proposition 1, connected computons always have at least one computation unit.

**Definition 3** (Connected Computon)**.** *Let* $(U \sqcup P, I \sqcup O, s', t')$ *be the bipartite graph* $G$ *of a computon* $\lambda$ *with* $s', t'\colon I \sqcup O \to U \sqcup P$ *given as follows:* $s'(e) = \begin{cases} s(e) & \text{if } e \in I \\ \sigma(e) & \text{if } e \in O \end{cases}$ $\qquad t'(e) = \begin{cases} \tau(e) & \text{if } e \in I \\ t(e) & \text{if } e \in O \end{cases}$

*We say that a computon is connected if, for every port* $p \in P^+ \cup s(I)$*, there is a path* $(e_1, e_2, \ldots, e_n)$ *in* $G$ *of length* $n \geq 2$ *with* $s'(e_1) = p$ *and* $t'(e_n) \in P^-$*.*

**Proposition 1.** *If* $\lambda$ *is a connected computon, then* $U \neq \emptyset$*.*

A computon with only interface ports and no units or flows at all is called a *trivial computon* (see Definition 4 and Fig. 1a). A trivial computon with exactly one port is referred to as a *unit computon*.

**Definition 4.** *A computon* $\lambda$ *is trivial if it satisfies* $U = I = O = \emptyset$*. A trivial computon with* $|P| = 1$ *is called a unit computon.*

**Remark 1.** *By Definitions 1, 2 and 4, it is easy to see that a unit computon is an entity with exactly one port* $p \in P^+ \cap P^-$ *with* $c(p) = 0$*, i.e., it consists of only one control port that is inport and outport simultaneously. Ports in* $(P^+ \cap P^-) \cup (s(I) \cap t(O))$ *are referred to as inoutports.*

When a computon has exactly one computation unit to which all ports are attached, we say it is *primitive* (see Fig. 1b). Like trivials, a primitive has all ports at the interface, with the restriction that each inport is connected to the unique unit via a single inflow and outports are linked to the unit through a single outflow. This is enforced by the injectivity condition on the functions $s$ and $t$ in Definition 5.

**Definition 5.** *A computon $\lambda$ is primitive if $|U| = 1$, $P \cong I \sqcup O$ and $s$ and $t$ are injective.*

Note that Definition 1 enforces primitive computons to always have inflows and outflows due to the surjectivity of $\tau$ and $\sigma$. As $s$ and $t$ are not necessarily onto in that definition, it is possible to have ports with no flows attached. To prevent this, $P \cong I \sqcup O$ and injectivity over $s$ and $t$ are required, with the former condition implying that all ports lie at the interface (see Proposition 2).[2]

**Proposition 2.** *If $\lambda$ is a primitive computon, $P \cong P^+ \triangle P^-$.*

## 2.1 Computon Morphisms

A morphism in a computon category is intuitively an insertion of a computon into another. More precisely, it is a collection of six total functions that map units to units, ports to ports, inflows to inflows, outflows to outflows, types to types and devices to devices such that the two last mappings are inclusions to guarantee consistency in terms of port typing and computational behaviour (see Definition 6).

**Definition 6** (Computon Morphism)**.** *In a computon category over a set $\underline{n}$ of types and a set $B$ of computing devices, a morphism $\alpha\colon \lambda_1 \to \lambda_2$ is a 6-tuple $(\alpha_U, \alpha_P, \alpha_I, \alpha_O, \alpha_{\underline{n}}, \alpha_B)$ of total functions $\alpha_U : U_1 \to U_2$, $\alpha_P\colon P_1 \to P_2$, $\alpha_I\colon I_1 \to I_2$, $\alpha_O\colon O_1 \to O_2$, $\alpha_{\underline{n}}\colon \underline{n} \hookrightarrow \underline{n}$ and $\alpha_B\colon B \hookrightarrow B$ such that $\vec{i}(\alpha) \cup \vec{o}(\alpha) \subseteq P_1^+ \cup P_1^-$ and the following diagrams commute:*

$$\begin{array}{ccccc} I_1 & \xrightarrow{\tau_1} & U_1 & \xleftarrow{\sigma_1} & O_1 \\ {\scriptstyle \alpha_I}\downarrow & & \downarrow{\scriptstyle \alpha_U} & & \downarrow{\scriptstyle \alpha_O} \\ I_2 & \xrightarrow[\tau_2]{} & U_2 & \xleftarrow[\sigma_2]{} & O_2 \end{array} \qquad \begin{array}{ccccc} I_1 & \xrightarrow{s_1} & P_1 & \xleftarrow{t_1} & O_1 \\ {\scriptstyle \alpha_I}\downarrow & & \downarrow{\scriptstyle \alpha_P} & & \downarrow{\scriptstyle \alpha_O} \\ I_2 & \xrightarrow[s_2]{} & P_2 & \xleftarrow[t_2]{} & O_2 \end{array} \qquad \begin{array}{ccccc} B_1 & \xleftarrow{f_1} & O_1 & \xleftarrow{r_1} & I_1 \\ {\scriptstyle \alpha_B}\downarrow & & \downarrow{\scriptstyle \alpha_O} & & \downarrow{\scriptstyle \alpha_I} \\ B_2 & \xleftarrow[f_2]{} & O_2 & \xleftarrow[r_2]{} & I_2 \end{array} \qquad \begin{array}{ccc} P_1 & \xrightarrow{c_1} & \underline{n} \\ {\scriptstyle \alpha_P}\downarrow & & \downarrow{\scriptstyle \alpha_{\underline{n}}} \\ P_2 & \xrightarrow[c_2]{} & \underline{n} \end{array}$$

*Here, $\vec{i}(\alpha)$ and $\vec{o}(\alpha)$ are given by $\{p \in P_1 \mid \bullet\alpha_P(p) \setminus \alpha_P(\bullet p) \neq \emptyset\}$ and $\{p \in P_1 \mid \alpha_P(p)\bullet \setminus \alpha_P(p\bullet) \neq \emptyset\}$, respectively. From now on, we use natural numbers as subindices to distinguish between morphisms and abuse notation by omitting components, e.g., we write $\alpha(p)$ for $\alpha_P(p)$ when the context is clear.*

The leftmost diagram in Definition 6 ensures preservation of inflow and outflow adjacency with respect to computation units. The immediate diagram on the right enforces preservation of the adjacency between ports and their information flows. The isolated square ensures that port typing is retained, and the remaining one keeps the relationship between inflows, outflows and devices. The restriction $\vec{i}(\alpha) \cup \vec{o}(\alpha) \subseteq P_1^+ \cup P_1^-$ ensures that a computon can be inserted into another only at its interface, i.e., only inports or outports of the source computon can be attached to new computation units in the target computon. As a consequence, we have the following lemma.

**Lemma 1** ([4])**.** *For any computon morphism $\alpha\colon \lambda_1 \to \lambda_2$, $\alpha^{-1}(P_2^+) \subseteq P_1^+$ and $\alpha^{-1}(P_2^-) \subseteq P_1^-$.*

There is a special class of morphisms, called markers, which allow embedding a trivial computon into a computon interface (see Definition 7). When the embedding covers all the inports, it is called an *in-marker* and, when it covers all the outports, it is called an *out-marker*.

**Definition 7** (Computon Markers)**.** *A marker $\lambda^\square$ of a computon $\lambda$ is a computon monomorphism $\lambda_0 \to \lambda$ where $\lambda_0$ is a trivial computon and $\lambda^\square(P_0) = P^\square$ with $\square \in \{+, -\}$. If $\square = +$, it is called an* in-marker*; otherwise, an* out-marker*.*

[2] In Proposition 2, we use the symbol $\triangle$ to denote symmetric set difference.

Interestingly, there are markers whose domain is a unit computon, indicating that the codomain does not require (or produce) any data but just receives (or produces) a single control signal. Although a unit computon evidently embodies the minimal structure one can establish from Definition 1, such an entity is not an initial object in a computon category since there are $k$ morphisms from it to a computon with $k$ control ports, rather than a unique morphism, i.e., a computon category has no initial objects (cf. [4]).

## 2.2 Colimit Constructions

In the theory of computons, complex embeddings can be built out of "elementary" finite colimits, namely coproduct and pushout. Coproduct is simply the disjoint union of computon components, which gives rise to a composite structure that puts two computons side-by-side (see Definition 8). Pushout is the square complement of a span of computon morphisms, which merges two computons into a composite according to the instructions given by the span (see Definition 9). In [4], we showed that pushouts can only be formed if a span satisfies the restrictions from Definition 10 and that a category of computons has all coproducts (see Theorem 1). In the same work, we also showed that both coproduct and pushout satisfy the required universal properties in a computon category.

**Definition 8** (Coproduct)**.** *In a computon category over a set $\underline{n}$ of types and a set $B$ of computing devices, the coproduct $\lambda_1 + \lambda_2$ of $\lambda_1$ and $\lambda_2$ is given by the following diagram of finite sets and total functions:*

$$\begin{array}{ccccccc} & & O_1 \sqcup O_2 & \xrightarrow{f} & B & & \\ & \overset{\sigma}{\swarrow} & \uparrow{\scriptstyle r} & & & \overset{t}{\searrow} & \\ U_1 \sqcup U_2 & \xleftarrow[\tau]{} & I_1 \sqcup I_2 & & \xrightarrow[s]{} & P_1 \sqcup P_2 & \xrightarrow{c} \underline{n} \end{array}$$

*Supposing $\beta_1 \colon \lambda_1 \to \lambda_1 + \lambda_2$ and $\beta_2 \colon \lambda_2 \to \lambda_1 + \lambda_2$ are the canonical coproduct monomorphisms, the functions $\sigma$, $t$, $\tau$, $s$, $c$, $r$ and $f$ are computed sourcewise. For example, $\sigma$ is given as follows for all $o \in O_1 \sqcup O_2$:* $\sigma(o) = \begin{cases} \beta_1(\sigma_1(o_1)) & \text{if } o = \beta_1(o_1) \text{ for some } o_1 \in O_1 \\ \beta_2(\sigma_2(o_2)) & \text{if } o = \beta_2(o_2) \text{ for some } o_2 \in O_2 \end{cases}$

**Definition 9** (Pushout)**.** *In a computon category over a set $\underline{n}$ of types and a set $B$ of computing devices, the pushout $\lambda_1 +_{\lambda_0} \lambda_2$ of a span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms is given by the following diagram of finite sets and total functions:*

$$\begin{array}{ccccccc} & & O_1 \sqcup_{O_0} O_2 & \xrightarrow{f} & B & & \\ & \overset{\sigma}{\swarrow} & \uparrow{\scriptstyle r} & & & \overset{t}{\searrow} & \\ U_1 \sqcup_{U_0} U_2 & \xleftarrow[\tau]{} & I_1 \sqcup_{I_0} I_2 & & \xrightarrow[s]{} & P_1 \sqcup_{P_0} P_2 & \xrightarrow{c} \underline{n} \end{array}$$

*Supposing $\beta_1 \colon \lambda_1 \to \lambda_1 +_{\lambda_0} \lambda_2$ and $\beta_2 \colon \lambda_2 \to \lambda_1 +_{\lambda_0} \lambda_2$ are the pushout-induced computon morphisms, the functions $\sigma$, $t$, $\tau$, $s$, $c$, $r$ and $f$ are computed sourcewise. For example, $\tau$ is given as follows for all $i \in I_1 \sqcup_{I_0} I_2$:* $\tau(i) = \begin{cases} \beta_1(\tau_1(i_1)) & \text{if } i = \beta_1(i_1) \text{ for some } i_1 \in I_1 \\ \beta_2(\tau_2(i_2)) & \text{if } i = \beta_2(i_2) \text{ for some } i_2 \in I_2 \end{cases}$

**Definition 10** (Pushable Span)**.** *A span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ is pushable if $\alpha_1(\vec{i}(\alpha_2)) \cup \alpha_1(\vec{o}(\alpha_2)) \subseteq P_1^+ \cup P_1^-$ and $\alpha_2(\vec{i}(\alpha_1)) \cup \alpha_2(\vec{o}(\alpha_1)) \subseteq P_2^+ \cup P_2^-$.*

**Remark 2.** *Note that when $i \in I_1 \sqcup_{I_0} I_2$ is identified with elements $i_1 \in I_1$ and $i_2 \in I_2$, it is sufficient for $\tau$ to choose either $\beta_1(\tau_1(i_1))$ or $\beta_2(\tau_2(i_2))$ due to the commutativity equations from Definition 6. The same applies to $\sigma$, $s$, $t$, $c$, $r$ and $f$.*

**Theorem 1** ([4])**.** *Let $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2}$ be a span $\rho$ of computon morphisms. The pushout of $\rho$ exists if and only if $\rho$ is pushable. If $\alpha_1$ and $\alpha_2$ are computon markers, $\rho$ is pushable.*

# 3 Composition Operators

Pushouts and coproducts form the basis of the composition operators the theory of computons builds upon. In this section, we describe separate operators to form sequential, parallel and branching composites, which define explicit control flow for the invocation of computons in some precise order.

