

# The Denotational and Static Semantics of a Domain-Specific Language for Flow-Network Design

Assaf Kfoury

Boston University

**Abstract.** *Flow networks* are inductively defined, assembled from small *network modules* to produce arbitrarily large ones, with interchangeable and expanding functionally-equivalent parts. We carry out this induction formally using a *domain-specific language* (DSL). Associated with our DSL is a *typing system* (or *static semantics*), a system of formal annotations that enforce desirable properties of flow networks as *invariants* across their interfaces. A prerequisite for a type theory is a *formal semantics*, *i.e.*, a rigorous definition of the entities that qualify as feasible flows through the networks, possibly restricted to satisfy additional efficiency or safety requirements. We carry out this via a *denotational semantics*.

## 1 Introduction and Motivation

**Flow Networks.** Many large-scale, safety-critical systems can be viewed as interconnections of subsystems, or modules, each of which is a producer, consumer, or regulator of *flows*. These flows are characterized by a set of variables and a set of constraints thereof, reflecting *inherent* or *assumed* properties or rules governing how the modules operate and what constitutes safe operation. Our notion of flow encompasses streams of physical entities (*e.g.*, vehicles on a road, fluid in a pipe), data objects (*e.g.*, sensor network packets, video frames), or consumable resources (*e.g.*, electric energy, compute cycles).

Traditionally, the design and implementation of such *flow networks* follows a bottom-up approach, enabling system designers to certify desirable safety invariants of the system as a whole: Properties of the full system depend on a complete determination of the underlying properties of all subsystems. For example, the development of real-time applications necessitates the use of real-time kernels so that timing properties at the application layer (top) can be established through knowledge and/or tweaking of much lower-level system details (bottom), such as worst-case execution or context-switching times [DL97, LBJ<sup>+</sup>95, Reg02], specific scheduling and power parameters [AMM01, PLS01, SLM98, Sta00], among many others.

While justifiable in some instances, this vertical approach does not lend itself well to emerging practices in the assembly of complex large-scale systems – namely, the integration of various subsystems into a whole by “system integrators” who may not possess the requisite expertise or knowledge of the internals of these subsystems [KBS04]. This latter alternative can be viewed as a *horizontal* and *incremental* approach to system design and implementation, which has significant merits with respect to scalability and modularity. However, it also poses a major and largely unmet challenge with respect to verifiable trustworthiness – namely, how to formally certify that the system as a whole will satisfy specific safety invariants and to determine formal conditions under which it will remain so, as it is augmented, modified, or subjected to local component failures.

**Incremental and Modular Design.** Several approaches to system design, modeling and analysis have been proposed in recent years, overlapping with our notion of flow networks. Apart from the differences in the technical details – at the level of formalisms and mathematics that

are brought to bear – our approach distinguishes itself from the others by incorporating from its inception three inter-related features/goals: (A) the ability to pursue system design and analysis without having to wait for missing (or broken) components to be inserted (or replaced), (B) the ability to abstract away details through the retention of only the salient variables and constraints at network interfaces as we transition from smaller to larger networks, and (C) the ability to leverage diverse, unrelated theories to derive properties of components and small networks, as long as such networks share a common language at their interfaces – a strongly-typed Domain-Specific Language (DSL) that enables assembly and analysis that is agnostic to components’ internal details and to theories used to derive properties at their interfaces.

Our DSL provides two primitive constructors, one is of the form  $(\mathcal{M}_1 \parallel \mathcal{M}_2)$  and the other of the form **bind**  $(\mathcal{N}, \langle a, b \rangle)$ . The first juxtaposes two networks  $\mathcal{M}_1$  and  $\mathcal{M}_2$  in parallel, and the second binds an output port  $a$  to an input port  $b$  in a network  $\mathcal{N}$ . With these two primitive constructors, we define others as *derived* and according to need. A distinctive feature of our DSL is the presence of *holes* in network specifications, together with constructs of the form: **(let**  $X = \mathcal{M}$  **in**  $\mathcal{N}$ ), which says “network  $\mathcal{M}$  may be safely placed in the occurrences of hole  $X$  in network  $\mathcal{N}$ ”. What “safely” means, depends on the invariant properties that typings are formulated to enforce. There are other useful constructs involving holes which we discuss later in the paper.

**Types and Formal Semantics.** Associated with our DSL is a *type theory*, a system of formal annotations to express desirable properties of flow networks together with rules that enforce them as *invariants* across their interfaces, *i.e.*, the rules guarantee the properties are preserved as we build larger networks from smaller ones.

A prerequisite for a type theory is a *formal semantics*, *i.e.*, a rigorous definition of the entities that qualify as safe flows through the networks. There are standard approaches which can be extended to our DSL, one producing a *denotational* semantics and another an *operational* semantics. In the first approach, a safe flow through the network is denoted by a function, and the semantics of the network is the set of all such functions. In the second approach, the network is uniquely rewritten to another network in *normal form* (appropriately defined), and the semantics of the network is its normal form or directly extracted from it. Although it is possible to prove the equivalence of the two approaches, we choose the denotational approach to avoid an exponential growth in the size of network specifications when rewritten to normal form in the operational approach. We thus prove the soundness of the typing system (“a type-safe network construction guarantees that flows through the network satisfy the invariants properties enforced by types”) without having to explicitly carry out exponential-growth rewriting.

**Limited Notion of Safety.** For illustrative purposes only, we consider a single basic *safety* property – namely, to be *safe*, a flow must satisfy (1) linear constraints of *flow conservation* at nodes/hubs and (2) linear *capacity constraints* that restrict the range of permissible values along links/connections between nodes/hubs. Types and typings are then formulated precisely to enforce this kind of safety across interfaces.

In the full report [Kfo11] from which this paper is extracted, we consider further safety restrictions on flows. Specifically, to be *safe*, a flow must additionally satisfy an *objective function* that minimizes (or maximizes) some quantity. Such an objective function may be the *minimization of hop routing* (a “minimal hop route” being one with minimum number of links) or *minimization of arc utilization* (the “utilization of a link” being the ratio of the flow value at the link over the upper-bound allowed at the link). For other objective functions which can be examined in our type-theoretic framework, the reader is referred to [Kfo11], all inspired by studies in the area of “traffic engineering” (see, *e.g.*, [BL06] and references therein).

**Paper Organization.** Section 2 is devoted to preliminary definitions. Section 3 introduces the syntax of our DSL and lays out several conditions for the well-formedness of network

specifications written in it. We only include the **let-in** constructor in this report, delaying the treatment of **try-in**, **mix-in**, and others involving holes to a follow-up report.

The formal semantics of flow networks are introduced in Section 4 and a corresponding type theory is presented in the remaining sections. The type theory is syntax-directed, and therefore *modular*, as it infers or assigns typings to objects in a stepwise inside-out manner. If the order in which typings are inferred for the constituent parts does not matter, we additionally say that the theory is *fully compositional*. We add the qualifier “fully” to distinguish our notion of compositionality from similar, but different, notions in other areas of computer science.<sup>1</sup> We only include an examination of modular typing inference in this paper, leaving its (more elaborate) fully-compositional version to a follow-up report.

For lack of space, we omit all proofs in this paper, as well as all issues related to the complexity of our algorithms. All the proofs can be found in the full report [Kfo11].

**Acknowledgment.** The work reported in this paper is a fraction of a collective effort involving several people, under the umbrella of the **iBench Initiative** at Boston University. The website <https://sites.google.com/site/ibenchbu/> gives a list of other research activities. The DSL in this paper, with its formal semantics and type system, is a specialized version of a DSL we introduced earlier in our work for NetSketch, an integrated environment for the modeling, design and analysis of large-scale safety-critical systems with interchangeable parts [BKLO09, BKLO10, SBKL11]. In addition to its DSL, NetSketch has two other components currently under development: an automated verifier (AV), and a user interface (UI) that combines the DSL and the AV and adds appropriate tools for convenient interactive operation.

## 2 Preliminary Definitions

A *small network*  $\mathcal{A}$  is of the form  $\mathcal{A} = (\mathbf{N}, \mathbf{A})$  where  $\mathbf{N}$  is a set of nodes and  $\mathbf{A}$  a set of directed arcs. Capacities on arcs are determined by a lower-bound  $L : \mathbf{A} \rightarrow \mathbb{R}^+$  and an upper-bound  $U : \mathbf{A} \rightarrow \mathbb{R}^+$  satisfying the conditions  $L(a) \leq U(a)$  for every  $a \in \mathbf{A}$ . We write  $\mathbb{R}$  and  $\mathbb{R}^+$  for the sets of all reals and all non-negative reals, respectively. We identify the two ends of an arc  $a \in \mathbf{A}$  by writing  $head(a)$  and  $tail(a)$ , with the understanding that flow moves from  $tail(a)$  to  $head(a)$ . The set  $\mathbf{A}$  of arcs is the disjoint union (denoted “ $\uplus$ ”) of three sets: the set  $\mathbf{A}_\#$  of internal arcs, the set  $\mathbf{A}_{in}$  of input arcs, and the set  $\mathbf{A}_{out}$  of output arcs:

$$\begin{aligned} \mathbf{A} &= \mathbf{A}_\# \uplus \mathbf{A}_{in} \uplus \mathbf{A}_{out} \quad \text{where} \\ \mathbf{A}_\# &= \{ a \in \mathbf{A} \mid head(a) \in \mathbf{N} \text{ and } tail(a) \in \mathbf{N} \} \\ \mathbf{A}_{in} &= \{ a \in \mathbf{A} \mid head(a) \in \mathbf{N} \text{ and } tail(a) \notin \mathbf{N} \} \\ \mathbf{A}_{out} &= \{ a \in \mathbf{A} \mid head(a) \notin \mathbf{N} \text{ and } tail(a) \in \mathbf{N} \} \end{aligned}$$

The tail of an input arc, and the head of an output arc, are not attached to any node. We do not assume  $\mathcal{A}$  is connected as a directed graph. We assume  $\mathbf{N} \neq \emptyset$ , *i.e.*, there is at least one node in  $\mathbf{N}$ , without which there would be no input/output arc and nothing to say.

