The syntax and semantics of a domain-specific language for flow-network design

Kfoury, Assaf
Computer Science Department, Boston University

The Syntax and Semantics of a Domain-Specific Language for Flow-Network Design

Assaf Kfoury

Computer Science Dept, Boston University, 111 Cummington Street, Boston, MA 02215, USA

Abstract

Flow networks are inductively defined, assembled from small components to produce arbitrarily large ones, with interchangeable functionally-equivalent parts. We carry out this induction formally using a domain-specific language (DSL). Associated with our DSL are a semantics and a typing theory. The latter gives rise to a system of formal annotations that enforce desirable properties of flow networks as invariants across their interfaces. A prerequisite for a typing theory is a formal semantics, i.e., a rigorous characterization of flows that are safe (or just feasible in this report) for the network, possibly restricted to satisfy additional efficiency or safety requirements. We give a detailed presentation of a denotational semantics only, but also point out the elements that an equivalent operational semantics must include.

Keywords: network specification, flow conservation, capacity constraint, typing, vector space

1. Introduction and Motivation

The background leading to the research reported herein is a little unusual. The motivation comes from the modeling and analysis of software systems that are assembled in an incremental and modular way. We devote some space in this introduction to explain this background.

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 [1, 2, 3], specific scheduling and power parameters [4, 5, 6, 7], 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 [8]. 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 interrelated 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 \((M_1 \parallel M_2)\) and the other of the form \(\text{bind} (N, \langle a, b \rangle)\). The first juxtaposes two networks \(M_1\) and \(M_2\) in parallel, and the second binds an output port \(a\) to an input port \(b\) in a network \(N\). 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: \((\text{let } X = M \text{ in } N)\), which says “network \(M\) may be safely placed in the occurrences of hole \(X\) in network \(N\)”. 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.\(^1\)

**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 adapted 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. We give a detailed presentation of the denotational approach only, but also point out the elements that an equivalent operational approach must include, so that an equivalence can be established between the two.

We prefer the denotational approach for several reasons, one of which being 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.

\(^1\)Holes as placeholders have been used in other formal environments for design and analysis, such as in Susan (a text templating language tied to the object-oriented modeling languages Modelica and MetaModelica [9, 10, 11, 12]). However, these other uses of holes are different from ours in several respects. In particular, they do not involve types and typings that set conditions at hole interfaces/boundaries that must be satisfied for safe placement in the holes.
**Paper Organization and Scope.** 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. Section 4 defines the formal semantics of flow networks. Sections 5, 6, and 7, present a type theory based on the syntax and semantics of the preceding sections.

For illustrative purposes, we consider only one 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.

This paper presents the bare bones of a relatively small DSL for the purpose at hand. The concluding section, Section 8, discusses various extensions of the syntax, the semantics, and the invariant properties that a type theory may enforce.

The technical background presumed by the paper is familiarity with standard formalisms to define the syntax and semantics of programming languages, familiarity with conventions and notions of type systems for programming languages, and some knowledge of vector spaces up to and including optimization of linear functions (using any of the standard algorithms for linear programming).

No implementation issues of any of the algorithms, whether directly formulated or invoked, are taken up in this paper. In particular, we leave an analysis of time and space requirements to a subsequent report.

**Acknowledgment.** The work reported hereinafter is a fraction of a collective effort involving several people, under the umbrella of the iBench Initiative at Boston University, co-directed by Azer Bestavros and the author. An earlier version of the DSL in this paper, with its formal semantics and type system, was introduced in our work for NetSketch, an integrated environment for the modeling, design and analysis of large-scale safety-critical systems with interchangeable parts [13, 14, 15]. 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**

What we call a “small network” is not necessarily small in size. It refers to a fully-identified component, i.e., one without holes, in a larger network config-

---

2 The website https://sites.google.com/site/ibenchbu/ gives a list of other research activities.
uration. Because it is fully identified, it can be completely analyzed in isolation, though we may want to limit its size in order to make its analysis tractable.