## 3.1 Sequential Computons

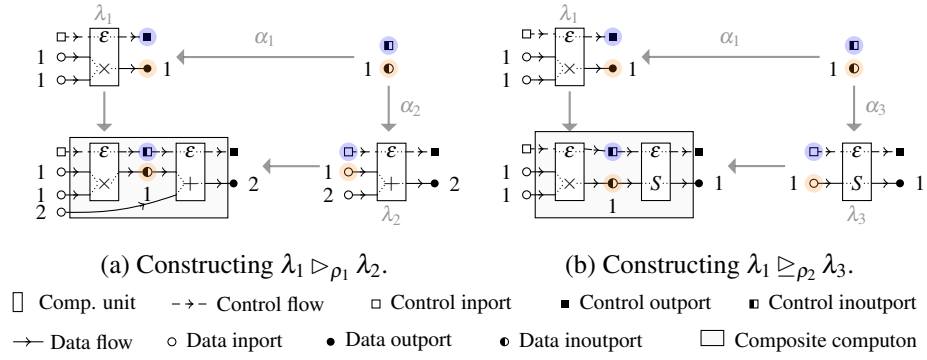
A sequential computon defines a control flow structure for the invocation of two computons in a pipeline. Such a composite is obtained by computing the pushout of a sequentiable span of computon morphisms (see Definition 11). It is total when all the outports of the "first-executed" computon are connected to all the inports of the "secondly-executed" computon, and partial otherwise (see Definition 12).

**Definition 11** (Sequentiable Span)**.** *A span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ is sequentiable if $\lambda_0$ is a trivial computon, and $\alpha_1$ and $\alpha_2$ are monomorphisms with $\alpha_1(P_0) \subseteq P_1^-$ and $\alpha_2(P_0) \subseteq P_2^+$.*

**Theorem 2** ([4])**.** *Every sequentiable span is pushable.*

**Definition 12** (Sequential Computon)**.** *The pushout of a sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ yields a total sequential computon $\lambda_1 \trianglerighteq_\rho \lambda_2$ if $\alpha_1(P_0) = P_1^-$ and $\alpha_2(P_0) = P_2^+$; otherwise, it yields a partial sequential computon $\lambda_1 \triangleright_\rho \lambda_2$.*

**Example 1.** *Suppose we have primitive computons $\lambda_1$, $\lambda_2$ and $\lambda_3$ with units that encapsulate devices for binary multiplication ($\times$), binary addition ($+$) and unary successor ($S$), respectively, each also having the device $\varepsilon$. With these primitives, we can form composites in diverse ways. For example, Fig. 2a shows the construction of a partial sequential computon $\lambda_1 \triangleright_{\rho_1} \lambda_2$, intended for computing $(a \times b) + c$. Partiality occurs because this composite is formed from a sequentiable span $\rho_1 := (\alpha_1, \alpha_2)$ that omits the 2-coloured data inport of $\lambda_2$. An example for the formation of a total sequential computon $\lambda_1 \trianglerighteq_{\rho_2} \lambda_3$ is shown in Fig. 2b, in which there is a sequentiable span $\rho_2 := (\alpha_1, \alpha_3)$ that identifies all the $\lambda_1$-outports with all the $\lambda_3$-inports, yielding a composite intended to compute the successor of $a \times b$. If 1 and 2 are the respective types of nonnegative integers and float numbers, then $a, b \geq 0$, and $c$ would be a float.*




(a) Constructing $\lambda_1 \triangleright_{\rho_1} \lambda_2$. (b) Constructing $\lambda_1 \trianglerighteq_{\rho_2} \lambda_3$.

Figure 2: Examples of partial and total sequencing.

As every computon always possesses control ports, it is always possible to connect at least one computon's control outport with the control inport of another, i.e., every two computons are always sequentiable (see Theorem 3). Lemmas 2 and 3 specify how interfaces are formed for sequential computons. Lemma 4 specifies that the morphisms induced by the pushout of a sequentiable span are mono. By Theorem 4, both total and partial sequencing are not commutative, and only total sequencing is associative. Theorem 5 says that a unit computon serves as the left- and right-identity for both classes of sequencing.

**Theorem 3.** *If $\lambda_1$ and $\lambda_2$ are computons, there exists a sequentiable span $\rho$ whose pushout is either $\lambda_1 \triangleright_\rho \lambda_2$ or $\lambda_1 \trianglerighteq_\rho \lambda_2$.*

**Lemma 2.** *Assume $\rho$ is a sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. If $\lambda_1 \xrightarrow{\beta_1} \lambda_3 \xleftarrow{\beta_2} \lambda_2$ is the cospan induced by the pushout of $\rho$, then $\beta_1(P_1^+) \subseteq P_3^+$ and $\beta_2(P_2^-) \subseteq P_3^-$.*

**Lemma 3.** *Assume that the pushout of $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ is a total sequential computon $\lambda_3$. If $\lambda_1 \xrightarrow{\beta_1} \lambda_3 \xleftarrow{\beta_2} \lambda_2$ is the cospan induced by the pushout of $\rho$, then $\beta_1(P_1^+) = P_3^+$ and $\beta_2(P_2^-) = P_3^-$.*

**Lemma 4.** *If $\lambda_1 \xrightarrow{\beta_1} \lambda_3 \xleftarrow{\beta_2} \lambda_2$ is the cospan induced by the pushout of a sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$, then $\beta_1$ and $\beta_2$ are computon monomorphisms.*

**Theorem 4.** *Total sequencing is associative up to isomorphism, but partial sequencing is not. Both total and partial sequencing are not commutative.*

**Theorem 5.** *Up to isomorphism, the unit computon is the left- and right-identity for both total and partial sequencing.*

*Proof.* It suffices to prove for partial sequencing since the proof of the other is analogous. If we consider that the unit computon $\lambda$ is the left- and right-identity for partial sequencing, we need to show $\lambda \rhd_{\rho_1} \lambda_1 \cong \lambda_1$ and $\lambda_1 \rhd_{\rho_2} \lambda \cong \lambda_1$ for some arbitrary computon $\lambda_1$ and sequentiable spans $\rho_1$ and $\rho_2$. We just prove $\lambda \rhd_{\rho_1} \lambda_1 \cong \lambda_1$ since the other is symmetric.

By Definition 11, we know that $\rho_1$ is a sequentiable span of the form $\lambda \xleftarrow{\alpha} \lambda_0 \xrightarrow{\alpha_1} \lambda_1$. Since $\lambda_0$ must be a trivial computon, $U_0 \cong \emptyset \cong U$, $I_0 \cong \emptyset \cong I$ and $O_0 \cong \emptyset \cong O$ must hold for the functions $\alpha_U$, $\alpha_I$ and $\alpha_O$ to be well-defined. Given that $\alpha$ must be a monomorphism to satisfy Definition 11, we have $P_0 \cong P$ in addition. Consequently, $\lambda_0 \cong \lambda$ holds which directly implies $\lambda \rhd_{\rho_1} \lambda_1 \cong \lambda +_{\lambda_0} \lambda_1 \cong \lambda_1$, as required. □

## 3.2 Parallel Computons

In any computon category, it is possible to form composites to encapsulate control flow for the asynchronous invocation of two computons, as formalised in Definition 13.

**Definition 13** (Async)**.** *An async computon $\lambda_1 + \lambda_2$ is the coproduct of $\lambda_1$ and $\lambda_2$.*

Given that a computon category has all coproducts [4], an async can always be formed (see Theorem 6). By Theorems 7 and 8, asynchronous parallelising (i.e., the operation to form an async) is associative and commutative, but has no identity.

**Theorem 6.** *An async computon $\lambda_1 + \lambda_2$ can always be constructed in a category of computons.*

**Theorem 7.** *Asynchronous parallelising is associative and commutative up to isomorphism.*

**Theorem 8.** *Asynchronous parallelising does not satisfy the identity law.*

*Proof.* Suppose for contradiction that there is a computon $\lambda$ serving as the left- and right-identity for asynchronous parallelising so that $\lambda + \lambda_1 \cong \lambda_1 \cong \lambda_1 + \lambda$ for any computon $\lambda_1$. As $\lambda + \lambda_1$ is the coproduct of $\lambda$ and $\lambda_1$ (see Definition 13), Definition 8 states $P \sqcup P_1 \cong P_1$, which is only true if $P = \emptyset$. But Definition 1 says that the set of ports cannot be empty so $\lambda$ cannot be a left-identity. Disproving the existence of a right-identity follows analogously. □

In [4], an operator for synchronous parallelising was described, relying on a special class of computons referred to as joins, which are primitive computons with exactly two control inports and one control outport. In the present work, we generalise such a notion in the form of a so-called *glue*.

**Definition 14** (Glue)**.** *A glue is a primitive computon $\lambda$ with $c(p) = 0$ for all $p \in P$.*

As a join is just a special glue, glues can be used to build *sync composites* to synchronise the execution of multiple computons (see Definition 15). By Theorem 9, such structures can always be formed.

**Definition 15** (Sync)**.** *Given computons $\lambda_1$ and $\lambda_2$, a sync computon is the composite $(\lambda_1+\lambda_2)\square\lambda_3$ where $\lambda_3$ is a glue, $\square \in \{\rhd_\rho, \trianglerighteq_\rho\}$ and $\rho$ is a sequentiable span that identifies all the control outports of $\lambda_1+\lambda_2$ with all the inports of $\lambda_3$.*

**Remark 3.** *Although every chosen glue computon $\lambda_3$ is required to satisfy $P_3^+ \cong C_1^- \sqcup C_2^-$ with $C_1^- := \{p \in P_1^- \mid c_1(p) = 0\}$ and $C_2^- := \{p \in P_2^- \mid c_2(p) = 0\}$, there are no specific requirements for $P_3^-$. So, the choice of $\lambda_3$ is not uniquely defined. Similarly, the choice of $\rho$ is not unique because the pushout can yield a total or a partial sequential computon. Therefore, a sync computon is not uniquely defined.*

**Theorem 9.** *A sync computon can always be constructed in a category of computons.*

*Proof.* Given objects $\lambda_1$ and $\lambda_2$ in a computon category over a set $\underline{n}$ of types and a set $B$ of computing devices, we define $C_j^- := \{p \in P_j^- \mid c_j(p) = 0\}$ for $j \in \{1,2\}$ and the coproduct $(C_1^- \sqcup C_2^-) \sqcup (C_1^- \sqcup C_2^-)$ with canonical injections $\iota_1, \iota_2 \colon C_1^- \sqcup C_2^- \to (C_1^- \sqcup C_2^-) \sqcup (C_1^- \sqcup C_2^-)$. With this, we construct a glue $\lambda_3$: $U_3 = \{u\}$, $P_3 \cong (C_1^- \sqcup C_2^-) \sqcup (C_1^- \sqcup C_2^-)$, $I_3 \cong C_1^- \sqcup C_2^-$ and $O_3 \cong C_1^- \sqcup C_2^-$. If $\phi \colon P_3 \to (C_1^- \sqcup C_2^-) \sqcup (C_1^- \sqcup C_2^-)$, $\gamma \colon I_3 \to C_1^- \sqcup C_2^-$ and $\zeta \colon O_3 \to C_1^- \sqcup C_2^-$ are the isomorphisms, we have: $s_3 = \phi^{-1} \circ \iota_1 \circ \gamma$, $t_3 = \phi^{-1} \circ \iota_2 \circ \zeta$, $\sigma_3(o) = u = \tau_3(i)$, $c_3(p) = 0$, $f_3(o) = \varepsilon$ and $r_3 = \zeta^{-1} \circ \gamma$ for all $i \in I_3$, $o \in O_3$ and $p \in P_3$.