A *flow*  $f$  in  $\mathcal{A}$  is a function that assigns a non-negative real to every  $a \in \mathbf{A}$ . Formally, a flow  $f : \mathbf{A} \rightarrow \mathbb{R}^+$ , if *feasible*, satisfies “flow conservation” and “capacity constraints” (below).

We call a bounded interval  $[r, r']$  of reals, possibly negative, a *type*. A *typing* is a function  $T$  that assigns a type to every subset of input and output arcs. Formally,  $T$  is of the form:<sup>2</sup>

$$T : \mathcal{P}(\mathbf{A}_{in} \cup \mathbf{A}_{out}) \rightarrow \mathbb{R} \times \mathbb{R}$$

<sup>1</sup> Adding to the imprecision of the word, “compositional” in the literature is sometimes used in the more restrictive sense of “modular” in our sense.

<sup>2</sup> Our notion of a “typing” as an assignment of types to the members of a powerset is different from a similarly-named notion in the study of type systems elsewhere. In the latter, a typing refers to a

where  $\mathcal{P}(\cdot)$  is the power-set operator, *i.e.*,  $\mathcal{P}(\mathbf{A}_{in} \cup \mathbf{A}_{out}) = \{A \mid A \subseteq \mathbf{A}_{in} \cup \mathbf{A}_{out}\}$ . As a function,  $T$  is not totally arbitrary and satisfies certain conditions, discussed in Section 5, which qualify it as a *network typing*. We write  $T(A) = [r, r']$  instead of  $T(A) = \langle r, r' \rangle$ , where  $A \subseteq \mathbf{A}_{in} \cup \mathbf{A}_{out}$ . We do not disallow the case  $r > r'$  which is an empty type satisfied by no flow.

Informally, a typing  $T$  imposes restrictions on a flow  $f$  relative to every  $A \subseteq \mathbf{A}_{in} \cup \mathbf{A}_{out}$  which, if satisfied, will guarantee that  $f$  is feasible. Specifically, if  $T(A) = [r, r']$ , then  $T$  requires that the part of  $f$  entering through the arcs in  $A \cap \mathbf{A}_{in}$  minus the part of  $f$  exiting through the arcs in  $A \cap \mathbf{A}_{out}$  must be within the interval  $[r, r']$ .

**Flow Conservation, Capacity Constraints, Type Satisfaction.** Though obvious, we precisely state fundamental concepts underlying our entire examination and introduce some of our notational conventions, in Definitions 1, 2, 3, and 4.

**Definition 1.** *If  $A$  is a subset of arcs in  $\mathcal{A}$  and  $f$  a flow in  $\mathcal{A}$ , we write  $\sum f(A)$  to denote the sum of flows assigned to all arcs in  $A$ :  $\sum f(A) = \sum \{f(a) \mid a \in A\}$ . By convention,  $\sum \emptyset = 0$ . If  $A = \{a_1, \dots, a_p\}$  is the set of all arcs entering node  $n$ , and  $B = \{b_1, \dots, b_q\}$  is the set of all arcs exiting node  $n$ , then conservation of flow at  $n$  is expressed by the linear equation:*

$$(1) \quad \sum f(A) = \sum f(B) \quad (\text{one such equation for every node } n \in \mathbf{N})$$

**Definition 2.** *A flow  $f$  satisfies the capacity constraints at arc  $a \in \mathbf{A}$  if:*

$$(2) \quad L(a) \leq f(a) \leq U(a) \quad (\text{two such inequalities for every arc } a \in \mathbf{A})$$

**Definition 3.** *A flow  $f$  is feasible iff two conditions:*

- for every node  $n \in \mathbf{N}$ , the equation in (1) is satisfied,
- for every arc  $a \in \mathbf{A}$ , the two inequalities in (2) are satisfied.

**Definition 4.** *Let  $T : \mathcal{P}(\mathbf{A}_{in} \cup \mathbf{A}_{out}) \rightarrow \mathbb{R} \times \mathbb{R}$  be a typing for the small network  $\mathcal{A}$ . We say the flow  $f$  satisfies  $T$  if, for every  $A \in \mathcal{P}(\mathbf{A}_{in} \cup \mathbf{A}_{out})$  with  $T(A) = [r, r']$ , it is the case:*

$$(3) \quad r \leq \sum f(A \cap \mathbf{A}_{in}) - \sum f(A \cap \mathbf{A}_{out}) \leq r'$$

### 3 DSL for Incremental and Modular Flow-Network Design (Untyped)

The definition of small networks in Sect. 2 is less general than our full definition of networks, but it has the advantage of being directly comparable to standard graph-theoretic definitions. Our networks in general involve what we call “holes”. A *hole*  $X$  is a pair  $(\mathbf{A}_{in}, \mathbf{A}_{out})$  where  $\mathbf{A}_{in}$  and  $\mathbf{A}_{out}$  are finite sets of input and output arcs. A hole  $X$  is a place holder where networks can be inserted, provided the *matching-dimensions* condition (in Sect. 3.2) is satisfied.

We use a BNF definition to generate formal expressions, each being a formal description of a network. Such an expression may involve subexpressions of the form: **let**  $X = \mathcal{M}$  **in**  $\mathcal{N}$ , which informally says “ $\mathcal{M}$  may be safely placed in the occurrences of hole  $X$  in  $\mathcal{N}$ ”. What “safely” means depends on the invariant properties that typings are formulated to enforce.

If  $\mathcal{A} = (\mathbf{N}, \mathbf{A})$  is a small network where  $\mathbf{A} = \mathbf{A}_{\#} \uplus \mathbf{A}_{in} \uplus \mathbf{A}_{out}$ , let  $\mathbf{in}(\mathcal{A}) = \mathbf{A}_{in}$ ,  $\mathbf{out}(\mathcal{A}) = \mathbf{A}_{out}$ , and  $\#(\mathcal{A}) = \mathbf{A}_{\#}$ . Similarly, if  $X = (\mathbf{A}_{in}, \mathbf{A}_{out})$  is a hole, let  $\mathbf{in}(X) = \mathbf{A}_{in}$ ,  $\mathbf{out}(X) =$

---

derivable “typing judgment” consisting of a program expression  $M$ , a type assigned to  $M$ , and a type environment with a type for every free variable in  $M$ .

$\mathbf{A}_{\text{out}}$ , and  $\#(X) = \emptyset$ . We assume the arc names of small networks and holes are all pairwise disjoint, *i.e.*, every small network and every hole has its own private set of arc names.

The formal expressions generated by our BNF are built up from: the set of names for small networks and the set of names for holes, using the constructors  $\parallel$ , **let**, and **bind**:

$\mathcal{A}, \mathcal{B}, \mathcal{C}$	$\in$ SMALLNETWORKS	
$X, Y, Z$	$\in$ HOLENAMES	
$\mathcal{M}, \mathcal{N}, \mathcal{P}$	$\in$ NETWORKS	$::= \mathcal{A}$ small network name
		$X$ hole name
		$\mathcal{M} \parallel \mathcal{N}$ parallel connection
		<b>let</b> $X = \mathcal{M}$ <b>in</b> $\mathcal{N}$ let-binding of hole $X$
		<b>bind</b> $(\mathcal{N}, \langle a, b \rangle)$ bind <i>head</i> ( $a$ ) to <i>tail</i> ( $b$ ), where $\langle a, b \rangle \in \mathbf{out}(\mathcal{N}) \times \mathbf{in}(\mathcal{N})$

where  $\mathbf{in}(\mathcal{N})$  and  $\mathbf{out}(\mathcal{N})$  are the input and output arcs of  $\mathcal{N}$ . In the full report [Kfo11], we define  $\mathbf{in}(\mathcal{N})$  and  $\mathbf{out}(\mathcal{N})$ , as well as the set  $\#(\mathcal{N})$  of internal arcs, by induction on the structure of  $\mathcal{N}$ . We say  $\mathcal{N}$  is *closed* if every hole  $X$  in  $\mathcal{N}$  is bound.