Formally, a small network $A$ is of the form $A = (N, A)$ where $N$ is a set of nodes and $A$ a set of directed arcs. Capacities on arcs are determined by a function $U : A \to \mathbb{R}^+$. 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 $a \in A$ by writing $\text{head}(a)$ and $\text{tail}(a)$, with the understanding that flow moves from $\text{tail}(a)$ to $\text{head}(a)$. The set $A$ of arcs is the disjoint union (denoted “$\sqcup$”) of three sets: the set $A_{\#}$ of internal arcs, the set $A_{\text{in}}$ of input arcs, and the set $A_{\text{out}}$ of output arcs:

$$A = A_{\#} \sqcup A_{\text{in}} \sqcup A_{\text{out}}$$

where

$$A_{\#} := \{ a \in A \mid \text{head}(a) \in N \text{ and } \text{tail}(a) \in N \}$$

$$A_{\text{in}} := \{ a \in A \mid \text{head}(a) \in N \text{ and } \text{tail}(a) \notin N \}$$

$$A_{\text{out}} := \{ a \in A \mid \text{head}(a) \notin N \text{ and } \text{tail}(a) \in N \}$$

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

A flow $f$ in $A$ assigns a non-negative real number to every $a \in A$. Formally, a flow $f : A \to \mathbb{R}^+$, 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 partial function $T$ (possibly total) that assigns types to some (possibly all) subsets of input and output arcs.\footnote{Our notion of a “typing” as an assignment of types to 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 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 $M$.} Formally, $T$ is of the following form, where $A_{\text{in, out}} := A_{\text{in}} \cup A_{\text{out}}$:

$$T : \mathcal{P}(A_{\text{in, out}}) \to \mathbb{R} \times \mathbb{R}$$

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

Informally, a typing $T$ imposes restrictions on the values of a flow $f$ at the external arcs $A_{\text{in, out}}$ of $A$ which, if satisfied, should guarantee that $f$ is “safe”
for $\mathcal{A}$. Specifically, if $T(\mathcal{A}) = [r, r']$, then $T$ requires that the part of $f$ entering through the arcs in $A \cap A_{in}$ minus the part of $f$ exiting through the arcs in $A \cap A_{out}$ must be within the interval $[r, r']$.

2.1. 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 $\mathcal{A}$ is a subset of arcs in $\mathcal{A}$ and $f$ a flow in $\mathcal{A}$, we write $\sum f(\mathcal{A})$ to denote the sum of flows assigned to all arcs in $\mathcal{A}$:

$$\sum f(\mathcal{A}) = \sum f(a) \mid a \in \mathcal{A}.$$

By convention, $\sum \emptyset = 0$. If $\mathcal{A} = \{a_1, \ldots, a_p\}$ is the set of all arcs entering node $\nu$, and $\mathcal{B} = \{b_1, \ldots, b_q\}$ is the set of all arcs exiting node $\nu$, then conservation of flow at $\nu$ is expressed by the linear equation:

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

For later reference, let $\mathcal{E}(\mathcal{A})$ denote the set of all such equations.

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

$$f(a) \leq U(a) \quad \text{(one such inequality for every arc } a \in \mathcal{A})$$

For later reference, let $\mathcal{C}(\mathcal{A})$ denote the set of all such inequalities.

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

- for every node $\nu \in N$, the equation in (1) is satisfied,
- for every arc $a \in \mathcal{A}$, the inequality in (2) is satisfied.

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

$$r \leq \sum f(A \cap A_{in}) - \sum f(A \cap A_{out}) \leq r'$$

6
3. DSL for Incremental and Modular Flow-Network Design (Untyped)

Our networks in general are assembled from the “small networks” introduced in Section 2 and what we call “holes”. Formally, a hole $X$ is a pair $(A_{in}, A_{out})$ where $A_{in}$ and $A_{out}$ are finite sets of input and output arcs. A hole $X$ is a placeholder where networks can be inserted, provided the matching-dimensions condition (in Section 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 $X = M$ in $N$`, which informally says “$M$ may be safely placed in the occurrences of hole $X$ in $N$”. What “safely” means depends on the invariant properties that typings are formulated to enforce.