Since $\phi^{-1}$, $\gamma$ and $\zeta$ are isomorphisms and $\iota_1$ and $\iota_2$ are necessarily injective, $s_3$ and $t_3$ are injective too. The functions $\sigma_3$ and $\tau_3$ are surjective because they map flows to the unique $u \in U_3$. The function $r_3$ is surjective because $\gamma$ and $\zeta^{-1}$ are. Since $|U_3| = 1$ and $P_3 \cong (C_1^- \sqcup C_2^-) \sqcup (C_1^- \sqcup C_2^-) \cong I_3 \sqcup O_3$, $\lambda_3$ is primitive as per Definition 5. It also is a glue because all its ports are zero-coloured through $c_3$.

Now, consider the trivial computon $\lambda_4$ given by $P_4 \cong I_3$ with $c_4(p) = 0$ for all $p \in P_4$. Assuming $\psi \colon P_4 \to I_3$ is the corresponding isomorphism, we construct the computon morphism $\alpha_1 \colon \lambda_4 \to \lambda_1 + \lambda_2$ by taking $\gamma \circ \psi$ as the $P$-component, and the identity functions on $\underline{n}$ and $B$ as the $\underline{n}$- and $B$-components, respectively. Here, the function $\gamma \circ \psi \colon P_4 \to C_1^- \sqcup C_2^-$ is well-defined because $C_1^- \sqcup C_2^- \subseteq P_1 \sqcup P_2$. Having the inclusions $\vec{i}(\alpha_1) \cup \vec{o}(\alpha_1) \subseteq P_4 = P_4^+ \cap P_4^- \subseteq P_4^+ \cup P_4^-$, $\alpha_1$ must be a valid computon morphism as per Definition 6. In fact, it is a monomorphism because all its components, including $\gamma \circ \psi$, are injective.

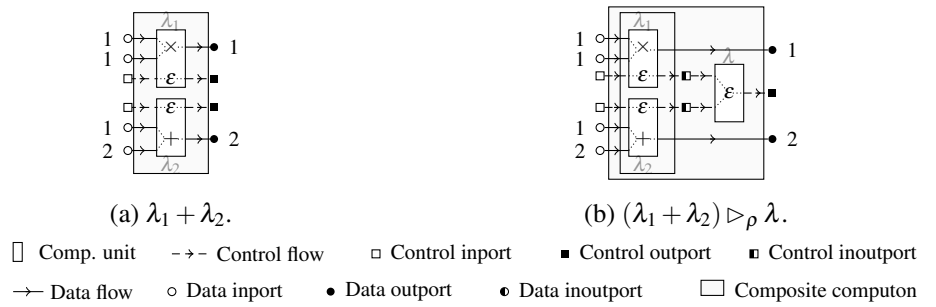
Now, construct $\alpha_2 \colon \lambda_4 \to \lambda_3$ by taking $s_3 \circ \psi$ and the identities on $\underline{n}$ and $B$ as the corresponding $P$-, $\underline{n}$- and $B$-components. Checking that $\alpha_2$ satisfies Definition 6 is analogous to $\alpha_1$, so $\alpha_2$ is also a computon morphism. Given that all the $\alpha_2$-components are injective, including $s_3 \circ \psi$, $\alpha_2$ is also a monomorphism.

As Definition 2 entails $\alpha_2(p) = s_3(\psi(p)) \notin P_3^-$ for all $p \in P_4$, we simply apply Proposition 2 to deduce $\alpha_2(p) \in P_3^+$, i.e., $\alpha_2(P_4) \subseteq P_3^+$. Having the fact $(\gamma \circ \psi)(P_4) \subseteq C_1^- \sqcup C_2^- \subseteq P_1^- \sqcup P_2^-$ in addition and considering that $\alpha_1$ and $\alpha_2$ are mono, the span $\rho := (\lambda_1 + \lambda_2) \xleftarrow{\alpha_1} \lambda_4 \xrightarrow{\alpha_2} \lambda_3$ must be sequentiable (see Definition 11). Therefore, $(\lambda_1 + \lambda_2) \rhd_\rho \lambda_3$ or $(\lambda_1 + \lambda_2) \trianglerighteq_\rho \lambda_3$ can be formed. □

Basically, the proof of Theorem 9 yields a 2-stage construction to form a sync composite out of computons $\lambda_1$ and $\lambda_2$. The idea is to first define $\lambda_1 + \lambda_2$ (for parallel execution) and then use sequencing to connect all the control outports of $\lambda_1 + \lambda_2$ with all the inports of a glue (which waits for the async's termination). Evidently, the apex of the sequentiable span must be a trivial computon with control ports only.

As we have been discussing composition semantics only, it might not seem entirely obvious how a sync behaves. To elucidate this, let us consider an example.

**Example 2.** *Figs. 3a and 3b use the same primitive computons as Fig. 2 to form $\lambda_1 + \lambda_2$ and $(\lambda_1 + \lambda_2) \rhd_\rho \lambda$, respectively, in order to perform binary product and binary addition in parallel. The only difference is that 3b awaits termination via the glue $\lambda$. Although the sequentiable span $\rho$ is not shown due to space constraints, it should be clear that it meets Definition 15 as it identifies all the control outports of $\lambda_1 + \lambda_2$ with all the inports of $\lambda$. In this case, the identification corresponds to partial sequencing because data ports are omitted. Consequently, there is an effect in which data ports traverse several composite layers.*




(a) $\lambda_1 + \lambda_2$. (b) $(\lambda_1 + \lambda_2) \rhd_\rho \lambda$.

Figure 3: Examples of asynchronous and synchronous parallelising.

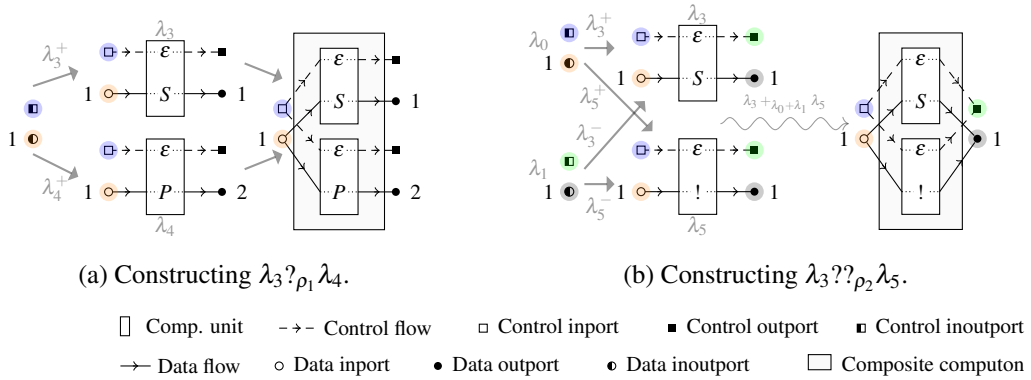
### 3.3 Branching Computons

A branching computon embodies an essential structure for non-deterministic decision-making, allowing to choose a computon out of two alternatives. Such a class of composites can be either open or closed. The former is simply the pushout of a span of in-markers of the operands (see Definition 16), whereas the latter is the colimit of a so-called *b-diagram* which basically is a span of in-markers together with a span of out-markers (see Definition 17). In both cases, the in-markers are embedded into every operand inport so a branching structure can only be formed when inports fully match. In the case of open branching, the outports of the operands remain untouched because there are no out-markers; thus, contrasting with the full outport identification enforced by closed branching.

**Definition 16** (Open Branching Computon)**.** *An open branching computon $\lambda_2 ?_\rho \lambda_3$ is the pushout of a span $\rho := \lambda_2 \xleftarrow{\lambda_2^+} \lambda_0 \xrightarrow{\lambda_3^+} \lambda_3$.*

**Definition 17** (Closed Branching Computon)**.** *A b-diagram $\rho$ is a pair of spans, $\lambda_2 \xleftarrow{\lambda_2^+} \lambda_0 \xrightarrow{\lambda_3^+} \lambda_3$ and $\lambda_2 \xleftarrow{\lambda_2^-} \lambda_1 \xrightarrow{\lambda_3^-} \lambda_3$, where $\lambda_2$ and $\lambda_3$ are connected computons. A closed branching computon $\lambda_2 ??_\rho \lambda_3$ is the colimit of $\rho$, computed as $\lambda_2 +_{\lambda_0+\lambda_1} \lambda_3$.*

**Example 3.** *To elucidate Definitions 16 and 17, suppose that in addition to the computon $\lambda_3$ from Fig. 2, we have primitives $\lambda_4$ and $\lambda_5$ for computing the predecessor of a natural number ($P$) and the factorial function ($!$), respectively. With them, we can form the branching computons displayed in Fig. 4.*




(a) Constructing $\lambda_3 ?_{\rho_1} \lambda_4$. (b) Constructing $\lambda_3 ??_{\rho_2} \lambda_5$.

Figure 4: Open and closed branching. The wavy arrow is not a morphism, but it expresses the operation $\lambda_3 +_{\lambda_0+\lambda_1} \lambda_5$ to form the closed branching computon $\lambda_3 ??_{\rho_2} \lambda_5$ from the b-diagram $\rho_2$ shown on the left of (b). The open branching computon $\lambda_3 ?_{\rho_1} \lambda_4$ is the pushout of the span $\rho_1$ of in-markers $\lambda_3^+$ and $\lambda_4^+$.

*Fig. 4a shows an example of an open branching composite able to choose between the successor and predecessor primitives in a non-deterministic manner, whereas Fig. 4b displays a closed branching composite for choosing either the successor or the factorial function. Fig. 4a particularly unfolds the flexibility unleashed by the operator described in Definition 16, i.e., the operand outports do not have to fully match as in the case of closed branching.*

By Theorems 10 and 12, both operators satisfy associativity and commutativity, but only open branching has left- and right-identities (see Theorems 11 and 13).