### 3.1 Derived Constructors

From the three constructors already introduced, namely:  $\parallel$ , **let**, and **bind**, we can define several other constructors. Below, we present four of these derived constructors precisely, and mention several others in Remark 1. Our four derived constructors are used as in the following expressions, where  $\mathcal{N}, \mathcal{N}_i$ , and  $\mathcal{M}_j$ , are network specifications and  $\theta$  is set of arc pairs:

$$\mathbf{bind}(\mathcal{N}, \theta) \quad \mathbf{conn}(\mathcal{N}_1, \mathcal{N}_2, \theta) \quad \mathcal{N}_1 \oplus \mathcal{N}_2 \quad \mathbf{let} X \in \{\mathcal{M}_1, \dots, \mathcal{M}_n\} \mathbf{in} \mathcal{N}$$

The second above depends on the first, the third on the second, and the fourth is independent of the three preceding it. Let  $\mathcal{N}$  be a network specification. We write  $\theta \subseteq_{1-1} \mathbf{out}(\mathcal{N}) \times \mathbf{in}(\mathcal{N})$  to denote a partial one-one map from  $\mathbf{out}(\mathcal{N})$  to  $\mathbf{in}(\mathcal{N})$ . We may write the entries in  $\theta$  explicitly, as in:  $\theta = \{\langle a_1, b_1 \rangle, \dots, \langle a_k, b_k \rangle\}$  where  $\{a_1, \dots, a_k\} \subseteq \mathbf{out}(\mathcal{N})$  and  $\{b_1, \dots, b_k\} \subseteq \mathbf{in}(\mathcal{N})$ .

Our first derived constructor generalizes **bind** and uses the same name. In this generalization of **bind** the second argument is now  $\theta$  rather than a single pair  $\langle a, b \rangle \in \mathbf{out}(\mathcal{N}) \times \mathbf{in}(\mathcal{N})$ . The expression **bind**  $(\mathcal{N}, \theta)$  is expanded as follows:

$$\mathbf{bind}(\mathcal{N}, \theta) \implies \mathbf{bind}(\mathbf{bind}(\dots \mathbf{bind}(\mathcal{N}, \langle a_k, b_k \rangle) \dots, \langle a_2, b_2 \rangle), \langle a_1, b_1 \rangle)$$

where we first connect the head of  $a_k$  to the tail of  $b_k$  and lastly connect the head of  $a_1$  to the tail of  $b_1$ . A little proof shows that the order in which we connect arc heads to arc tails does not matter as far as our formal semantics and typing theory is concerned.

Our second derived constructor, called **conn** (for “connect”), uses the preceding generalization of **bind** together with the constructor  $\parallel$ . Let  $\mathcal{N}_1$  and  $\mathcal{N}_2$  be network specifications, and  $\theta \subseteq_{1-1} \mathbf{out}(\mathcal{N}_1) \times \mathbf{in}(\mathcal{N}_2)$ . We expand the expression **conn**  $(\mathcal{N}_1, \mathcal{N}_2, \theta)$  as follows:

$$\mathbf{conn}(\mathcal{N}_1, \mathcal{N}_2, \theta) \implies \mathbf{bind}((\mathcal{N}_1 \parallel \mathcal{N}_2), \theta)$$

In words, **conn** connects some of the output arcs in  $\mathcal{N}_1$  with as many input arcs in  $\mathcal{N}_2$ .

Our third derived constructor is a special case of the preceding **conn**. Let  $\mathcal{N}_1$  be a network where the number  $m \geq 1$  of output arcs is the number of input arcs in another network  $\mathcal{N}_2$ , say:

$$\mathbf{out}(\mathcal{N}_1) = \{a_1, \dots, a_m\} \quad \text{and} \quad \mathbf{in}(\mathcal{N}_2) = \{b_1, \dots, b_m\}$$

Unless otherwise stated, we assume that in every network there is a fixed ordering of the input arcs and another fixed ordering of the output arcs – these orderings, together with the arc names that uniquely label the positions in them, are called the *input* and *output dimensions* of the network. Suppose the entries in  $\mathbf{out}(\mathcal{N}_1)$  and  $\mathbf{in}(\mathcal{N}_2)$  are listed, from left to right, in the assumed ordering of their output and input dimensions, respectively. Let

$$\theta = \{\langle a_1, b_1 \rangle, \dots, \langle a_m, b_m \rangle\} = \mathbf{out}(\mathcal{N}_1) \times \mathbf{in}(\mathcal{N}_2)$$

*i.e.*, the first output  $a_1$  of  $\mathcal{N}_1$  is connected to the first input  $b_1$  of  $\mathcal{N}_2$ , the second output  $a_2$  of  $\mathcal{N}_1$  to the second input  $b_2$  of  $\mathcal{N}_2$ , etc. Our derived constructor  $(\mathcal{N}_1 \oplus \mathcal{N}_2)$  is expanded as:

$$(\mathcal{N}_1 \oplus \mathcal{N}_2) \Longrightarrow \mathbf{conn}(\mathcal{N}_1, \mathcal{N}_2, \theta)$$

which implies that  $\mathbf{in}(\mathcal{N}_1 \oplus \mathcal{N}_2) = \mathbf{in}(\mathcal{N}_1)$  and  $\mathbf{out}(\mathcal{N}_1 \oplus \mathcal{N}_2) = \mathbf{out}(\mathcal{N}_2)$ . As expected,  $\oplus$  is associative as far as our formal semantics and typing theory are concerned, *i.e.*, the semantics and typings for  $\mathcal{N}_1 \oplus (\mathcal{N}_2 \oplus \mathcal{N}_3)$  and  $(\mathcal{N}_1 \oplus \mathcal{N}_2) \oplus \mathcal{N}_3$  are the same.

A fourth derived constructor generalizes **let** and is expanded into nested let-bindings:

$$\begin{aligned} & (\mathbf{let} X \in \{\mathcal{M}_1, \dots, \mathcal{M}_n\} \mathbf{in} \mathcal{N}) \Longrightarrow \\ & (\mathbf{let} X_1 = \mathcal{M}_1 \mathbf{in} (\dots (\mathbf{let} X_n = \mathcal{M}_n \mathbf{in} (\mathcal{N}_1 \parallel \dots \parallel \mathcal{N}_n)) \dots)) \end{aligned}$$

where  $X_1, \dots, X_n$  are fresh hole names and  $\mathcal{N}_i$  is  $\mathcal{N}$  with  $X_i$  substituted for  $X$ , for every  $1 \leq i \leq n$ . Informally, this constructor says that *every one* of the networks  $\{\mathcal{M}_1, \dots, \mathcal{M}_n\}$  can be “safely” placed in the occurrences of  $X$  in  $\mathcal{N}$ .

*Remark 1.* While the preceding derived constructors are expanded using our primitive constructors, not every useful constructor can be so expanded. For example, the constructor

$$\mathbf{try} X \in \{\mathcal{M}_1, \dots, \mathcal{M}_n\} \mathbf{in} \mathcal{N}$$

which we take to mean that *at least one*  $\mathcal{M}_i$  can be “safely” placed in all the occurrences of  $X$  in  $\mathcal{N}$ , cannot be expanded using our primitives so far and the way we later define their semantics. Another constructor also requiring a more developed examination is

$$\mathbf{mix} X \in \{\mathcal{M}_1, \dots, \mathcal{M}_n\} \mathbf{in} \mathcal{N}$$

which we take to mean that a mixture of *several*  $\mathcal{M}_i$  can be selected at the same time and “safely” placed in the occurrences of  $X$  in  $\mathcal{N}$ , generally placing different  $\mathcal{M}_i$  in different occurrences. The constructors **try** and **mix** are examined in a follow-up report. An informal understanding of how they differ from the constructor **let** can be gleaned from Example 1.

Another useful constructor introduces recursively defined components with (unbounded) repeated patterns. In its simplest form, it can be written as:

$$\mathbf{letrec} X = \mathcal{M}[X] \mathbf{in} \mathcal{N}[X]$$

where we write  $\mathcal{M}[X]$  to indicate that  $X$  occurs free in  $\mathcal{M}$ , and similarly in  $\mathcal{N}$ . Informally, this construction corresponds to placing an open-ended network of the form  $\mathcal{M}[\mathcal{M}[\mathcal{M}[\dots]]]$  in the occurrences of  $X$  in  $\mathcal{N}$ . A well-formedness condition here is that the input and output dimensions of  $\mathcal{M}$  must match those of  $X$ . We leave for future examination the semantics and typing of **letrec**, which are still more involved than those of **try** and **mix**.

### 3.2 Well-Formed Network Specifications

In the full report [Kfo11], we spell out 3 conditions, not enforced by the BNF definition at the beginning of Section 3, which guarantee what we call the *well-formedness* of network specifications. We call them:

- the *matching-dimensions* condition,
- the *unique arc-naming* condition,
- the *one binding-occurrence* condition.

We only briefly explain what the second condition specifies: To avoid ambiguities in the formal semantics of Section 4, we need to enforce in the specification of a network  $\mathcal{N}$  that no arc name refers to two different arcs. This in turn requires that we distinguish the arcs of the different copies of the same hole  $X$ . Thus, if we use  $k \geq 2$  copies of  $X$ , we rename their arcs so that each copy has its own set of arcs. We write  ${}^1X, \dots, {}^kX$  to refer to these  $k$  copies of  $X$ . Further details on the *unique arc-naming* condition are in [Kfo11].