If $A = (N, A)$ is a small network where $A = A_{#} \cup A_{in} \cup A_{out}$, let $in(A) = A_{in}$, $out(A) = A_{out}$, and $\#(A) = A_{#}$. Similarly, if $X = (A_{in}, A_{out})$ is a hole, let $in(X) = A_{in}$, $out(X) = A_{out}$, 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 $bind$. The BNF is shown in Figure 1, where $in(N)$ and $out(N)$ are the input and output arcs of $N$. We define the input arcs, output arcs, and internal

\[
\begin{array}{ll}
A, B \in \text{SMALL NETWORK} \\
X, Y \in \text{HOLE NAME} \\
M, N \in \text{NETWORK} \\
& ::= A \quad \text{small network name} \\
& | X \quad \text{hole name} \\
& | M \parallel N \quad \text{parallel connection} \\
& | \text{let } X = M \text{ in } N \quad \text{let-binding of hole } X \\
& | \text{bind }(N, \{a, b\}) \quad \text{bind } \text{head}(a) \text{ to } \text{tail}(b), \text{ where } \\
& \quad \{a, b\} \in \text{out}(N) \times \text{in}(N)
\end{array}
\]
arcs of a network specification $\mathcal{N}$, simultaneously by induction on $\mathcal{N}$:

$$
\left( \text{in}(\mathcal{N}), \text{out}(\mathcal{N}), \#(\mathcal{N}) \right) :=
\begin{cases}
\left( \text{in}(A), \text{out}(A), \#(A) \right) & \text{if } \mathcal{N} \text{ is the name of small network } A, \\
\left( \text{in}(X), \text{out}(X), \#(X) \right) & \text{if } \mathcal{N} \text{ is the name of hole } X, \\
\left( \text{in}(\mathcal{M}) \cup \text{in}(\mathcal{M}'), \text{out}(\mathcal{M}) \cup \text{out}(\mathcal{M}'), \#(\mathcal{M}) \cup \#(\mathcal{M}') \right) & \text{if } \mathcal{N} = (\mathcal{M} \parallel \mathcal{M}'), \\
\left( \text{in}(\mathcal{M}'), \text{out}(\mathcal{M}'), \#(\mathcal{M}) \cup \#(\mathcal{M}) \right) & \text{if } \mathcal{N} = (\text{let } X = M \text{ in } \mathcal{M}'), \\
\left( \text{in}(\mathcal{M}) - \{b\}, \text{out}(\mathcal{M}) - \{a\}, \#(\mathcal{M}) \cup \{a\} \right) & \text{if } \mathcal{N} = \text{bind } (\mathcal{M}, \langle a, b \rangle), \text{ with head}(a) := \text{tail}(b).
\end{cases}
$$

We say a flow network $\mathcal{N}$ is closed if every hole $X$ in $\mathcal{N}$ is bound. We say $\mathcal{N}$ is totally closed if it is closed and $\text{in}(\mathcal{N}) = \text{out}(\mathcal{N}) = \emptyset$, i.e., $\mathcal{N}$ has no input arcs and no output arcs.

**Remark 5.** A network specification $\mathcal{N}$, as defined by the BNF above, does not introduce capacities on arcs. $\mathcal{N}$ only defines a topology of a large network, starting from a collection of small networks. Capacities are introduced when we set up a formal semantics of network specifications and a corresponding typing theory.

Our typing theory will attempt to infer typings for all the well-formed subparts (or subexpressions) of a network specification $\mathcal{N}$ and for $\mathcal{N}$ itself. If it succeeds to do this inference, the typings will certify that the construction of every larger part from smaller parts respects the invariant properties we wish to impart to all of $\mathcal{N}$.

Among invariant properties, we will want, at a minimum, that if there are feasible flows in the smaller parts, then there are feasible flows in the larger parts.