**Theorem 10.** *Open branching is associative and commutative up to isomorphism.*

**Theorem 11.** *If $\rho$ is the span $\lambda_2 \xleftarrow{\lambda_2^+} \lambda_0 \xrightarrow{\lambda_3^+} \lambda_3$ of in-markers with $\lambda_0 \cong \lambda_2$, then $\lambda_2 ?_\rho \lambda_3 \cong \lambda_2 +_{\lambda_0} \lambda_3 \cong \lambda_3$.*

*Proof.* As $\lambda_2$ must evidently be a trivial computon because $\lambda_0 \cong \lambda_2$, we use Definition 9 to deduce that $\lambda_2 ?_\rho \lambda_3$ has $|U_3|$ units, $|P_2|+|P_3|-|P_0|$ ports, $|I_3|$ inflows and $|O_3|$ outflows. Particularly, $\lambda_2 ?_\rho \lambda_3$ has $|P_3|$ ports because $|P_0| = |P_2|$ by $\lambda_0 \cong \lambda_2$. As the sets of types and devices is the same for every computon in the same category, we conclude $\lambda_2 ?_\rho \lambda_3 \cong \lambda_2 +_{\lambda_0} \lambda_3 \cong \lambda_3$. □

**Theorem 12** ([4])**.** *Closed branching is associative and commutative up to isomorphism.*

**Theorem 13.** *Closed branching does not satisfy the identity law.*

*Proof.* Suppose for contradiction that a computon $\lambda_2$ is the left-identity of closed branching so $\lambda_2 ??_\rho \lambda_3 \cong \lambda_3$ for some computon $\lambda_3$ and some b-diagram $\rho$ formed by spans $\lambda_2 \xleftarrow{\lambda_2^+} \lambda_0 \xrightarrow{\lambda_3^+} \lambda_3$ and $\lambda_2 \xleftarrow{\lambda_2^-} \lambda_1 \xrightarrow{\lambda_3^-} \lambda_3$. As $\lambda_2$ and $\lambda_3$ must be connected as per Definition 17, Proposition 1 says that units $u_2 \in U_2$ and $u_3 \in U_3$ must exist. If we assume that $\beta_1 \colon \lambda_2 \to \lambda_2 ??_\rho \lambda_3$ and $\beta_2 \colon \lambda_3 \to \lambda_2 ??_\rho \lambda_3$ are the pushout-induced morphisms by $\lambda_2 +_{\lambda_0+\lambda_1} \lambda_3$, we have two possible scenarios:

1. There is some unit $u$ in $\lambda_2 ??_\rho \lambda_3$ where $\beta_1(u_2) = u = \beta_2(u_3)$. So, there must also be a unit in $\lambda_0 + \lambda_1$ identified with $u$. Thus, contradicting that $\lambda_0 + \lambda_1$ has no units because $U_0 \sqcup U_1 = \emptyset \sqcup \emptyset = \emptyset$ by the fact that $\lambda_0$ and $\lambda_1$ are trivial computons (see Definitions 7 and 8).
2. There is no unit in $\lambda_3$ identified with $\beta_1(u_2)$ through $\beta_2$. In this case, $\lambda_2 +_{\lambda_0+\lambda_1} \lambda_3$ has $|U_2|+|U_3|$ units because $|U_0|+|U_1| = 0$. As $|U_2| \geq 1$ because $\lambda_2$ is connected, $|U_2|+|U_3| > |U_3|$ so that $\lambda_2 +_{\lambda_0+\lambda_1} \lambda_3 \cong \lambda_2 ??_\rho \lambda_3 \cong \lambda_3$ cannot hold.

Proving that there is no right-identity is completely symmetric. Hence, the theorem holds. □

## 4 Operational Semantics

In this section, we describe execution semantics for arbitrary computons over a fixed set $\underline{n}$ of types and a fixed set $B$ of devices. For the sake of simplicity, we assume that a type is just a set of values, as in the formalisation of the operational semantics of coloured Petri nets [16]. As we are dealing with natural numbers to represent types, we require a deterministic way of mapping numbers to concrete types. This is done through the typing function described in Definition 18.

**Definition 18** (Typing Function)**.** *The typing function $T$ for a computon $\lambda$ has the signature $\underline{n} \to \mathscr{U}$ where $\mathscr{U}$ is the universe of types of some fixed type system. We assume $\mathscr{C}$ is in $\mathscr{U}$, which is the control type containing a single value $*$ that represents a control signal.*

A typing function specifies the type of values a port can buffer. Definition 19 establishes that the collection of values associated to each port at time $j$ gives the state of a computon at $j$.

**Definition 19** (Computon State)**.** *The state of a computon $\lambda$ at $j$ is a function $\delta^j\colon P \to (\bigcup_{x\in\underline{n}} T(x)) \cup \{\bot\}$ where, for all $p \in P$, $\delta^j(p)$ is of type $T(c(p))$. We say $\delta^j$ is initial if $j = 0$, $\delta^j(p) \neq \bot$ for every $p \in P^+$ and $\delta^j(q) = \bot$ for all $q \notin P^+$. Here, we use $\bot$ to express value absence.*

In each state, units can be enabled or idle. Definition 20 states that a unit is enabled at $j$ if all the ports connected to it have values assigned by $\delta^j$; otherwise, it is idle. When all units are idle, a final state is reached (see Definition 21).

**Definition 20** (Computation Unit Status)**.** *A unit $u \in U$ of a computon $\lambda$ is enabled at time $j$ if $\delta^j(p) \neq \bot$ for all $p \in \bullet u$; otherwise, $u$ is idle under $\delta^j$.*

**Definition 21** (Termination)**.** *A state $\delta^j$ of a computon $\lambda$ is final if each unit $u \in U$ is idle under $\delta^j$.*

When a parallel or a branching composite are in an initial state, multiple units can be enabled simultaneously. Particularly, in parallel computons, all enabled units are triggered simultaneously. In the case of branching, only one unit is chosen for activation at a time. To deal with such non-determinism, units are partitioned according to the source ports they share. For example, if $u_1$ and $u_2$ are two units with $\bullet u_1 = \bullet u_2$, then they belong to the same partition. To select a concrete unit from each partition, we simply invoke the axiom of choice which non-deterministically chooses a representative unit. The collection of representatives yields the set of units ready for evaluation (see Definition 22).

**Definition 22** (Ready Computation Units)**.** *Given a computon $\lambda$, let $E^j$ be the finite set of computation units enabled under $\delta^j$ and $\sim$ be the equivalence relation on $E^j$ given by $u_1 \sim u_2 \iff \bullet u_1 = \bullet u_2$ for all $u_1, u_2 \in E^j$. If $A$ is the partition induced by $\sim$ and $h$ is the random choice function on $A$, $\{h(E) \mid E \in A\}$ is the set $R^j$ of computation units that are ready to be evaluated under $\delta^j$.*

After forming the set $R^j$ under a state $\delta^j$, the computing devices of each $R^j$-unit are invoked to yield a new state at time $j+1$ (see Definitions 23 and 24).

**Definition 23** (Computation Unit Evaluation)**.** *Given an outflow $o \in O$ of a computon $\lambda$ and a state $\delta^j$, the result $[\![f(o)]\!]^j$ of evaluating a computing device $f(o)$ under $\delta^j$ is $f(o)(\delta^j(p_1), \ldots, \delta^j(p_k))$ such that there is a bijection $\phi\colon \{1,\ldots,k\} \to s(r^{-1}(o))$ satisfying $p_m = \phi(m)$ for all $m \in \{1,\ldots,k\}$. We say that the value $[\![f(o)]\!]^j$ is well-typed if and only if it is an element of $(T \circ c \circ t)(o)$.*

**Definition 24** (State Transition)**.** *Given the state $\delta^j$ of a computon $\lambda$ at time $j \geq 0$, $\delta^{j+1}$ is given as follows for all $p \in P$:*

$$\delta^{j+1}(p) = \begin{cases} * & (\exists u \in R^j)[p \in u\bullet \text{ and } T(c(p)) = \mathscr{C}] \\ [\![f(o)]\!]^j & (\exists u \in R^j)(\exists o \in O)[\sigma(o) = u \text{ and } t(o) = p] \\ \delta^j(p) & (\nexists u \in R^j)[p \in \bullet u \cup u\bullet] \\ \bot & \text{otherwise} \end{cases}$$

At time $j+1$, a new state is constructed from $\delta^j$ according to the four cases considered by Definition 24. The first case serves to store a control signal in each control port connected from each computation unit in $R^j$. The second one assigns the result of each computing device from each $R^j$-unit. The third case serves the role of a memory to keep untouched the values of those ports attached to idle units under $\delta^j$. The last case simply assigns the value $\bot$ to the ports connected to each unit in $R^j$, indicating that those (input) values have been consumed by ready units. These four cases collectively define a state transition during a computon's execution, a process that continues until reaching a state in which all units are idle. In other words, termination occurs when there is a finite orbit of states from the computon's initial state to a computon's final state. Although termination is not guaranteed in general, it is expected for composites.

## 5 Implementation

We implemented the semantics of the computon model in Idris 2 [9], a functional programming language that treats (dependent) types as first-class entities, which allowed an almost direct realisation of the key semantic constructs. The first step in this process was to select a suitable representation for finite sets and total functions. For this, we found `Fin m` adequate, which is a type with exactly $m$ inhabitants corresponding to the natural numbers $0,\ldots,m-1$. Totality of `Fin` functions is a built-in property enforced by the Idris compiler. After deciding such representations, we implemented a suite of Idris functions to perform basic set operations such as fiber and image. With this, we subsequently implemented functions to compute disjoint union, union and pushout over `Fin` types.

Our purpose is to support automated composition via control-driven composition operators, so we are interested in the computational angle of colimit constructions. Accordingly, both union and disjoint union are characterised as records with four fields each: the cardinality of two operands and two canonical injections from the operands to the colimit construction being built. Cardinalities play an important role in our implementation since they define actual finite sets.[3] Accordingly, considering that `Fin m` has exactly $m$ monotonically increasing inhabitants, the disjoint union and union of `Fin x` with `Fin y` is simply `x+y` and `maximum(x,y)`, correspondingly. The pushout of `Fin` functions is a record that contains the cardinality of the pushout object as well as the two pushout-induced functions into the pushout object. Automatically constructing such functions is done through Algorithm 1 (presented in Appendix B).

The implementation of pushout and coproduct over `Fin` forms the basis on which the notion of a category of computons is implemented upon. A computon object $\lambda$ is particularly a record with 15 fields, holding the cardinalities of $U$, $P$, $I$, $O$ and $\underline{n}$ as well as the total functions $s$, $t$, $\sigma$, $\tau$, $c$, $r$ and $f$ (see Appendix C). The cardinality of $B$ is not needed since we use the `String` type to represent computing devices. Accordingly, $f\colon O \to B$ is an Idris function `Fin o` $\to$ `String`, where `o` is the cardinality of $O$. In addition to the components from Definition 1, the `Computon` record includes fields to prove the non-emptiness of $P$ and $\underline{n}$. By the definition of a `Fin n` type, having $n > 0$ entails that the number 0 is always included. Proofs of surjectivity for $\sigma$, $\tau$ and $r$ are not required within the `Computon` record since we only deal primitives and trivials explicitly from which composites are built. Thus, such proofs are just needed for the record representing primitive computons, which in addition requires proofs of injectivity for $s$ and $t$ and proofs that $|U| = 1$ and $P \cong I \sqcup O$.[4] The record representing trivial computons does not require surjectivity proofs at all, since such a property holds directly from Definition 4. In this case, it is sufficient to keep fields for the proofs of $|U| = |I| = |O| = 0$.

Unlike computons, a computon morphism is specified as a record with two arguments for delimiting domain and codomain (see Appendix C). Apart from the six components from Definition 6, a `Morphism` record requires a proof that the function between computing devices is an inclusion as well as a proof that both boundary conditions from Definition 6 are met.[5] This record forms the basis of a computon monomorphism which also is a record but with proofs that all the components from Definition 6 are injective. A monomorphism serves in turn as the basis for specifying in- and out-markers. The former requires a mono from a trivial computon into all the inports of the codomain computon. The latter is similar but requires a proof that the trivial computon can be inserted into all the outports of the codomain.

To assist developers in forming trivials, primitives, (mono)morphisms and markers, we provide Idris functions that facilitate the construction of such objects. To specify colimits, we also define

[3] This simplification comes from the fact that the isomorphism class of a finite set $A$ can be identified with $|A|$.

[4] If `Fin p`, `Fin i` and `Fin o` are the types representing the finite sets $P$, $I$ and $O$ of a primitive computon $\lambda$, $P \cong I \sqcup O$ reduces to verifying `p=i+o`.

[5] The proof of inclusion for the $\underline{n}$-component is constructed dynamically to reduce proof tasks for developers.

records for spans, coproducts and pushouts. Coproduct and pushout are simply computed as in Definitions 8 and 9 through Algorithm 2 (discussed in Appendix B). By offering functions to compute coproducts and pushouts, we provide a basis for the implementation of the elementary composition operators described in Sect. 3. The signatures of the operators for total/partial sequencing, asynchronous parallelising and closed/open branching, along with all the computon constructions presented in this section, are described in Appendix C. The operator for synchronous parallelising is not implemented since this is built out of partial sequencing and asynchronous parallelising. The source code of the implemented programming environment, together with examples, are publicly available at `https://github.com/damianarellanes/computons-idris`.