*Example 1.* We illustrate several notions. We use one hole  $X$ , and four small networks:  $\mathbf{F}$  (“fork”),  $\mathbf{M}$  (“merge”),  $\mathcal{A}$ , and  $\mathcal{B}$ . We do not assign lower-bound and upper-bound capacities to the arcs of  $\mathbf{F}$ ,  $\mathbf{M}$ ,  $\mathcal{A}$ , and  $\mathcal{B}$  because they play no role before our typing theory is introduced. Graphic representations of  $\mathbf{F}$ ,  $\mathbf{M}$ , and  $X$  are shown in Figure 1, and of  $\mathcal{A}$  and  $\mathcal{B}$  in Figure 2. A possible network specification  $\mathcal{N}$  with two bound occurrences of  $X$  may read as follows:

$$\mathcal{N} = \mathbf{let} X \in \{\mathcal{A}, \mathcal{B}\} \mathbf{in} \mathbf{conn}(\mathbf{F}, \mathbf{conn}({}^1X, \mathbf{conn}({}^2X, \mathbf{M}, \theta_3), \theta_2), \theta_1)$$

where  $\theta_1 = \{\langle c_2, {}^1e_1 \rangle, \langle c_3, {}^1e_2 \rangle\}$ ,  $\theta_2 = \{\langle {}^1e_3, {}^2e_1 \rangle, \langle {}^1e_4, {}^2e_2 \rangle\}$ , and  $\theta_3 = \{\langle {}^2e_3, d_1 \rangle, \langle {}^2e_4, d_2 \rangle\}$ . We wrote  $\mathcal{N}$  using some derived constructors introduced in Section 3.1. Note that:

- all the output arcs  $\{c_2, c_3\}$  of  $\mathbf{F}$  are connected to all the input arcs  $\{{}^1e_1, {}^1e_2\}$  of  ${}^1X$ ,
- all the output arcs  $\{{}^1e_3, {}^1e_4\}$  of  ${}^1X$  are connected to all the input arcs  $\{{}^2e_1, {}^2e_2\}$  of  ${}^2X$ ,
- all the output arcs  $\{{}^2e_3, {}^2e_4\}$  of  ${}^2X$  are connected to all the input arcs  $\{d_1, d_2\}$  of  $\mathbf{M}$ ,

Hence, according to Section 3.1, we can write more simply:

$$\mathcal{N} = \mathbf{let} X \in \{\mathcal{A}, \mathcal{B}\} \mathbf{in} (\mathbf{F} \oplus {}^1X \oplus {}^2X \oplus \mathbf{M})$$

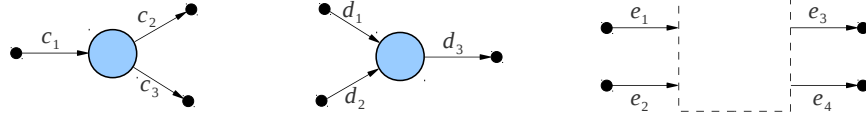
The specification  $\mathcal{N}$  says that  $\mathcal{A}$  or  $\mathcal{B}$  can be selected for insertion wherever hole  $X$  occurs.  $\mathcal{N}$  can be viewed as representing two different network configurations:

$$\mathcal{N}_1 = \mathbf{F} \oplus {}^1\mathcal{A} \oplus {}^2\mathcal{A} \oplus \mathbf{M} \quad \text{and} \quad \mathcal{N}_2 = \mathbf{F} \oplus {}^1\mathcal{B} \oplus {}^2\mathcal{B} \oplus \mathbf{M}$$

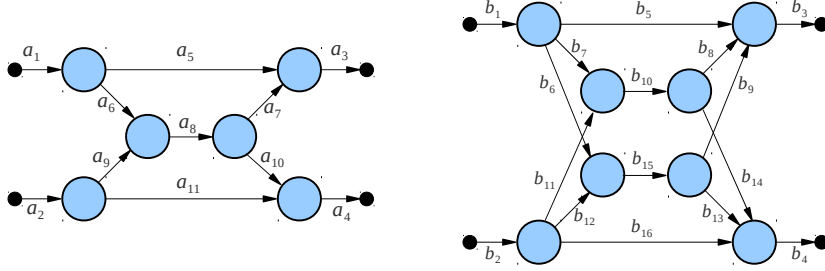
We can say nothing here about properties, such as safety, being satisfied or violated by  $\mathcal{N}_1$  and  $\mathcal{N}_2$ . The semantics of our **let** constructor later will be equivalent to requiring that both configurations be “safe” to use. By contrast, the constructor **try** mentioned in Remark 1 requires only  $\mathcal{N}_1$  or  $\mathcal{N}_2$ , but not necessarily both, to be safe, and the constructor **mix** additionally requires:

$$\mathcal{N}_3 = \mathbf{F} \oplus {}^1\mathcal{A} \oplus {}^2\mathcal{B} \oplus \mathbf{M} \quad \text{and} \quad \mathcal{N}_4 = \mathbf{F} \oplus {}^1\mathcal{B} \oplus {}^2\mathcal{A} \oplus \mathbf{M}$$

to be safe. Safe substitution into holes according to **mix** implies safe substitution according to **let**, which in turn implies safe substitution according to **try**.



**Fig. 1:** Small network  $\mathbf{F}$  (left), small network  $\mathbf{M}$  (middle), and hole  $X$  (right), in Example 1.



**Fig. 2:** Small networks  $\mathcal{A}$  (on the left) and  $\mathcal{B}$  (on the right) in Example 1.

## 4 Formal Semantics of Flow Networks

The preceding section explained what we need to write to specify a network formally. Let  $\mathcal{N}$  be such a network specification. By well-formedness, every small network  $\mathcal{A}$  appearing in  $\mathcal{N}$  has its own separate set of arc names, and every bound occurrence  ${}^i X$  of a hole  $X$  also has its own separate set of arc names, where  $i \geq 1$  is a renaming index. With every small network  $\mathcal{A}$ , we associate two sets of functions, its *full semantics*  $\llbracket \mathcal{A} \rrbracket$  and its *IO-semantics*  $\langle\langle \mathcal{A} \rangle\rangle$ . Let  $\mathbf{A}_{\text{in}} = \text{in}(\mathcal{A})$ ,  $\mathbf{A}_{\text{out}} = \text{out}(\mathcal{A})$ , and  $\mathbf{A}_{\#} = \#(\mathcal{A})$ . The sets  $\llbracket \mathcal{A} \rrbracket$  and  $\langle\langle \mathcal{A} \rangle\rangle$  are defined thus:

$$\begin{aligned} \llbracket \mathcal{A} \rrbracket &= \{ f : \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}} \uplus \mathbf{A}_{\#} \rightarrow \mathbb{R}^+ \mid f \text{ is a feasible flow in } \mathcal{A} \} \\ \langle\langle \mathcal{A} \rangle\rangle &= \{ f : \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}} \rightarrow \mathbb{R}^+ \mid f \text{ can be extended to a feasible flow } f' \text{ in } \mathcal{A} \} \end{aligned}$$

Let  $X$  be a hole, with  $\text{in}(X) = \mathbf{A}_{\text{in}}$  and  $\text{out}(X) = \mathbf{A}_{\text{out}}$ . The *full semantics*  $\llbracket X \rrbracket$  and the *IO-semantics*  $\langle\langle X \rangle\rangle$  are the same set of functions:

$$\llbracket X \rrbracket = \langle\langle X \rangle\rangle \subseteq \{ f : \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}} \rightarrow \mathbb{R}^+ \mid f \text{ is a bounded function} \}$$

This definition of  $\llbracket X \rrbracket = \langle\langle X \rangle\rangle$  is ambiguous: In contrast to the uniquely defined full semantics and IO-semantics of a small network  $\mathcal{A}$ , there are infinitely many  $\llbracket X \rrbracket = \langle\langle X \rangle\rangle$  for the same  $X$ , but exactly one (possibly  $\llbracket X \rrbracket = \langle\langle X \rangle\rangle = \emptyset$ ) will satisfy the requirement in clause 4 below.

Starting from the full semantics of small networks and holes, we define by induction the full semantics  $\llbracket \mathcal{N} \rrbracket$  of a network specification  $\mathcal{N}$  in general. In a similar way, we can define the IO-semantics  $\langle\langle \mathcal{N} \rangle\rangle$  of  $\mathcal{N}$  by induction, starting from the IO-semantics of small networks and holes. For conciseness, we define  $\llbracket \mathcal{N} \rrbracket$  separately first, and then define  $\langle\langle \mathcal{N} \rangle\rangle$  from  $\llbracket \mathcal{N} \rrbracket$ .