### 3.1. Derived Constructors

From the three constructors already introduced, namely: $\parallel$, $\text{let}$, and $\text{bind}$, we can define several other constructors. Below, we present a sample of four derived constructors precisely, and mention several others in Remark 7. 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:

$$
\begin{align*}
\text{bind} \left( \mathcal{N}, \theta \right) & \quad \text{conn} \left( \mathcal{N}_1, \mathcal{N}_2, \theta \right) & \quad \mathcal{N}_1 \oplus \mathcal{N}_2 & \quad \text{let } X \in \{ \mathcal{M}_1, \ldots, \mathcal{M}_n \} \text{ in } \mathcal{N}
\end{align*}
$$
The second above depends on the first, and the third on the second. The fourth is strictly a shorthand, rather than a derived constructor, and independent of the three preceding.

Let \( \mathcal{N} \) be a network specification. We write \( \theta \subseteq_{1-1} \text{out}(\mathcal{N}) \times \text{in}(\mathcal{N}) \) to denote a partial one-one map from out(\( \mathcal{N} \)) to in(\( \mathcal{N} \)). We may write the entries in \( \theta \) explicitly, as in: \( \theta = \{(a_1, b_1), \ldots, (a_k, b_k)\} \) where \( \{a_1, \ldots, a_k\} \subseteq \text{out}(\mathcal{N}) \) and \( \{b_1, \ldots, b_k\} \subseteq \text{in}(\mathcal{N}) \).

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 \( (a, b) \in \text{out}(\mathcal{N}) \times \text{in}(\mathcal{N}) \). The expression \( \text{bind}(\mathcal{N}, \theta) \) is expanded as follows:

\[
\text{bind}(\mathcal{N}, \theta) = \text{bind} (\text{bind} (\ldots \text{bind} (\mathcal{N}, (a_k, b_k)) \ldots, (a_2, b_2)), (a_1, b_1))
\]

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 \( \theta \subseteq_{1-1} \text{out}(\mathcal{N}_1) \times \text{in}(\mathcal{N}_2) \). We expand the expression \( \text{conn}(\mathcal{N}_1, \mathcal{N}_2, \theta) \) as follows:

\[
\text{conn}(\mathcal{N}_1, \mathcal{N}_2, \theta) = \text{bind} ((\mathcal{N}_1 \parallel \mathcal{N}_2), \theta)
\]

In words, \( \text{conn} \) connects some 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 \( \text{conn} \). 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:

\[
\text{out}(\mathcal{N}_1) = \{a_1, \ldots, a_m\} \quad \text{and} \quad \text{in}(\mathcal{N}_2) = \{b_1, \ldots, b_m\}
\]

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 out(\( \mathcal{N}_1 \)) and in(\( \mathcal{N}_2 \)) are listed, from left to right, in the assumed ordering of their output and input dimensions, respectively. Let

\[
\theta = \{(a_1, b_1), \ldots, (a_m, b_m)\} = \text{out}(\mathcal{N}_1) \times \text{in}(\mathcal{N}_2)
\]

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:

\[(N_1 \oplus N_2) \Rightarrow \text{conn}(N_1, N_2, \theta)\]

which implies that \(\text{in}(N_1 \oplus N_2) = \text{in}(N_1)\) and \(\text{out}(N_1 \oplus N_2) = \text{out}(N_2)\). As expected, \(\oplus\) is associative as far as our formal semantics and typing theory are concerned, \(i.e.,\) the semantics and typings for \(N_1 \oplus (N_2 \oplus N_3)\) and \((N_1 \oplus N_2) \oplus N_3\) are the same.

A fourth derived constructor generalizes \textbf{let} and is expanded into nested \textbf{let}-bindings:

\[
\begin{align*}
(\text{let } X \in \{M_1, \ldots, M_n\} \text{ in } N) & \Rightarrow \\
(\text{let } X_1 = M_1 \text{ in } (\cdots (\text{let } X_n = M_n \text{ in } (N_1 \parallel \cdots \parallel N_n)) \cdots))
\end{align*}
\]

where \(X_1, \ldots, X_n\) are fresh hole names and \(N_i\) is \(N\) with \(X_i\) substituted for \(X\), for every \(1 \leq i \leq n\). Informally, this constructor says that every one of the networks \(\{M_1, \ldots, M_n\}\) can be “safely” placed in the occurrences of \(X\) in \(N\).