As the environment was implemented in Idris, the universe of data types for the operational semantics correspond to it. To avoid parsing the static structure of a computon at run-time, we defined a data type that embodies a structural simplification, using vectors for direct access. For a computon $\lambda$, such a simplification stores $P^{-}$, a vector that specifies which ports buffer control, vectors for mapping each $u \in U$ to $\bullet u$ and to $u\bullet$, a vector from each $p \in t(O)$ to the unit-indexed computing devices that $p$ reads from and a vector that specifies which units are enabled at a particular point in time. A state is simply a vector of size $|P|$ where each index represents the value of each port. Such values are of `IO` type because they have side effects, as a result of treating each computing device as a string that represents the network endpoint of a behaviour given in the form of a web service. That is, computing devices are implemented as web services. We treat devices in this way to enable interoperability while avoiding colimit computations over large strings or source code compilation.[6] Instead, $B$-elements are simply IP addresses contacted upon evaluation as per Definition 23. Evaluating a computation unit yields a new vector/state to be processed in the next time step, as prescribed by Definitions 22 and 24. This process continues until reaching a final state of the computon being executed.

## 6 Related Work

The computon model, originally introduced in [4], describes composition operators for total/partial sequencing, synchronous/asynchronous parallelising, closed branching and head- and tail-iteration. Unfortunately, such an original formulation is a generic reference rather a concrete realisation that dictates how to compute at the low-level. Accordingly, the original computon definition neither defines functional relations between inflows and outflows nor imposes particular flow order within computation units, leading to a situation in which a unit $u$ has to either replicate the same value to all the data ports in $u\bullet$ or enforce $u\bullet$ to have only one data port (for holding a single datum or a product type value). To remediate this structural inflexibility with operational consequences, Definition 1 equips computons with a set $B$ of computing devices, together with a function chain $I \overset{\mathrm{r}}{\twoheadrightarrow} O \overset{f}{\rightarrow} B$ that satisfies $\sigma \circ r = \tau$ to encapsulate flows within units.

Unlike [4], Sect. 4 describes operational semantics to dictate how devices are triggered within units. In [4], there is no concrete implementation or programming environment, and execution is described generically at a higher-level of abstraction via P/T Petri nets. Although similar operational semantics to ours have been proposed in [3], such a work handles units homogeneously in the form of families of NAND operators to show that the computon model is able to perform any Boolean function when treated as a non-uniform model of computation. In our work, a computation unit is a collection of computing devices each represented as a finite sequence of symbols over a finite alphabet.

A crucial difference with respect to [3] and [4] is that they do not provide any operator for open

[6] By supporting interoperability through web services, one computing device can be written in Java, another in Haskell and another in Python, just to give a few examples. That is, our implementation enables a practical hybrid model of computation.

branching, needed to enhance flexibility towards more expressive decision-making composites. In the present work, we found that there is no need to provide an explicit operator for synchronous parallelising since such a behaviour can be constructed compositionally out of partial sequencing and asynchronous parallelising (see Sect. 3.2). Although we do not offer operators for looping due to the lack of space, they can be easily integrated and implemented within our framework. Unlike [4], our work studies identity laws for composition operators and replaces the original formulation of sequencing to enable this property. In particular, connectivity in the sense of Definition 3 is not further required.

Beyond the original computon model, there have been other MHCs that treat control flow explicitly. Process algebras [6, 20] form perhaps the most prominent family of MHCs which provide concrete operators to compose processes by control flow. Although data flow can separately be observable and analysable, there is no explicit support for partial sequencing. String diagrams [21] are becoming increasingly popular to specify high-level processes formally since they are grounded in categorical semantics for reasoning about computation in a compositional manner. As syntax, they provide a graphical notation for monoidal categories by visually encoding composition rules: wires denote data flows coming into/from boxes which, in turn, represent processes. Although there are no special wires for representing control flow, string diagrams support sequential, asynchronous parallel and iterative composition through morphism composition, tensor product and traces [22]. Sequencing is only total and there is no support for synchronous parallelising. Recently, (probabilistic) branching has been introduced [25] and there have been efforts to colouring wires, but not for separating concerns [14]. In any case, string diagrams assume that data follows control, as evidenced by [8]; consequently, they do not support situations in which data reaches computing devices independently from control signals.

Workflow nets [1] allow the formal specification of control-driven computing devices, but do not offer any separation of concerns and, unlike process algebras and string diagrams, they do not provide any formal operators to compositionally form control-driven composites. Exogenous Connectors [19] and Behaviour Trees [12] alleviate this composition issue by providing separate operators for sequencing, branching, parallelising and looping but, unfortunately, the data dimension is left implicit. In the case of [19], there have been attempts to separate data from control, albeit without any formal semantics [5, 18]. Prosave [26] and SCADE [11] offer such a separation, but also in an informal manner.

## 7 Conclusions and Future Work

In this paper, we introduced and extended a novel MHC, referred to as the computon model, in which (trivial and primitive) computons are the fundamental building blocks. A trivial computon is intuitively a collection of typed ports, whereas a primitive one has a unique computation unit that encapsulates a collection of (potentially interrelated) computing devices, only accessible through a well-defined port-based interface. Computon interfaces have ports to buffer control signals, whereas data ports are optional. This deliberate design facilitates inductive composition for forming explicit control flow structures for the invocation of computing devices in some order. In Sect. 3, we described finite colimit constructions for composition operators to allow the formation of total/partial sequential computons, synchronous/asynchronous parallel computons and open/closed branching ones. In that section, we showed that total sequencing satisfies associativity and identity, whereas partial sequencing only fulfils identity. Asynchronous parallelising and closed branching are both associative and commutative. Only open branching satisfies all the three properties. In the future, we plan to study further algebraic laws (e.g., inversibility) and provide extra colimit constructions to support other forms of composition such as conditional looping. Offering more complex forms of (compositional) concurrency is also a future direction.

In any theoretical extension, control flow must still be a first-class composition dimension because it is what gives rise to the notion of computation. Although control is always present in any low- or even

high-level computing device [2], data flow must not be neglected but it must be treated as a secondary dimension always governed by control [24]. Governance does not imply that data follows control as, in some high-level devices, control signals may arrive before data values, leading to constituent computing devices waiting until receiving all the input data they need. This situation can particularly arise in partial sequencing or asynchronous parallelising. Despite that data does not necessarily follow control, computation order is still predictable in our proposal due to the synchronous evaluation of computation units, as described in Sect. 4. More precisely, Sect. 4 stipulates that a computation unit remains idle until receiving information in all the ports connected to it. When all input information becomes available, an idle unit transitions to an enabled state. Given that non-determinism is introduced by branching composites, multiple units can be enabled simultaneously in which case only one is chosen for evaluation arbitrarily. In the future, we would like to extend the computon model with support for probabilistic choice and enhance its expressivity through conditional evaluation. Enabled units are currently evaluated by triggering their encapsulated computing devices on the inputs they receive from the external environment or from other computons. Upon evaluation, new values are stored in the ports connected from those units, in order to reach a new state. This process continues until reaching a state in which all units become idle.

The operational semantics described in Sect. 4 can be interpreted in a categorical setting for a more rigorous study of computon execution. Operational semantics is not the focus of this paper but colimit-based composition is. In the future, we plan to formalise operational semantics in the language of symmetric monoidal categories by defining ports and computing devices as objects and morphisms of a category, respectively. Although we use categorical semantics to formalise our proposed model, it is important to mention that other formal languages can be used instead. That is, the proposed MHC is independent of its specification, just as it occurs with any other foundational model.

To bridge theory and practice, we implemented the categorical semantics of the computon model in Idris 2, including the formalisation of computons, computon morphisms and colimit constructions. The implementation yields a programming environment for forming control-driven composite computons with strong structural correctness by construction. We envision that primitives and composites could populate a repository of computational behaviours to be reused across several domains. For example, in a real e-commerce scenario, there could be primitive computons for payment processing and email delivery which can altogether be reused in both travel and healthcare applications. The computon model is general enough to be used not only in software engineering but in any domain requiring computable objects, e.g., it has already been applied in Artificial Intelligence for compositionally forming a Long Short-Term Memory [4], in biomedicine for modelling the oscillatory dynamics of a mdm2-p53 regulatory pathway [3] and in electronic engineering for compositional circuit design [3]. Given the generality of the computon model, several other application domains could benefit from it, e.g., quantum mechanics and linguistics for compositionally describing quantum teleportation and sentence meaning, respectively.

The implemented programming environment currently provides a basis to support cross-domain applications. In our short-term vision, end-users just have to select computons from a global repository to compose more complex computons via the control-driven composition operators presented in Sect. 3. Although our proposal enables computons to be semantically correct by construction, this is not sufficient to achieve correct reusability. For this, we need guarantees of behaviour correctness with respect to some specification, so one can equip computons with correctness proofs before storing them in a global repository. Full computon certification is an aspect we plan to investigate in the future. We particularly believe that the bottom-up composition approach enforced by the computon model could facilitate *a priori* certification [13], i.e., if a computon is guaranteed to meet its specification, it will remain correct even if it ever becomes part of a composite. For example, if computons $\lambda_1$ and $\lambda_2$ are certified, the specification of $\lambda_1 \trianglerighteq_\rho \lambda_2$ will be predictable prior composition, leading to predictable system assembly.

## References


[1] W. M. P. Van der Aalst, K. M. Van Hee, A. H. M. Ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve & M. T. Wynn (2011): *Soundness of workflow nets: classification, decidability, and analysis*. Formal Aspects of Computing 23(3), pp. 333–363, doi:10.1007/s00165-010-0161-4.

[2] Damian Arellanes (2025): *Models of High-Level Computation*. Front. Comput. Sci. 7, pp. 1–7, doi:10.3389/fcomp.2025.1564048.

[3] Damian Arellanes (2026): *Compositional Control-Driven Boolean Circuits*. In: 22nd International Conference on Relational and Algebraic Methods in Computer Science (RAMICS), Springer, pp. 20–40, doi:10.1007/978-3-032-22469-92.

[4] Damian Arellanes (2026): *Compositional Separation of Control Flow and Data Flow*. Journal of Logical and Algebraic Methods in Programming 150:101125, doi:10.1016/j.jlamp.2026.101125.

[5] Damian Arellanes, Kung-Kiu Lau & Rizos Sakellariou (2023): *Decentralized Data Flows for the Functional Scalability of Service-Oriented IoT Systems*. The Computer Journal 66(6), pp. 1477–1506, doi:10.1093/comjnl/bxac023.

[6] J. C. M. Baeten, T. Basten & M. A. Reniers (2009): *Process Algebra: Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, doi:10.1017/CBO9781139195003.

[7] John C. Baez & Jade Master (2020): *Open Petri nets*. Mathematical Structures in Computer Science 30(3), pp. 314–341, doi:10.1017/S0960129520000043.

[8] Filippo Bonchi, Alessandro Di Giorgio & Elena Di Lavore (2025): *A Diagrammatic Algebra for Program Logics*. In Parosh Aziz Abdulla & Delia Kesner, editors: 28th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS), Springer Nature Switzerland, Cham, pp. 308–330, doi:10.1007/978-3-031-90897-2_15.

[9] Edwin Brady (2021): *Idris 2: Quantitative Type Theory in Practice*. In Anders Moller & Manu Sridharan, editors: 35th European Conference on Object-Oriented Programming (ECOOP 2021), Leibniz International Proceedings in Informatics (LIPIcs) 194, Schloss Dagstuhl – Leibniz-Zentrum fur Informatik, Dagstuhl, Germany, pp. 1–26.

[10] Edmund Clarke, Anubhav Gupta, Himanshu Jain & Helmut Veith (2008): *Model Checking: Back and Forth between Hardware and Software*. In Bertrand Meyer & Jim Woodcock, editors: Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 251–255, doi:10.1007/978-3-540-88009-7.

[11] Jean-Louis Colaco, Bruno Pagano & Marc Pouzet (2017): *SCADE 6: A formal language for embedded critical software development (invited paper)*. In: 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 1–11, doi:10.1109/TASE.2017.8285623.