We need a few preliminary notions. Let  $\mathcal{M}$  be a network specification. By our convention of listing all input arcs first, all output arcs second, and all internal arcs third, let:

$$\text{in}(\mathcal{M}) = \{a_1, \dots, a_k\}, \quad \text{out}(\mathcal{M}) = \{a_{k+1}, \dots, a_{k+\ell}\}, \quad \text{and} \quad \#(\mathcal{M}) = \{a_{k+\ell+1}, \dots, a_{k+\ell+m}\}.$$

If  $f \in \llbracket \mathcal{M} \rrbracket$  with  $f(a_1) = r_1, \dots, f(a_{k+\ell+m}) = r_{k+\ell+m}$ , we may represent  $f$  by the sequence  $\langle r_1, \dots, r_{k+\ell+m} \rangle$ . We may therefore represent:

- $[f]_{\mathbf{in}(\mathcal{M})}$  by the sequence  $\langle r_1, \dots, r_k \rangle$ ,
- $[f]_{\mathbf{out}(\mathcal{M})}$  by the sequence  $\langle r_{k+1}, \dots, r_{k+\ell} \rangle$ , and
- $[f]_{\#(\mathcal{M})}$  by the sequence  $\langle r_{k+\ell+1}, \dots, r_{k+\ell+m} \rangle$ ,

where  $[f]_{\mathbf{in}(\mathcal{M})}$ ,  $[f]_{\mathbf{out}(\mathcal{M})}$ , and  $[f]_{\#(\mathcal{M})}$ , are the restrictions of  $f$  to the subsets  $\mathbf{in}(\mathcal{M})$ ,  $\mathbf{out}(\mathcal{M})$ , and  $\#(\mathcal{M})$ , of its domain. Let  $\mathcal{N}$  be another network specification and  $g \in \llbracket \mathcal{N} \rrbracket$ . We define  $f \parallel g$  as follows:

$$(f \parallel g) = [f]_{\mathbf{in}(\mathcal{M})} \cdot [g]_{\mathbf{in}(\mathcal{N})} \cdot [f]_{\mathbf{out}(\mathcal{M})} \cdot [g]_{\mathbf{out}(\mathcal{N})} \cdot [f]_{\#(\mathcal{M})} \cdot [g]_{\#(\mathcal{N})}$$

where “ $\cdot$ ” is sequence concatenation. The operation “ $\parallel$ ” on flows is associative, but not commutative, just as the constructor “ $\parallel$ ” on network specifications. We define the full semantics  $\llbracket \mathcal{M} \rrbracket$  for every subexpression  $\mathcal{M}$  of  $\mathcal{N}$ , by induction on the structure of the specification  $\mathcal{N}$ :

1. If  $\mathcal{M} = \mathcal{A}$ , then  $\llbracket \mathcal{M} \rrbracket = \llbracket \mathcal{A} \rrbracket$ .
2. If  $\mathcal{M} = {}^i X$ , then  $\llbracket \mathcal{M} \rrbracket = {}^i \llbracket X \rrbracket$ .
3. If  $\mathcal{M} = (\mathcal{P}_1 \parallel \mathcal{P}_2)$ , then  $\llbracket \mathcal{M} \rrbracket = \{ (f_1 \parallel f_2) \mid f_1 \in \llbracket \mathcal{P}_1 \rrbracket \text{ and } f_2 \in \llbracket \mathcal{P}_2 \rrbracket \}$ .
4. If  $\mathcal{M} = (\mathbf{let } X = \mathcal{P} \mathbf{ in } \mathcal{P}')$ , then  $\llbracket \mathcal{M} \rrbracket = \llbracket \mathcal{P}' \rrbracket$ , provided two conditions:<sup>3</sup>
  - (a)  $\dim(X) \approx \dim(\mathcal{P})$ ,
  - (b)  $\llbracket X \rrbracket \approx \{ [g]_A \mid g \in \llbracket \mathcal{P} \rrbracket \}$  where  $A = \mathbf{in}(\mathcal{P}) \cup \mathbf{out}(\mathcal{P})$ .
5. If  $\mathcal{M} = \mathbf{bind}(\mathcal{P}, \langle a, b \rangle)$ , then  $\llbracket \mathcal{M} \rrbracket = \{ f \mid f \in \llbracket \mathcal{P} \rrbracket \text{ and } f(a) = f(b) \}$ .

All of  $\mathcal{N}$  is a special case of a subexpression of  $\mathcal{N}$ , so that the semantics of  $\mathcal{N}$  is simply  $\llbracket \mathcal{N} \rrbracket$ . Note, in clause 2, all bound occurrences  ${}^i X$  of the same hole  $X$  are assigned the same semantics  $\llbracket X \rrbracket$ , up to renaming of arc names. We now define the IO-semantics of  $\mathcal{N}$  as follows:

$$\langle \langle \mathcal{N} \rangle \rangle = \{ [f]_A \mid f \in \llbracket \mathcal{N} \rrbracket \}$$

where  $A = \mathbf{in}(\mathcal{N}) \cup \mathbf{out}(\mathcal{N})$  and  $[f]_A$  is the restriction of  $f$  to  $A$ .

*Remark 2.* We can define rewrite rules on network specifications in order to reduce each into an equivalent finite set of network specifications in *normal form*, a normal form being free of let-bindings. We can do this so that the formal semantics of network specifications are an *invariant* of this rewriting. This establishes the *soundness* of the *operational semantics* (represented by the rewrite rules) of our DSL relative to the formal semantics defined above.

**Flow Conservation, Capacity Constraints, Type Satisfaction (Continued).** The fundamental concepts stated in relation to small networks  $\mathcal{A}$  in Definitions 1, 2, and 3, are extended to arbitrary network specifications  $\mathcal{N}$ . These are stated as “properties” (not “definitions”) because they apply to  $\llbracket \mathcal{N} \rrbracket$  (not to  $\mathcal{N}$ ), and  $\llbracket \mathcal{N} \rrbracket$  is built up inductively from  $\{ \llbracket \mathcal{A} \rrbracket \mid \mathcal{A} \text{ occurs in } \mathcal{N} \}$ .

<sup>3</sup> “ $\dim(X) \approx \dim(\mathcal{P})$ ” means the number of input arcs and their ordering (or input dimension) and the number of output arcs and their ordering (or output dimension) of  $X$  match those of  $\mathcal{P}$ , up to arc renaming. “ $\llbracket X \rrbracket \approx \{ [g]_A \mid g \in \llbracket \mathcal{P} \rrbracket \}$ ” means for every  $f : \mathbf{in}(X) \cup \mathbf{out}(X) \rightarrow \mathbb{R}^+$ , it holds that  $f \in \llbracket X \rrbracket$  iff there is  $g \in \llbracket \mathcal{P} \rrbracket$  such that  $f \approx [g]_A$ .

*Property 1.* The nodes of  $\mathcal{N}$  are all the nodes in the small networks occurring in  $\mathcal{N}$ , because our DSL in Section 3 does not introduce new nodes beyond those in the small networks. Hence,  $\llbracket \mathcal{N} \rrbracket$  satisfies *flow conservation* because, for every small network  $\mathcal{A}$  in  $\mathcal{N}$ , every  $f \in \llbracket \mathcal{A} \rrbracket$  satisfies flow conservation at every node, *i.e.*, the equation in (1) in Definition 1.

*Property 2.* The arcs introduced by our DSL, beyond the arcs in the small networks, are the input/output arcs of the holes. Lower-bound and upper-bound capacities on the latter arcs are set in order not to conflict with those already defined on the input/output arcs of small networks. Hence,  $\llbracket \mathcal{N} \rrbracket$  satisfies *the capacity constraints* because, for every small network  $\mathcal{A}$  in  $\mathcal{N}$ , every  $f \in \llbracket \mathcal{A} \rrbracket$  satisfies the capacity constraints on every arc, *i.e.*, the inequalities in (2) in Definition 2.

However, stressing the obvious, even if  $\llbracket \mathcal{A} \rrbracket \neq \emptyset$  for every small network  $\mathcal{A}$  in  $\mathcal{N}$ , it may still be that  $\mathcal{N}$  is unsafe to use, *i.e.*, it may be that there is no feasible flow in  $\mathcal{N}$  and  $\llbracket \mathcal{N} \rrbracket = \emptyset$ . We use the type system (Section 7) to reject unsafe network specifications  $\mathcal{N}$ .