The fourth derived constructor is strictly a shorthand, because the expression on the left of “\(\Rightarrow\)” cannot be “plugged” wherever the expression on the right of “\(\Rightarrow\)” can. If \(N\) has \(k\) input arcs and \(\ell\) output arcs, which are also the input and output arcs of the expression on the left, then the expression on the right has \(n \cdot k\) input arcs and \(n \cdot \ell\) output arcs. Hence, the input/output dimensions of the two expressions are different (unless \(n = 1\)). This will not cause a problem if we keep in mind that the expression on the left is just a shorthand for the formal specification on the right.

\textbf{Remark 6.} For graphical representations of constructions such as \textbf{bind} \((N, \theta)\) and \(N_1 \oplus N_2 \oplus N_3\), the order in which we connect the arcs in the graphs does not matter, obviously. But we will invoke graphical representations only informally. To formally translate our network specifications into graphical representations in some unique normal form – which requires not only expanding all derived constructors but also, more challengingly, introducing formal rules to reduce all \textbf{let}-bindings – is the basis of an \textit{operational} (or \textit{reduction}) approach to the semantics of network specifications. However, this is something we try to avoid, for reasons we further elaborate in Remark 14.

\textbf{Remark 7.} While the preceding derived constructors are expanded using our primitive constructors, not every useful constructor can be so expanded. For example, the constructor

\[
\text{try } X \in \{M_1, \ldots, M_n\} \text{ in } N
\]
which we take to mean that at least one $M_i$ can be “safely” placed in all the occurrences of $X$ in $N$, 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

\[
\text{mix } X \in \{M_1, \ldots, M_n\} \text{ in } N
\]

which we take to mean that a mixture of several $M_i$ can be selected at the same time and “safely” placed in the occurrences of $X$ in $N$, generally placing different $M_i$ in different occurrences. An informal understanding of how they differ from the constructor let can be gleaned from Example 9.

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

\[
\text{letrec } X = M[X] \text{ in } N[X]
\]

where we write $M[X]$ to indicate that $X$ occurs free in $M$, and similarly in $N$. Informally, this construction corresponds to placing an open-ended network of the form $M[M[M[\ldots]]]$ in the occurrences of $X$ in $N$. A well-formedness condition here is that the input and output dimensions of $M$ must match those of $X$.

The constructors try, mix, and letrec, will be part of a follow-up report, including their formal semantics and typing rules.

### 3.2. Well-Formed Network Specifications

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.

These three conditions are automatically satisfied by small networks. Although they could be incorporated into an inductive definition of the formal syntax, more general than the BNF in Figure 1, they would obscure the relatively simple structure of our network specifications.
Matching dimensions of input/output arcs

Let \( \mathcal{M} \) be a network specification. As already mentioned in the definition of the derived constructor \( \oplus \), we assume there is a fixed ordering of the entries in \( \text{in}(\mathcal{M}) \) and \( \text{out}(\mathcal{M}) \). More explicitly now, if we need to refer to both together, we agree that the arcs in \( \text{in}(\mathcal{M}) \) are listed before those in \( \text{out}(\mathcal{M}) \):

\[
\dim_{\text{in}}(\mathcal{M}) \text{ is } \text{in}(\mathcal{M}) \text{ as an ordered set -- input dimension of } \mathcal{M}.
\]

\[
\dim_{\text{out}}(\mathcal{M}) \text{ is } \text{out}(\mathcal{M}) \text{ as an ordered set -- output dimension of } \mathcal{M}.
\]

\[
\dim(\mathcal{M}) = \dim_{\text{in}}(\mathcal{M}) \cdot \dim_{\text{out}}(\mathcal{M}) \text{ is } \text{in}(\mathcal{M}) \cup \text{out}(\mathcal{M}) \text{ as an ordered set -- I/O dimension of } \mathcal{M}.
\]