[12] Michele Colledanchise & Petter Ogren (2018): *Behavior Trees in Robotics and AI: An Introduction*, 1st edition edition. CRC Press, Boca Raton London New York, doi:10.1201/9780429489105.

[13] Stefania Costantini (2022): *Ensuring trustworthy and ethical behaviour in intelligent logical agents*. J Logic Computation 32(2), pp. 443–478, doi:10.1093/logcom/exab091.

[14] Matthew Earnshaw & Pawel Sobocinski (2023): *String Diagrammatic Trace Theory*. In Jerome Leroux, Sylvain Lombardy & David Peleg, editors: 48th International Symposium on Mathematical Foundations of Computer Science, 272, pp. 1–15.

[15] David Harel (1987): *Statecharts: a visual formalism for complex systems*. Science of Computer Programming 8(3), pp. 231–274, doi:10.1016/0167-6423(87)90035-9.

[16] Kurt Jensen (1996): *Coloured Petri Nets*. Monographs in Theoretical Computer Science: EATCS Series, Springer, Berlin, Heidelberg, doi:10.1007/978-3-662-03241-1.

[17] G. Kahn (1974): *The Semantics of a Simple Language for Parallel Programming*. North-Holland Publishing Co.

[18] Kung-Kiu Lau, Lily Safie, Petr Stepan & Cuong Tran (2011): *A component model that is both control-driven and data-driven*. In: *14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ACM, New York, NY, USA, pp. 41–50, doi:10.1145/2000229.2000236.

[19] Kung-Kiu Lau, Perla Velasco Elizondo & Zheng Wang (2005): *Exogenous Connectors for Software Components*. In George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens Szyperski & Kurt Wallnau, editors: *8th International Symposium on Component-Based Software Engineering (CBSE 2005)*, Springer, Berlin, Heidelberg, pp. 90–106, doi:10.1007/11424529_7.

[20] Cornelis A. Middelburg (2024): *Imperative Process Algebra and Models of Parallel Computation*. *Theory Comput Syst* 68(3), pp. 529–570, doi:10.1007/s00224-024-10164-0.

[21] Robin Piedeleu & Fabio Zanasi (2025): *An Introduction to String Diagrams for Computer Scientists*, 1st edition. Elements in Applied Category Theory, Cambridge University Press, doi:10.1017/9781009625715.

[22] P. Selinger (2011): *A Survey of Graphical Languages for Monoidal Categories*. In Bob Coecke, editor: *New Structures for Physics*, Springer, Berlin, Heidelberg, pp. 289–355.

[23] Michael Sipser (2013): *Introduction to the Theory of Computation*, 3rd edition. Cengage Learning, Boston, MA.

[24] Stavros Tripakis, Christos Stergiou, Chris Shaver & Edward A. Lee (2013): *A modular formal semantics for Ptolemy*. *Mathematical Structures in Computer Science* 23(4), pp. 834–881, doi:10.1017/S0960129512000278.

[25] Alejandro Villoria, Henning Basold & Alfons Laarman (2025): *Enriching Diagrams with Algebraic Operations*. In: *28th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2024)*, Springer.

[26] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu & Paul Pettersson (2009): *Formal Semantics of the ProCom Real-Time Component Model*. In: *35th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 478–485.

[27] Donald Yau (2018): *Operads of Wiring Diagrams*, 1st edition. Springer, New York, NY, doi:10.1007/978-3-319-95001-3.

# A Proofs

In this appendix, we provide the proofs of all the theorems and propositions presented in the main body of the paper, which were omitted due to space constraints.

**Proposition 1.** *If $\lambda$ is a connected computon, then $U \neq \emptyset$.*

*Proof.* Let $p \in P^+ \cup s(I)$ be a port of a connected computon $\lambda$ so there is a path $(e_1, e_2, \ldots, e_n)$ in the bipartite graph of $\lambda$ where $s'(e_1) = p$. As $e_1 \in O$ would contradict $s'(e_1) = p \in P$, $e_1 \in I$ must hold. Hence, by the totality of $\tau$, $\tau(e_1) \in U$. □

**Proposition 2.** *If $\lambda$ is a primitive computon, $P \cong P^+ \triangle P^-$.*

*Proof.* Assume $|I| = m$ and $|O| = n$ so $|s(I)| = m$ and $|t(O)| = n$ because $s$ and $t$ are total. Now, if we assume for contradiction that $|P^- \cap P^+| > 0$, we know there must be some port $p \in P^- \cap P^+$, i.e., $p \notin s(I) \cup t(O)$ which implies $|P| > |s(I)| + |t(O)|$. As $|P| > m + n$ clearly contradicts the property $|P| =$

$|I|+|O| = m+n$ from Definition 5, we have $|P^- \cap P^+| = 0$. Considering $|P^+| = |P \setminus t(O)| = m+n-n = m$ and $|P^-| = |P \setminus s(I)| = m+n-m = n$, we deduce:

$$|P^+ \triangle P^-| = |P^+| + |P^-| - 2|P^- \cap P^+| = m+n = |I|+|O| = |P|$$

Having $|P^+ \triangle P^-| = |P|$ allow us to conclude $P^+ \triangle P^- \cong P$, as required. □

**Theorem 3.** *If $\lambda_1$ and $\lambda_2$ are computons, there exists a sequentiable span $\rho$ whose pushout is either $\lambda_1 \rhd_\rho \lambda_2$ or $\lambda_1 \trianglerighteq_\rho \lambda_2$.*

*Proof.* Definition 1 states that every computon has at least one inport and at least one outport. So, for any two computons $\lambda_1$ and $\lambda_2$, we can always form a sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ where $\lambda_0$ is a trivial computon with $\alpha_1(P_0) \subseteq P_1^-$ and $\alpha_2(P_0) \subseteq P_2^+$. As the pushability conditions from Definition 10 follow trivially, the pushout of $\rho$ exists which, by Definition 12, must be either $\lambda_1 \rhd_\rho \lambda_2$ or $\lambda_1 \trianglerighteq_\rho \lambda_2$. □

**Lemma 2.** *Assume $\rho$ is a sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. If $\lambda_1 \xrightarrow{\beta_1} \lambda_3 \xleftarrow{\beta_2} \lambda_2$ is the cospan induced by the pushout of $\rho$, then $\beta_1(P_1^+) \subseteq P_3^+$ and $\beta_2(P_2^-) \subseteq P_3^-$.*

*Proof.* We just show $\beta_1(P_1^+) \subseteq P_3^+$ since the proof of $\beta_2(P_2^-) \subseteq P_3^-$ is completely analogous. For this, assume for contrapositive $p_3 \notin P_3^+$ so there is some $o_3 \in O_3$ with $t_3(o_3) = p_3$. As $O_3 = O_1 \sqcup_{O_0} O_2$ according to Definition 9, we have three possible cases:

1. There exclusively is some $o_1 \in O_1$ with $\beta_1(o_1) = o_3$ so, by the commutativity squares of $\beta_1$, $\beta_1(t_1(o_1)) = t_3(\beta_1(o_1)) = t_3(o_3) = p_3$. As $t_1(o_1) \notin P_1^+$ as per Definition 2, $p_3 \notin \beta_1(P_1^+)$.
2. There exclusively is some $o_2 \in O_2$ with $\beta_2(o_2) = o_3$. In this case, the proof is analogous to that of 1.
3. There are $o_1 \in O_1$ and $o_2 \in O_2$ for which $\beta_1(o_1) = o_3 = \beta_2(o_2)$. This case never holds since the apex of $\rho$ is a trivial computon with no flows at all.

Proving that 3 does not hold and that $p_3 \notin P_3^+ \implies p_3 \notin \beta_1(P_1^+)$ for 1 and 2 entail $\beta_1(P_1^+) \subseteq P_3^+$, as required. □

**Lemma 3.** *Assume that the pushout of $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ is a total sequential computon $\lambda_3$. If $\lambda_1 \xrightarrow{\beta_1} \lambda_3 \xleftarrow{\beta_2} \lambda_2$ is the cospan induced by the pushout of $\rho$, then $\beta_1(P_1^+) = P_3^+$ and $\beta_2(P_2^-) = P_3^-$.*

*Proof.* We only prove $\beta_1(P_1^+) = P_3^+$ since the other is completely analogous. For this, let $p_3 \in P_3^+$. Using Lemma 1 and the fact that $P_3$ is given by $P_1 \sqcup_{P_0} P_2$ (see Definition 9), we have three cases:

1. $p_3$ is exclusively identified with $\lambda_1$-ports so $p_3 \in \beta_1(P_1^+)$.
2. $p_3$ is exclusively identified with $\lambda_2$-ports. Here, let $p_2 \in P_2^+$ with $\beta_2(p_2) = p_3$. As $\rho$ satisfies $\alpha_2(P_0) = P_2^+$ (by Definition 12), $p_2 \in P_2^+ \iff p_2 \in \alpha_2(P_0)$. Clearly, a contradiction to the fact that $p_3$ is not identified with any $\lambda_1$-port.
3. there exist $p_1 \in P_1^+$ and $p_2 \in P_2^+$ where $\beta_1(p_1) = p_3 = \beta_2(p_2)$. As $p_1 \in P_1^+$, $p_3 \in \beta_1(P_1^+)$ holds directly.

Disproving 2 and proving $p_3 \in P_3^+ \implies p_3 \in \beta_1(P_1^+)$ for 1 and 3 imply $P_3^+ \subseteq \beta_1(P_1^+)$. Simply using Lemma 2 we conclude $\beta_1(P_1^+) = P_3^+$. □

**Lemma 4.** *If $\lambda_1 \xrightarrow{\beta_1} \lambda_3 \xleftarrow{\beta_2} \lambda_2$ is the cospan induced by the pushout of a sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$, then $\beta_1$ and $\beta_2$ are computon monomorphisms.*

*Proof.* We prove only for $\beta_1$, since the other is analogous. For this, first note that the $\underline{n}$- and $B$-components of $\beta_1$ are always injective because Definition 6 says they are inclusion functions. For the $I$-component, $I_1$-elements are never identified with $I_2$-elements by the fact that $\lambda_0$ is a trivial computon with no flows at all, i.e., $I_0 = \emptyset$. Thus, by Definition 9, $I_3 = I_1 \sqcup_{I_0} I_2$ has an equivalence class for each $I_1$-element, i.e., the $I$-component of $\beta_1$ is injective. The same reasoning holds for the $O$- and $U$-components of $\beta_1$ considering $O_0 = \emptyset = U_0$.

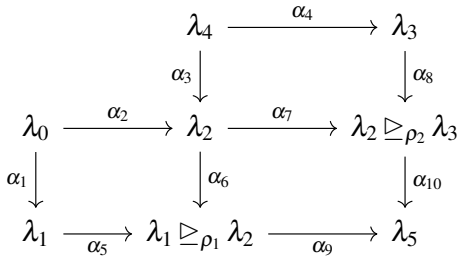
Now, assume for contradiction that the $P$-component of $\beta_1$ is not injective so there are ports $p_1, q_1 \in P_1$ with $\beta_1(p_1) = \beta_1(q_1)$. As $P_3 = P_1 \sqcup_{P_0} P_2$, we have two possible scenarios:

1. There is no $p_2 \in P_2$ with $\beta_2(p_2) = \beta_1(p_1) = \beta_1(q_1)$. In this case, $p_1$ and $q_1$ form individual equivalence classes in $P_3$ for satisfying the reflexivity of the equivalence relation for $P_1 \sqcup_{P_0} P_2$. Thus, contradicting our assumption that $\beta_1(p_1) = \beta_1(q_1)$.
2. There is some $p_2 \in P_2$ where $\beta_2(p_2) = \beta_1(p_1) = \beta_1(q_1)$. In this case, the commutativity property of pushouts entails there are $p_0, q_0 \in P_0$ with $\alpha_1(p_0) = p_1$, $\alpha_1(q_0) = q_1$ and $\alpha_2(p_0) = p_2 = \alpha_2(q_0)$. Clearly contradicting that $\alpha_2$ is injective (see Definition 11).