**Definition 5.** Let  $\mathcal{N}$  be a network, with  $\mathbf{A}_{in} = \mathbf{in}(\mathcal{N})$ ,  $\mathbf{A}_{out} = \mathbf{out}(\mathcal{N})$ , and  $\mathbf{A}_{\#} = \#\mathcal{N}$ . A typing  $T$  for  $\mathcal{N}$ , also denoted  $(\mathcal{N} : T)$ , is a function  $T : \mathcal{P}(\mathbf{A}_{in} \cup \mathbf{A}_{out}) \rightarrow \mathbb{R} \times \mathbb{R}$ , which may, or may not, be satisfied by  $f \in \llbracket \mathcal{N} \rrbracket$  or by  $f \in \llbracket \mathcal{A} \rrbracket$ . We say  $f \in \llbracket \mathcal{N} \rrbracket$  or  $f \in \llbracket \mathcal{A} \rrbracket$  satisfies  $T$  iff, for every  $A \subseteq \mathbf{A}_{in} \cup \mathbf{A}_{out}$  with  $T(A) = [r, r']$ , it is the case that:

$$(4) \quad r \leq \sum f(A \cap \mathbf{A}_{in}) - \sum f(A \cap \mathbf{A}_{out}) \leq r'$$

## 5 Typings Are Polytopes

Let  $\mathcal{N}$  be a network specification, and let  $\mathbf{A}_{in} = \mathbf{in}(\mathcal{N})$  and  $\mathbf{A}_{out} = \mathbf{out}(\mathcal{N})$ . Let  $T$  be a typing for  $\mathcal{N}$  that assigns an interval  $[r, r']$  to  $A \subseteq \mathbf{A}_{in} \cup \mathbf{A}_{out}$ . Let  $|\mathbf{A}_{in}| + |\mathbf{A}_{out}| = m$ , for some  $m \geq 0$ . As usual, there is a fixed ordering on the arcs in  $\mathbf{A}_{in}$  and again on the arcs in  $\mathbf{A}_{out}$ . With no loss of generality, suppose:

$$A_1 = A \cap \mathbf{A}_{in} = \{a_1, \dots, a_k\} \quad \text{and} \quad A_2 = A \cap \mathbf{A}_{out} = \{a_{k+1}, \dots, a_\ell\},$$

where  $\ell \leq m$ . Instead of writing  $T(A) = [r, r']$ , we may write:

$$T(A) : \quad a_1 + \dots + a_k - a_{k+1} - \dots - a_\ell : [r, r']$$

where the inserted polarities, + or -, indicate whether the arcs are input or output, respectively. A flow through the arcs  $\{a_1, \dots, a_k\}$  contributes a *positive* quantity, and through the arcs  $\{a_{k+1}, \dots, a_\ell\}$  a *negative* quantity, and these two quantities together should add up to a value within the interval  $[r, r']$ .

A typing  $T$  for  $\mathbf{A}_{in} \cup \mathbf{A}_{out}$  induces a *polytope* (or *bounded polyhedron*), which we call  $\text{Poly}(T)$ , in the Euclidean hyperspace  $\mathbb{R}^m$ . We think of the  $m$  arcs in  $\mathbf{A}_{in} \cup \mathbf{A}_{out}$  as the  $m$  dimensions of the space  $\mathbb{R}^m$ .  $\text{Poly}(T)$  is the non-empty intersection of at most  $2 \cdot (2^m - 1)$  halfspaces, because there are  $(2^m - 1)$  non-empty subsets in  $\mathcal{P}(\mathbf{A}_{in} \cup \mathbf{A}_{out})$ . The interval  $[r, r']$ , which  $T$  assigns to such a subset  $A = \{a_1, \dots, a_\ell\}$  as above, induces two linear inequalities in the variables  $\{a_1, \dots, a_\ell\}$ , denoted  $T_{\geq}(A)$  and  $T_{\leq}(A)$ :

$$(5) \quad T_{\geq}(A) : \quad a_1 + \dots + a_k - a_{k+1} - \dots - a_\ell \geq r \quad \text{and} \quad T_{\leq}(A) : \quad a_1 + \dots + a_k - a_{k+1} - \dots - a_\ell \leq r'$$

and, therefore, two halfspaces  $\text{Half}(T_{\geq}(A))$  and  $\text{Half}(T_{\leq}(A))$ :

$$(6) \quad \text{Half}(T_{\geq}(A)) = \{ \mathbf{r} \in \mathbb{R}^m \mid \mathbf{r} \models T_{\geq}(A) \} \quad \text{and} \quad \text{Half}(T_{\leq}(A)) = \{ \mathbf{r} \in \mathbb{R}^m \mid \mathbf{r} \models T_{\leq}(A) \}$$

where “ $\mathbf{r} \models T_{\geq}(A)$ ” means “ $\mathbf{r}$  satisfies  $T_{\geq}(A)$ ” and similarly for “ $\mathbf{r} \models T_{\leq}(A)$ ”. We can therefore define  $\text{Poly}(T)$  formally as follows:

$$\text{Poly}(T) = \bigcap \{ \text{Half}(T_{\geq}(A)) \cap \text{Half}(T_{\leq}(A)) \mid \emptyset \neq A \subseteq \mathbf{A}_{in} \cup \mathbf{A}_{out} \}$$

## 5.1 Uniqueness and Redundancy in Typings

We can view a network typing  $T$  as a syntactic expression, with its semantics  $\text{Poly}(T)$  being a polytope in Euclidean hyperspace. As in other situations connecting syntax and semantics, there are generally distinct typings  $T$  and  $T'$  such that  $\text{Poly}(T) = \text{Poly}(T')$ . This is an obvious consequence of the fact that the same polytope can be defined by many different equivalent sets of linear inequalities. To achieve uniqueness of typings, as well as some efficiency of manipulating them, we may try an approach that eliminates redundant inequalities in the collection:

$$(7) \{ T_{\geq}(A) \mid \emptyset \neq A \in \mathcal{P}(\mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}) \} \cup \{ T_{\leq}(A) \mid \emptyset \neq A \in \mathcal{P}(\mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}) \}$$

where  $T_{\geq}(A)$  and  $T_{\leq}(A)$  are as in (5) above. There are standard procedures which determine if a finite set of inequalities are linearly independent and, if they are not, select an equivalent subset of linearly independent inequalities. These issues are taken up in the full report [Kfo11].

If  $\mathcal{N}_1 : T_1$  and  $\mathcal{N}_2 : T_2$  are typings for networks  $\mathcal{N}_1$  and  $\mathcal{N}_2$  with matching input and output dimensions, we write  $T_1 \equiv T_2$  whenever  $\text{Poly}(T_1) \approx \text{Poly}(T_2)$ , in which case we say that  $T_1$  and  $T_2$  are *equivalent*. If  $\mathcal{N}_1 = \mathcal{N}_2$ , then  $T_1 \equiv T_2$  whenever  $\text{Poly}(T_1) = \text{Poly}(T_2)$ .

**Definition 6.** Let  $T : \mathcal{P}(\mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}) \rightarrow \mathbb{R} \times \mathbb{R}$  be a typing for network specification  $\mathcal{N}$ , where  $\mathbf{A}_{\text{in}} = \mathbf{in}(\mathcal{N})$  and  $\mathbf{A}_{\text{out}} = \mathbf{out}(\mathcal{N})$ .  $T$  is a *tight typing* if for every typing  $T'$  such that  $T \equiv T'$  and every  $A \subseteq \mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}$ , the interval  $T(A)$  is contained in  $T'(A)$ , i.e.,  $T(A) \subseteq T'(A)$ .

**Proposition 1 (Every Typing Is Equivalent to a Tight Typing).** *There is an algorithm  $\text{Tight}()$  which, given a typing  $(\mathcal{N} : T)$  as input, always terminates and returns an equivalent tight typing  $(\mathcal{N} : \text{Tight}(T))$ .*

## 5.2 Valid Typings and Principal Typings

Let  $\mathcal{N}$  be a network,  $\mathbf{A}_{\text{in}} = \mathbf{in}(\mathcal{N})$  and  $\mathbf{A}_{\text{out}} = \mathbf{out}(\mathcal{N})$ . A typing  $\mathcal{N} : T$  is *valid* iff it is sound:

**(soundness)** Every  $f_0 : \mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}} \rightarrow \mathbb{R}^+$  satisfying  $T$  can be extended to a feasible  $f \in \llbracket \mathcal{N} \rrbracket$ .

We say the typing  $\mathcal{N} : T$  for  $\mathcal{N}$  is a *principal typing* if it is both sound and complete:

**(completeness)** Every feasible flow  $f \in \llbracket \mathcal{N} \rrbracket$  satisfies  $T$ .

More succinctly, using the IO-semantics  $\langle\langle \mathcal{N} \rangle\rangle$  instead of the full semantics  $\llbracket \mathcal{N} \rrbracket$ , the typing  $\mathcal{N} : T$  is *valid* iff  $\text{Poly}(T) \subseteq \langle\langle \mathcal{N} \rangle\rangle$ , and it is *principal* iff  $\text{Poly}(T) = \langle\langle \mathcal{N} \rangle\rangle$ . A useful notion in type theories is *subtyping*. If  $T_1$  is a *subtype* of  $T_2$ , in symbols  $T_1 <: T_2$ , this means that any object of type  $T_1$  can be safely used in a context where an object of type  $T_2$  is expected:

**(subtyping)**  $T_1 <: T_2$  iff  $\text{Poly}(T_2) \subseteq \text{Poly}(T_1)$ .

**Proposition 2 (Principal Typings Are Subtypes of Valid Typings).** *If  $(\mathcal{N} : T_1)$  is a principal typing, and  $(\mathcal{N} : T_2)$  a valid typing for the same  $\mathcal{N}$ , then  $T_1 <: T_2$ .*

**Proposition 3 (Principal Typings Are Equivalent).** *If  $(\mathcal{N} : T_1)$  and  $(\mathcal{N} : T_2)$  are two principal typings for the same network  $\mathcal{N}$ , then  $T_1 \equiv T_2$ . If  $T_1$  and  $T_2$  are tight, then  $T_1 = T_2$ .*

## 6 Inferring Typings for Small Networks

**Theorem 1 (Inferring Principal Typings for Small Networks).** *Let  $\mathcal{A}$  be a small network. We can effectively compute a principal and uniformly tight typing  $T$  for  $\mathcal{A}$ .*

*Example 2.* Consider again the two small networks  $\mathcal{A}$  and  $\mathcal{B}$  from Example 1. We assign capacities to their arcs and compute their respective principal typings. The sets of arcs in  $\mathcal{A}$  and  $\mathcal{B}$  are, respectively:  $\mathbf{A} = \{a_1, \dots, a_{11}\}$  and  $\mathbf{B} = \{b_1, \dots, b_{16}\}$ . All the lower-bounds and most of the upper-bounds are trivial, *i.e.*, they do not restrict flow. Specifically, the lower-bound capacity on every arc is 0, and the upper-bound capacity on every arc is a “very large number”, unless indicated otherwise in Figure 3 by the numbers in rectangular boxes, namely:

$$\begin{array}{llll} U(a_5) = 5, & U(a_8) = 10, & U(a_{11}) = 15, & \text{nontrivial upper-bounds in } \mathcal{A}, \\ U(b_5) = 3, & U(b_6) = 2, & U(b_9) = 2, & U(b_{10}) = 10, \text{ nontrivial upper-bounds in } \mathcal{B}, \\ U(b_{11}) = 8, & U(b_{13}) = 8, & U(b_{15}) = 10, & U(b_{16}) = 7, \text{ nontrivial upper-bounds in } \mathcal{B}. \end{array}$$

We compute the principal typings  $T_{\mathcal{A}}$  and  $T_{\mathcal{B}}$ , by assigning a bounded interval to every subset in  $\mathcal{P}(\{a_1, a_2, a_3, a_4\})$  and  $\mathcal{P}(\{b_1, b_2, b_3, b_4\})$ . This is a total of 16 intervals for each, but we can ignore the empty set to which we assign the empty interval  $\emptyset$ , as well as interval assignments that are implied by those listed below. We use the construction in the proof (omitted in this paper, included in the full report [Kfo11]) of Theorem 1 to compute  $T_{\mathcal{A}}$  and  $T_{\mathcal{B}}$ .

$T_{\mathcal{A}}$  assignments :

$$\begin{array}{llll} \boxed{a_1 : [0, 15]} & \boxed{a_2 : [0, 25]} & \boxed{-a_3 : [-15, 0]} & \boxed{-a_4 : [-25, 0]} \\ \boxed{a_1 + a_2 : [0, 30]} & \boxed{a_1 - a_3 : [-10, 10]} & \boxed{a_1 - a_4 : [-25, 15]} & \\ \boxed{a_2 - a_3 : [-15, 25]} & \boxed{a_2 - a_4 : [-10, 10]} & \boxed{-a_3 - a_4 : [-30, 0]} & \end{array}$$

$T_{\mathcal{B}}$  assignments :

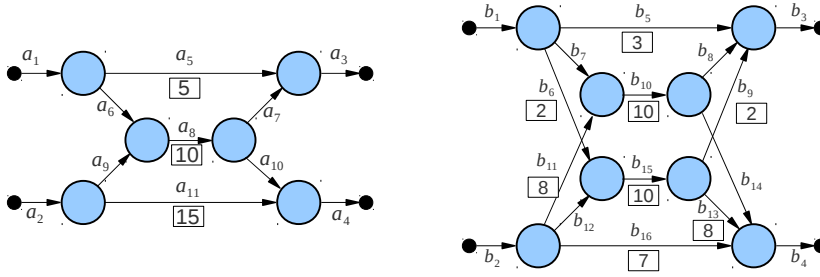
$$\begin{array}{llll} \boxed{b_1 : [0, 15]} & \boxed{b_2 : [0, 25]} & \boxed{-b_3 : [-15, 0]} & \boxed{-b_4 : [-25, 0]} \\ \boxed{b_1 + b_2 : [0, 30]} & \boxed{b_1 - b_3 : [-10, 12]} & \boxed{b_1 - b_4 : [-25, 15]} & \\ \boxed{b_2 - b_3 : [-15, 25]} & \boxed{b_2 - b_4 : [-12, 10]} & \boxed{-b_3 - b_4 : [-30, 0]} & \end{array}$$

The types in rectangular boxes are those of  $[T_{\mathcal{A}}]_{\text{in}}$  and  $[T_{\mathcal{B}}]_{\text{in}}$  which are equivalent, and those of  $[T_{\mathcal{A}}]_{\text{out}}$  and  $[T_{\mathcal{B}}]_{\text{out}}$  which are equivalent. Thus,  $[T_{\mathcal{A}}]_{\text{in}} \equiv [T_{\mathcal{B}}]_{\text{in}}$  and  $[T_{\mathcal{A}}]_{\text{out}} \equiv [T_{\mathcal{B}}]_{\text{out}}$ . Nevertheless,  $T_{\mathcal{A}} \neq T_{\mathcal{B}}$ , the difference being in the (underlined) types assigned to some subsets mixing input and output arcs.

In this example,  $T_{\mathcal{B}} < T_{\mathcal{A}}$  because  $\text{Poly}(T_{\mathcal{A}}) \subseteq \text{Poly}(T_{\mathcal{B}})$ . The converse does not hold. As a result, there are feasible flows in  $\mathcal{B}$  which are not feasible in  $\mathcal{A}$  (some computed in [Kfo11]).

## 7 A Typing System

We set up a formal system for assigning typings to network specifications. The process of inferring typings, based on this system, is deferred to Section 8. The system is in Figure 4, where we follow standard conventions in formulating the rules. We call  $\Gamma$  a *typing environment*, which is a finite set of *typing assumptions* for holes, each of the form  $(X : T)$ . If  $(X : T)$  is a typing assumption, with  $\text{in}(X) = \mathbf{A}_{\text{in}}$  and  $\text{out}(X) = \mathbf{A}_{\text{out}}$ , then  $T : \mathcal{P}(\mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}) \rightarrow \mathbb{R} \times \mathbb{R}$ .



**Fig. 3:** An assignment of arc capacities for small networks  $\mathcal{A}$  (left) and  $\mathcal{B}$  (right) in Example 2.

HOLE	$\frac{(X : T) \in \Gamma}{\Gamma \vdash {}^i X : {}^i T}$	$i \geq 1$ is the smallest available renaming index
SMALL	$\frac{}{\Gamma \vdash \mathcal{A} : T}$	$T$ is a typing for small network $\mathcal{A}$
PAR	$\frac{\Gamma \vdash \mathcal{N}_1 : T_1 \quad \Gamma \vdash \mathcal{N}_2 : T_2}{\Gamma \vdash (\mathcal{N}_1 \parallel \mathcal{N}_2) : (T_1 \parallel T_2)}$	
BIND	$\frac{\Gamma \vdash \mathcal{N} : T}{\Gamma \vdash \mathbf{bind}(\mathcal{N}, \langle a, b \rangle) : \mathbf{bind}(T, \langle a, b \rangle)}$	$\langle a, b \rangle \in \mathbf{out}(\mathcal{N}) \times \mathbf{in}(\mathcal{N})$
LET	$\frac{\Gamma \vdash \mathcal{M} : T_1 \quad \Gamma \cup \{(X : T_2)\} \vdash \mathcal{N} : T}{\Gamma \vdash (\mathbf{let} X = \mathcal{M} \mathbf{in} \mathcal{N}) : T}$	$T_1 \approx T_2$

**Fig. 4:** Typing Rules. The operations on typings,  $(T_1 \parallel T_2)$  and  $\mathbf{bind}(T, \langle a, b \rangle)$ , are defined in [Kfo11].

If a typing  $T$  is derived for a network specification  $\mathcal{N}$  according to the rules in Figure 4, it will be the result of deriving an *assertion* (or *judgment*) of the form “ $\Gamma \vdash \mathcal{N} : T$ ”. If  $\mathcal{N}$  is closed, then this final typing judgment will be of the form “ $\vdash \mathcal{N} : T$ ” where all typing assumptions have been discharged.