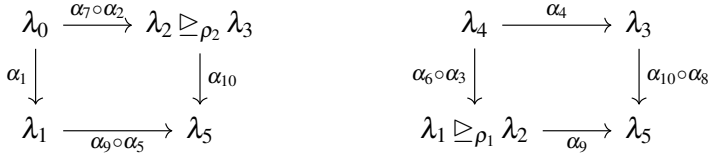
As all its components are injective functions, we conclude that $\beta_1$ is a computon monomorphism, as required. □

**Theorem 4.** *Total sequencing is associative up to isomorphism, but partial sequencing is not. Both total and partial sequencing are not commutative.*

*Proof.* Disproving the commutativity of both total and partial sequencing can be done through the counterexamples presented in [4]. In the same paper, there is a counterexample that can be used to disprove the associativity of partial sequencing. Here, we just prove that total sequencing is associative up to isomorphism. For this, assume that $\lambda_1 \trianglerighteq_{\rho_1} \lambda_2$ and $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$ are total sequential computons with $\rho_1 := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ and $\rho_2 := \lambda_2 \xleftarrow{\alpha_3} \lambda_4 \xrightarrow{\alpha_4} \lambda_3$. So, we have the following commutative diagram:




where the cospans $(\alpha_5, \alpha_6)$ and $(\alpha_7, \alpha_8)$ are induced by the pushouts of $\rho_1$ and $\rho_2$, respectively. Composing computon morphisms horizontally and vertically, we obtain the following commutative diagrams:



To show that every $\lambda_0$-port is mapped to an inport of $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$, consider the following chain of equivalences: $p \in \alpha_7(\alpha_2(P_0)) \iff p \in \alpha_7(P_2^+)$ (because $\alpha_2(P_0) = P_2^+$ by the fact that the pushout of $\rho_1$ is a total sequential computon) $\iff$ $p$ is an inport of $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$ (as per Lemma 3).

With Lemma 4 and the fact that the pushout of $\rho_1$ is total sequential, we have that $\alpha_2$ and $\alpha_7$ are mono so $\alpha_7 \circ \alpha_2$ is also a mono. Having $\alpha_1(P_0) = P_1^-$, because the pushout of $\rho_1$ is total sequential, Definition 11 says that $\rho_3 := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_7 \circ \alpha_2} \lambda_2 \trianglerighteq_{\rho_2} \lambda_3$ must be sequentiable. Here, $\lambda_0$ is the apex computon whilst $\lambda_1$ and $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$ are the left and right operands, respectively. Using Definition 12, we determine that the pushout of $\rho_3$ is $\lambda_1 \trianglerighteq_{\rho_3} (\lambda_2 \trianglerighteq_{\rho_2} \lambda_3)$. As a similar reasoning allows us to deduce that $(\lambda_1 \trianglerighteq_{\rho_1} \lambda_2) \trianglerighteq_{\rho_4} \lambda_3$ is the pushout of the span $\rho_4 := \lambda_1 \trianglerighteq_{\rho_1} \lambda_2 \xleftarrow{\alpha_6 \circ \alpha_3} \lambda_4 \xrightarrow{\alpha_4} \lambda_3$, we conclude $\lambda_1 \trianglerighteq_{\rho_3} (\lambda_2 \trianglerighteq_{\rho_2} \lambda_3) \cong \lambda_5 \cong (\lambda_1 \trianglerighteq_{\rho_1} \lambda_2) \trianglerighteq_{\rho_4} \lambda_3$. □

***Theorem** 6. An async computon $\lambda_1 + \lambda_2$ can always be constructed.*

*Proof.* It is trivial to verify that coproduct can always be computed in a category of computons because such a colimit construction is computed componentwise in the category of finite sets (which has all coproducts). A full proof has been provided in [4]. □

**Theorem** 7. *Asynchronous parallelising is associative and commutative up to isomorphism.*

*Proof.* The proof follows directly from the well-known fact that coproduct in the category of finite sets is both associative and commutative up to isomorphism. □

**Theorem** 10. *Open branching is associative and commutative up to isomorphism.*

*Proof.* Associativity and commutativity follow from the fact that pushout in the category of finite sets and total functions satisfies those properties up to isomorphism. The proof that the pushout of in-markers can be computed in a category of computons is analogous to the proof presented in [4]. □

## B Addendum to Section 5: Coproduct and Pushout Algorithms

In this appendix, we describe the two main algorithms for the construction of pushout-induced functions and coproduct equalities over finite sets, which form the basis for the implementation of colimit constructions in a category of computons.

The first is Algorithm 1 which receives a span $Y \xleftarrow{g} X \xrightarrow{h} Z$ of total functions and incrementally constructs functions $i_Y : Y \to Y \sqcup_X Z$ and $i_Z : Z \to Y \sqcup_X Z$ by initially treating them as partial and updating them at every step. To update such functions, the algorithm first obtains the fiber $g^{-1}(y)$ of each $y \in Y$ and, for each $x \in g^{-1}(y)$, it checks whether $h(x)$ is in the domain of the current (partial) function $i_Z$. If $h(x)$ is in the domain, then $y$ and every element in $g^{-1}(y)$ are mapped to $i_Z(h(x))$; thus, imposing equality. Otherwise, $y$ and its fiber elements are all mapped to the current pushout element (0 initially) which is then increased by one. After analysing all the $Y$-elements, Algorithm 1 finalises the construction of the total function $i_Y$, but $i_Z$ might still be partial because some $Z$-elements might not fall under the image of $h$. To make $i_Z$ total, those previously ignored elements are monotonically assigned to natural numbers in an increasing manner, starting with the latest pushout element. After completing this, Algorithm 1 simply returns the total functions $i_Y$ and $i_Z$ together with the pushout size (i.e., the latest pushout element).

The second is Algorithm 2 which simply adheres to Definition 8 to compute the functions of a coproduct object in the category of finite sets, while considering practical issues related to efficiency. In particular, we avoid backtracking elements to their source by resorting to dynamic programming through

which we define Idris vectors that store values for the actual mappings of coproduct functions. The components of a vector are values obtained through Algorithm 2. For example, the source function $s\colon I_1 \sqcup I_2 \to P_1 \sqcup P_2$ of a coproduct object $\lambda_1 + \lambda_2$ is specified as a vector of size $|I_1| + |I_2|$ where $s(i)$ is given by $\texttt{FindEq}(I_1 \sqcup I_2, P_1 \sqcup P_2, i, s_1, s_2)$.

**Algorithm 1** BuildPushoutFunctions

**Input:** a span $Y \xleftarrow{g} X \xrightarrow{h} Z$ of total functions where $X = [0,m] \cap \mathbb{Z}$, $Y = [0,n] \cap \mathbb{Z}$ and $Z = [0,o] \cap \mathbb{Z}$.
**Output:** $(i_Y, i_Z, p)$ where $i_Y\colon Y \to Y \sqcup_X Z$ and $i_Z\colon Y \to Y \sqcup_X Z$ are total functions and $p \in \mathbb{N}$.

```
i_Y ← ∅
i_Z ← ∅
currentPEl ← 0
for y ∈ Y do
    eq ← currentPEl
    for x ∈ g^{-1}(y) do
        if i_Z(h(x)) exists then
            eq ← i_Z(h(x))
            break
        end if
    end for
    for x ∈ g^{-1}(y) do
        i_Z ← i_Z[h(x) ↦ eq]
    end for
    i_Y ← i_Y[y ↦ eq]
    if eq = currentPEl then
        currentPEl ← currentPEl + 1
    end if
end for
for z ∈ Z \ Dom(i_Z) do
    i_Z ← i_Z[z ↦ currentPEl]
    currentPEl ← currentPEl + 1
end for
return (i_Y, i_Z, currentPEl)
```

**Algorithm 2** FindEq

**Input:** disjoint union $(A_1, A_2, inj_1^A\colon A_1 \to A_1 + A_2, inj_2^A\colon A_2 \to A_1 + A_2)$, disjoint union $(B_1, B_2, inj_1^B\colon B_1 \to B_1 + B_2, inj_2^B\colon B_2 \to B_1 + B_2)$, an element $a \in A_1 + A_2$, and total functions $g_1\colon A_1 \to B_1$ and $g_2\colon A_2 \to B_2$ .
**Output:** an element $b \in B_1 + B_2$.

```
if a = inj_1^A(a_1) for some a_1 ∈ A_1 then
    return inj_1^B(g_1(a_1))
else if a = inj_2^A(a_2) for some a_2 ∈ A_2 then
    return inj_2^B(g_2(a_2))
end if
```

For convention, for a coproduct `Fin x+y`, the injection `Fin x` $\to$ `Fin x+y` assigns all the $x$-elements to themselves, whereas `Fin y` $\to$ `Fin x+y` maps all the $y$-elements from $x$ to $x+(y-1)$. Thus, yielding a deterministic procedure akin to tagging elements in canonical coproducts. We do not tag explicitly since this would unnecessarily complicate the type of finite sets. Tagging information can be retrieved from the canonical injective morphisms when needed, as shown in Algorithm 2.

Computon pushouts are computed in a similar manner but using Algorithm 2 over pushout constructions, e.g., the source function $s\colon I_1 \sqcup_{I_0} I_2 \to P_1 \sqcup_{P_0} P_2$ of a pushout object $\lambda_1 +_{\lambda_0} \lambda_2$ is an Idris vector of size $|I_1 \sqcup_{I_0} I_2|$ where $s(i)$ is given by `FindEqPush`$(I_1 \sqcup_{I_0} I_2, P_1 \sqcup_{P_0} P_2, i, s_1, s_2)$.

# C Addendum to Section 5: Key Constructs of the Programming Environment Implemented in Idris 2

In this section, we describe the key constructs we implemented in Idris for the realisation of the categorical semantics of the computon model. The complete source code of our implementation is publicly available at https://github.com/damianarellanes/computons-idris and is ready to be used for building complex computing devices with strong structural correctness by construction. The idea is that end-users (e.g., software developers) just define spans of computon morphisms built out of trivial and primitive computons, before using the environment's support to automatically construct composite computons. Such support is provided in the form of Idris functions that realise the colimit behaviour of the elementary composition operators presented in Sect. 3. To differentiate between builtin and implemented types, we use different colours: green and orange, respectively. Keywords are just displayed in blue.

The first important construct, displayed in Listing 1, is the record that captures the essence of Definition 1, which relies on builtin types to represent vectors, lists and strings. Such a record uses the `integerLessThanNat` function, provided by the `Data.Fin` module, to express proofs of inequality between natural numbers. As such a function returns a boolean value rather than an actual proof, we resort to the dependent type `So` from the `Data.So` module, in order to turn a `Bool` value (known to be true) into an explicit proof object.

Listing 1: Data type for computons.

```
record Computon where
    constructor MkComputon
    u : Nat
    p : Nat
    i : Nat
    o : Nat
    n : Nat
    s : Fin i -> Fin p
    t : Fin o -> Fin p
    sigma : Fin o -> Fin u
    tau : Fin i -> Fin u
    c : Fin p -> Fin n
    r : Fin i -> Fin o
    f : Fin o -> String
    inLabels : Vect u (List (Fin p, String))
    pNotEmpty : So (integerLessThanNat 0 p)
    nNotEmpty : So (integerLessThanNat 0 n)
```

As colimits may rename ports by introducing equalities, port identifiers can change upon composition. To avoid the inefficient process of backtracking over colimit-induced morphisms until finding

original port identifiers, Listing 1 reveals that computons are equipped with the `inLabels` field which is updated with corresponding equalities, i.e., `inLabels` is used to keep track of port identifiers. Port identifiers are needed to tell computing devices where to read data from.

Both trivial and primitive computons are specified on top of Listing 1, i.e., they correspond to the records shown in Listings 2 and 3 which adhere to Definitions 4 and 5, correspondingly. In particular, primitives require injectivity proofs for which we use the `Injective` interface from the `Control.Function` module. For surjectivity proofs, we provide the `Surjective` function shown in Listing 3.