**Theorem 2 (Existence of Principal Typings for Networks in General).** *Let  $\mathcal{N}$  be a closed network specification and  $T$  a typing for  $\mathcal{N}$  derived according to the rules in Figure 4, i.e., the judgment “ $\vdash \mathcal{N} : T$ ” is derivable according to the rules. If the typing of every small network  $\mathcal{A}$  in  $\mathcal{N}$  is principal (resp., valid) for  $\mathcal{A}$ , then  $T$  is a principal (resp., valid) typing for  $\mathcal{N}$ .*

## 8 Inferring Typings for Flow Networks in General

The main difficulty in typing inference is in relation to **let**-bindings. Consider a specification  $\mathcal{N}$  of the form  $(\mathbf{let} X = \mathcal{M} \mathbf{in} \mathcal{P})$ . Let  $\mathbf{A}_{\text{in}} = \mathbf{in}(X)$  and  $\mathbf{A}_{\text{out}} = \mathbf{out}(X)$ . Suppose  $X$  occurs  $n \geq 1$  times in  $\mathcal{P}$ , so that its input/output arcs are renamed in each of the  $n$  occurrences according to:  ${}^1(\mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}), \dots, {}^n(\mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}})$ . A typing for  $X$  and for its occurrences  ${}^i X$  in  $\mathcal{P}$  can be given *concretely* or *symbolically*. If concretely, then these typings are functions of the form:

$$T_X : \mathcal{P}(\mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}) \rightarrow \mathbb{R} \times \mathbb{R} \quad \text{and} \quad {}^i T_X : \mathcal{P}({}^i \mathbf{A}_{\text{in}} \cup {}^i \mathbf{A}_{\text{out}}) \rightarrow \mathbb{R} \times \mathbb{R}$$

for every  $1 \leq i \leq n$ . According to the typing rules in Figure 4, a valid typing for  $\mathcal{N}$  requires that:  $T_X \approx {}^1 T_X \approx \dots \approx {}^n T_X$ . If symbolically, then for every  $B \subseteq \mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}$ , the interval  $T_X(B)$

is written as  $[x_B, y_B]$  where the two ends  $x_B$  and  $y_B$  are yet to be determined, and similarly for  ${}^i T_X(B)$  and every  $B \subseteq {}^i \mathbf{A}_{\text{in}} \cup {}^i \mathbf{A}_{\text{out}}$ . For later reference, call  $x_B$  a *lower-end parameter* and  $y_B$  an *upper-end parameter*. We can infer a typing for  $\mathcal{N}$  in one of two ways, which produce the same end result but whose organizations are very different:

**(sequential)** First infer a principal typing  $T_{\mathcal{M}}$  for  $\mathcal{M}$ , then use  $k$  copies  ${}^1 T_{\mathcal{M}}, \dots, {}^n T_{\mathcal{M}}$  to infer a principal typing  $T_{\mathcal{P}}$  for  $\mathcal{P}$ , which is also a principal typing  $T_{\mathcal{N}}$  for  $\mathcal{N}$ .

**(parallel)** Infer principal typings  $T_{\mathcal{M}}$  for  $\mathcal{M}$  and  $T_{\mathcal{P}}$  for  $\mathcal{P}$ , separately.  $T_{\mathcal{P}}$  is parametrized by the typings  ${}^i T_X$  written symbolically. A typing for  $\mathcal{N}$  is obtained by setting lower-end and upper-end parameters in  ${}^i T_X$  to corresponding lower-end and upper-end values in  $T_{\mathcal{M}}$ .

Both approaches are *modular*, in that both are syntax-directed according to the inductive definition of  $\mathcal{N}$ . However, the parallel approach has the advantage of being independent of the order in which the inference proceeds (*i.e.*, it does not matter whether  $T_{\mathcal{M}}$  is inferred before or after, or simultaneously with,  $T_{\mathcal{P}}$ ). We therefore qualify the parallel approach as being additionally *fully compositional*, in contrast to the sequential approach which is not. Moreover, the latter requires that the whole specification  $\mathcal{N}$  be known before typing inference can start, justifying the additional qualification of being a *whole-specification* analysis. The sequential approach is simpler to define and is presented in full in [Kfo11]. We delay the examination of the parallel/fully-compositional approach to a follow-up report.

## 9 Related and Future Work

Ours is not the only study that uses *intervals* as types and *polytopes* as typings. There were earlier attempts that heavily drew on linear algebra and polytope theory, mostly initiated by researchers who devised “types as abstract interpretations” – see [Cou97] and references therein. However, the motivations for these earlier attempts were entirely different and applied to programming languages unrelated to our DSL. For example, polytopes were used to define “invariant safety properties”, or “types” by another name, for ESTEREL – an imperative synchronous language for the development of reactive systems [Hal93].

Apart from the difference in motivation with these earlier works, there are also technical differences in the use of polytopes. Whereas the earlier works consider polytopes defined by unrestricted linear constraints [CH78, Hal93], our polytopes are defined by linear constraints where every coefficient is  $+1$  or  $-1$ , as implied by our Definitions 1, 2, 3, and 4. Ours are in fact identical to linear constraints (but not necessarily the linear objective function) that arise in the *network simplex method* [Cun79], *i.e.*, linear programming applied to problems of network flows. There is still on-going research to improve network-simplex algorithms (*e.g.*, [RT09]), which will undoubtedly have a bearing on the efficiency of typing inference for our DSL.

As alluded in the Introduction, we omitted an operational semantics of our DSL in this paper to stay clear of complexity issues arising from the associated rewrite (or reduction) rules. Among other benefits, relying on a denotational semantics allowed us to harness this complexity by performing a static analysis, via our typing theory, without carrying out a naive hole-expansion (or **let-in** elimination). We thus traded the intuitively simpler but costlier operational semantics for the more compact denotational semantics.

However, as we introduce other constructs involving holes in follow-up reports (**try-in**, **mix-in**, and **letrec-in** mentioned in Remark 1 of Section 3) this trade-off will diminish in importance. An operational semantics of our DSL involving these more complex hole-binders will bring it closer in line with various calculi involving *patterns* (instead of *holes*) and where rewriting consists in eliminating *pattern-binders*. See [JK06, BCKL03, CLW03, CKL04, BBCK07] and references therein. It remains to be seen how much of the theory developed for these pattern calculi can be adapted to an operational semantics of our DSL.

## References

- [AMM01] H. Aydin, R. Melhem, and D. Moss. Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics. In *Proc. of EuroMicro Conference on Real-Time Systems*, pages 225–232, 2001.
- [BBCK07] P. Baldan, C. Bertolissi, H. Cirstea, and C. Kirchner. A Rewriting Calculus for Cyclic Higher-Order Term Graphs. *Math. Structures in Computer Science*, 17:363–406, 2007.
- [BCKL03] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Proc. 30th ACM Symp. on POPL*, pages 250–261, 2003.
- [BKLO09] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: Tool and Use Cases. In *Proc. of IEEE/RTSS Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Washington D.C., December 2009.
- [BKLO10] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: The Formal Framework. In *Proc. of 13th ACM HSCC*, Stockholm, April 2010.
- [BL06] S. Balon and G. Leduc. Dividing the Traffic Matrix to Approach Optimal Traffic Engineering. In *14th IEEE Int’l Conf. on Networks*, volume 2, pages 566–571, September 2006.
- [CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. 5th ACM Symp. on POPL*, pages 84–96, Tucson, January 1978.
- [CKL04] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. *Electronic Notes in Theoretical Computer Science*, 71:3–19, 2004.
- [CLW03] H. Cirstea, L. Liquori, and B. Wack. Rewriting Calculus with Fixpoints: Untyped and First-order Systems. In *Post-proceedings of TYPES, LNCS*, pages 147–161. Springer, 2003.
- [Cou97] P. Cousot. Types as Abstract Interpretations, invited paper. In *Proc. of 24th ACM Symp. on Principles of Programming Languages*, pages 316–331, Paris, January 1997.
- [Cun79] W. H. Cunningham. Theoretical Properties of the Network Simplex Method. *Mathematics of Operations Research*, 4(2):196–208, May 1979.
- [DL97] Z. Deng and J. W.-S. Liu. Scheduling Real-Time Applications in an Open Environment. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 308–319, 1997.
- [Hal93] N. Halbwachs. Delay Analysis in Synchronous Programs. In *Fifth Conference on Computer-Aided Verification*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.
- [JK06] C. Barry Jay and D. Kesner. Pure Pattern Calculus. In *European Symposium on Programming*, pages 100–114, 2006.
- [KBS04] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
- [Kfo11] A. Kfoury. A Domain-Specific Language for Incremental and Modular Design of Large-Scale Safety-Critical Flow Networks (Part 1). Technical Report BUCS-TR-2011-011, Boston Univ, May 2011.
- [LBJ<sup>+</sup>95] S.S. Lim, Y.H. Bae, G.T. Jang, B.D. Rhee, S.L. Min, C.Y. Park, H.S., K. Park, S.M. Moon, and C.S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. In *IEEE REAL-TIME SYSTEMS SYMPOSIUM*, pages 97–108, 1995.
- [PLS01] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. In *Mobile Computing and Networking - Mobicom*, pages 251–259, 2001.
- [Reg02] J. Regehr. Inferring Scheduling Behavior with Hourglass. In *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pages 143–156, 2002.
- [RT09] H. Rashidi and E.P.K. Tsang. An Efficient Extension of Network Simplex Algorithm. *Journal of Industrial Engineering*, 2:1–9, 2009.
- [SBKL11] N. Soule, A. Bestavros, A. Kfoury, and A. Lapets. Safe Compositional Equation-based Modeling of Constrained Flow Networks. In *Proc. of 4th Int’l Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, Zürich, September 2011.
- [SLM98] D.C. Schmidt, D.L. Levine, and S. Mungee. The Design of the tao real-time object request broker. *Computer Communications*, 21:294–324, 1998.
- [Sta00] J.A. Stankovic. VEST: A Toolset for Constructing and Analyzing Component Based Embedded Systems. In *Proc. EMSOFT01, LNCS 2211*, pages 390–402. Springer, 2000.