Listing 2: Data type for trivial computons.

```
record Trivial where
    constructor MkTrivial
    obj : Computon
    {auto uEmpty : obj.u = 0}
    {auto iEmpty : obj.i = 0}
    {auto oEmpty : obj.o = 0}
```

Listing 3: Data type for primitive computons.

```
record Primitive where
    constructor MkPrimitive
    obj : Computon
    sInjective : Injective obj.s
    tInjective : Injective obj.t
    sigmaSurjective : Surjective obj.sigma
    tauSurjective : Surjective obj.tau
    rSurjective : Surjective obj.r
    onlyInterfacePorts : obj.i + obj.o = obj.p
    {auto unity : obj.u = 1}
Surjective : {a, b : Type} -> (a -> b) -> Type
Surjective f = (b : b) -> (a : a ** f a = b)
```

With Listing 1, we specify a record to capture the notion of a computon morphism, as prescribed by Definition 6. Listing 4 shows that this data type relies on `Subset`, a function we implemented to express proofs of finite set inclusion. This function relies on the builtin `Elem` type from the `Data.List` module. For specifying boundary conditions, we implemented the functions `inports` and `outports` which respectively construct the sets $P^+$ and $P^-$ of a computon $\lambda$. We also implemented `vecI` and `vecO` to compute the sets $\vec{i}(\alpha)$ and $\vec{o}(\alpha)$ from the $U$- and $P$-components of some computon morphism $\alpha$. Listing 5 shows that the notion of a monomorphism is also a record built on top of Listing 4. This record requires proofs of injectivity for all the components of a computon morphism, excluding the `morphism.mf` field which already is injective as per the field `morphism.mfIncl`.

Listing 4: Data type for computon morphisms.

```
record Morphism (comp1, comp2 : Computon) where
    constructor MkMorphism
    mu : Fin comp1.u -> Fin comp2.u
    mp : Fin comp1.p -> Fin comp2.p
    mi : Fin comp1.i -> Fin comp2.i
    mo : Fin comp1.o -> Fin comp2.o
    mn : Fin comp1.n -> Fin comp2.n
    mf : String -> String
```

```
    mfIncl : (str : String) -> mf str = str
    boundaryCond : Subset
        (union (vecI comp1 comp2 mu mp) (vecO comp1 comp2 mu mp))
        (union (inports comp1) (outports comp1))
Subset : List a -> List a -> Type
Subset [] ys = ()
Subset (x :: xs) ys = (Elem x ys, Subset xs ys)
```

Listing 5: Data type for computon monomorphisms.

```
record Monomorphism (comp1, comp2 : Computon) where
    constructor MkMonomorphism
    morphism : Morphism comp1 comp2
    muInj : Injective morphism.mu
    mpInj : Injective morphism.mp
    miInj : Injective morphism.mi
    moInj : Injective morphism.mo
    mnInj : Injective morphism.mn
```

Marker morphisms are built upon the data type shown in Listing 5 since they have to be mono according to Definition 7. Their notion is captured by the records displayed in Listing 6 which, in addition to the proofs of boundary equality, require a proof that the types of the domain computon (i.e., the trivial computon) is less or equal to the number of types in the codomain computon. For this purpose, we use the `LTE` type from the `Data.Nat` module.

Listing 6: Data type for marker morphisms.

```
record InMarker (comp1 : Trivial) (comp2 : Computon) where
    constructor MkInMarker
    mono : Monomorphism comp1.obj comp2
    inportsPf : image (mono.morphism.mp) = inports comp2
    nPf : LTE comp1.obj.n comp2.n
record OutMarker (comp1 : Trivial) (comp2 : Computon) where
    constructor MkOutMarker
    mono : Monomorphism comp1.obj comp2
    outportsPf : image (mono.morphism.mp) = outports comp2
    nPf : LTE comp1.obj.n comp2.n
```

With the constructs shown in Listings 1-6, we implemented factories that abstract away complex details to allow developers focus on composition tasks. Such factories, shown in Listing 7, leverage Idris's capabilities of proof search as well as the `integerLessThanNat` function to minimise the amount of explicit proofs that developers need to provide explicitly. The factories `mkPrimitive`, `mkComputonMorphism`, `mkComputonMonomorphism`, `mkInMarker` and `mkOutMarker` may fail since their implementation automatically decides some properties (e.g., the boundary conditions from Definition 6) in order to further minimise proof tasks for developers.

Listing 7: Signatures of the factories provided by our implemented environment.

```
mkTrivial : {p, n : Nat}
    -> {auto pPf : So(integerLessThanNat 0 p)}
    -> {auto nPf : So(integerLessThanNat 0 n)}
    -> (Fin p -> Fin n)
    -> Trivial
```

```
mkPrimitive : {p1, i1, o1, n1 : Nat}
    -> {auto onlyInterfacePorts : i1 + o1 = p1}
    -> {auto pPf : So(integerLessThanNat 0 p1)}
    -> {auto nPf : So(integerLessThanNat 0 n1)}
    -> (s : Fin i1 -> Fin p1) -> (t : Fin o1 -> Fin p1)
    -> (sigma : Fin o1 -> Fin 1) -> (tau : Fin i1 -> Fin 1)
    -> (c : Fin p1 -> Fin n1)
    -> (r : Fin i1 -> Fin o1)
    -> (f : Fin o1 -> String)
    -> Injective s -> Injective t
    -> Surjective sigma -> Surjective tau -> Surjective r
    -> Either String Primitive
mkComputonMorphism : (comp1, comp2 : Computon)
    -> (mu : Fin comp1.u -> Fin comp2.u)
    -> (mp : Fin comp1.p -> Fin comp2.p)
    -> (mi : Fin comp1.i -> Fin comp2.i)
    -> (mo : Fin comp1.o -> Fin comp2.o)
    -> (mn : Fin comp1.n -> Fin comp2.n)
    -> Maybe (Morphism comp1 comp2)
mkComputonMonomorphism : (comp1, comp2 : Computon)
    -> (mu : (Fin comp1.u -> Fin comp2.u))
    -> (mp : (Fin comp1.p -> Fin comp2.p))
    -> (mi : (Fin comp1.i -> Fin comp2.i))
    -> (mo : (Fin comp1.o -> Fin comp2.o))
    -> (mn : (Fin comp1.n -> Fin comp2.n))
    -> Injective mu -> Injective mp -> Injective mi
    -> Injective mo -> Injective mn
    -> Maybe (Monomorphism comp1 comp2)
mkInMarker : (comp1 : Trivial) -> (comp2 : Computon)
    -> (mp : (Fin comp1.obj.p -> Fin comp2.p))
    -> (mn : (Fin comp1.obj.n -> Fin comp2.n))
    -> Injective mp -> Injective mn
    -> image mp = inports comp2
    -> Maybe (InMarker comp1 comp2)
mkOutMarker : (comp1 : Trivial) -> (comp2 : Computon)
    -> (mp : (Fin comp1.obj.p -> Fin comp2.p))
    -> (mn : (Fin comp1.obj.n -> Fin comp2.n))
    -> Injective mp -> Injective mn
    -> image mp = outports comp2
    -> Maybe (OutMarker comp1 comp2)
```

To abstract away details related to the computation of colimits in a category of computons, we implemented the notions of span, coproduct and pushout in the form of the records shown in Listing 8. These constructions form the basis for the implementation of Idris functions that enable the actual computation of pushouts, coproducts and the universal property of coproducts.

Listing 8: Data types for colimit constructions in a category of computons.

```
record Span where
    constructor MkSpan
    apex : Computon
    base1 : Computon
```

```
    base2 : Computon
    leg1 : Morphism apex base1
    leg2 : Morphism apex base2
record Coproduct (u, p, i, o, n : Nat) where
    constructor MkCoproduct
    coapex : Computon
    cobase1 : Computon
    cobase2 : Computon
    coleg1 : Morphism cobase1 coapex
    coleg2 : Morphism cobase2 coapex
    uPf : u = coapex.u
    pPf : p = coapex.p
    iPf : i = coapex.i
    oPf : o = coapex.o
    nPf : n = coapex.n
record Pushout where
    constructor MkPushout
    coapex : Computon
    cobase1 : Computon
    cobase2 : Computon
    coleg1 : Morphism cobase1 coapex
    coleg2 : Morphism cobase2 coapex
```

Listing 8 particularly shows that `Span` has fields for the apex computon as well as the two computon bases and the two legs from the apex. `Pushout` is similar, with the difference that legs go from the cobases to the coapex. The `Coproduct` record requires arguments for the size of computation units, ports, inflows, outflows and types, in addition to the coapex, the two colegs and the two cobases. It also carries a proof that the arguments are definitionally equal to the corresponding components of the coapex. These proofs are necessary for computing unique morphisms derived from the universal property of coproducts for which we provide the `uniqueFromCoproduct` function displayed in Listing 9. Listing 9 shows the signatures of the functions to compute coproducts and pushouts in a category of computons. Particularly, `computeCoproduct` relies on the function `maximum`, provided by the `Data.Nat` module, for determining the cardinality of the set of types of a coproduct object, whereas `computePushout` just requires a span for its operation.

Listing 9: Signatures of the Idris functions to compute coproducts, the universal property of coproducts and pushouts in a category of computons.

```
computeCoproduct : (comp1, comp2 : Computon)
    -> Coproduct (comp1.u + comp2.u) (comp1.p + comp2.p)
        (comp1.i + comp2.i) (comp1.o + comp2.o)
        (maximum comp1.n comp2.n)
uniqueFromCoproduct : (comp1, comp2, comp3 : Computon)
    -> Coproduct (comp1.u + comp2.u) (comp1.p + comp2.p)
        (comp1.i + comp2.i) (comp1.o + comp2.o)
        (maximum comp1.n comp2.n)
    -> Morphism comp1 comp3
    -> Morphism comp2 comp3
    -> LTE (maximum comp1.n comp2.n) comp3.n
    -> Morphism copr.coapex comp3
computePushout : Span -> Pushout
```

To finalise this section, Listing 10 shows the signatures of the Idris functions that capture the behaviour of the elementary composition operators described in Sect. 3, namely partial and total sequencing (`SEQ`), asynchronous parallelising (`P_ASYNC`), closed branching (`BRA_CLOSED`) and open branching (`BRA_OPEN`). The synchronous parallelising operator was not explicitly implemented since this can be built out of `P_ASYNC` and `SEQ` (as discussed in Sect. 3.2), i.e., it is not elementary. A glance at Listing 10 shows that `BRA_CLOSED` requires a proof that the operands are connected in the sense of Definition 3. For this, we provide the function `isConnected` which algorithmically returns a boolean value whenever such a property is satisfied. Connectivity is raised to the level of proof types by the use of the `So` type.

Listing 10: Signatures of the colimit-based, control-driven composition operators described in Section 3.

```
SEQ : (comp0 : Trivial) -> (comp1, comp2 : Computon)
    -> (alpha1 : Monomorphism comp0.obj comp1)
    -> (alpha2 : Monomorphism comp0.obj comp2)
    -> Subset (image alpha1.morphism.mp) (outports comp1)
    -> Subset (image alpha2.morphism.mp) (inports comp2)
    -> Computon
P_ASYNC : Computon -> Computon -> Computon
BRA_CLOSED : (comp0, comp1 : Trivial) -> (comp2, comp3 : Computon)
    -> InMarker comp0 comp2 -> InMarker comp0 comp3
    -> OutMarker comp1 comp2 -> OutMarker comp1 comp3
    -> So (isConnected comp2) -> So (isConnected comp3)
    -> Computon
BRA_OPEN : (comp0 : Trivial) -> (comp1, comp2 : Computon)
    -> InMarker comp0 comp1 -> InMarker comp0 comp2
    -> Computon
```

Although `computePushout` operates on arbitrary spans that can even be non-pushable, we do not expect end-users to explicitly invoke such a function. Instead, they are expected to directly use the operators provided in Listing 10. This facade-based simplification derives from the fact that all the operators built upon pushouts do not require explicit proofs for the satisfaction of Definition 10. In the case of `BRA_CLOSED` and `BRA_OPEN`, pushability follows from Theorem 1, whereas for `SEQ` such a property is satisfied by Theorem 2.