In the \textbf{let}-binding of a hole \( X \) we must guarantee that the network considered for insertion in \( X \) has the same number of input arcs, the same number of output arcs, and both are ordered in the same way. More precisely, an expression of the form:

\[
\textbf{let } X = \mathcal{M} \textbf{ in } N
\]

is well-formed provided:

\[
\dim_{\text{in}}(X) \approx \dim_{\text{in}}(\mathcal{M}) \text{ and } \dim_{\text{out}}(X) \approx \dim_{\text{out}}(\mathcal{M})
\]

where “\( \approx \)” indicates that the first arc, second arc, etc., in \( X \) correspond to the first arc, second arc, etc., in \( \mathcal{M} \). Keep in mind that arcs are named differently in \( X \) and in \( \mathcal{M} \), which is why we write “\( \approx \)” instead of “\( \approx \)”. If the preceding condition is satisfied, we will say that \( X \) and \( \mathcal{M} \) have similar input and output dimensions. Thus, when we place \( \mathcal{M} \) in hole \( X \), we connect the designated first arc in \( X \) to the designated first arc in \( \mathcal{M} \), the designated second arc in \( X \) to the designated second arc in \( \mathcal{M} \), etc.

Moreover, if there are several, say \( k \geq 2 \), occurrences of \( X \) in \( N \), we want each of the \( k \) copies of \( X \) to have its distinct set of input arcs and distinct set of output arcs, as we discuss next.

Unique arc naming

We need to guarantee that, in the specification of a network \( N \), no arc name refers to two different arcs. This is needed in order to avoid ambiguities in the formal semantics and the typing theory later. This condition is not enforced by the BNF definition in Figure 1, but we can enforce it by appropriate “isomorphic renaming”, \textit{i.e.}, by renaming arc names in order to avoid a same name for several arcs without changing the topology of the network, as we explain next.
We first define the outer scope and inner scope of a let-binding for a hole $X$ in a network specification $\mathcal{N}$: the inner scope is the part of $\mathcal{N}$ where all the bound occurrences of $X$ are mentioned, here indicated by an underbrace:

$$\mathcal{N} = \underbrace{\cdots \left( \text{let } X = \cdots \text{ in } \cdots X \cdots \right)}_{\text{outer scope}} \cdots \underbrace{\cdots \left( \text{let } Y = \cdots \text{ in } \cdots Y \cdots \right)}_{\text{outer scope}}$$

Inner scopes may be disjoint, as in:

$$\mathcal{N} = \cdots \left( \text{let } X = \cdots \text{ in } \cdots X \cdots \right) \cdots \left( \text{let } Y = \cdots \text{ in } \cdots Y \cdots \right) \cdots$$

and they may be nested, as in:

$$\mathcal{N} = \cdots \left( \text{let } X = \cdots \text{ in } \cdots X \cdots \left( \text{let } Y = \cdots \text{ in } \cdots Y \cdots \right) \cdots \right) \cdots$$

We need to distinguish the arcs of the different copies of the same hole $X$ within the inner scope of $X$. Thus, if we use $k \geq 2$ copies of $X$ within the same scope, we rename their arcs so that each copy has its own set of arcs. We write $^1 X, \ldots, ^k X$ to refer to these $k$ copies of $X$. However, we do not rename the corresponding binding occurrence of $X$. Thus, the two last of the three schematic representations above should be written as:

$$\mathcal{N} = \cdots \left( \text{let } X = \cdots \text{ in } \cdots ^1 X \cdots \right) \cdots \left( \text{let } Y = \cdots \text{ in } \cdots ^1 Y \cdots \right) \cdots$$

$$\mathcal{N} = \cdots \left( \text{let } X = \cdots \text{ in } \cdots ^1 X \cdots \left( \text{let } Y = \cdots \text{ in } \cdots ^1 Y \cdots \right) \cdots \right) \cdots$$

As we also keep track of the fact that $^1 X, \ldots, ^k X$ are all copies of $X$, there will be no ambiguity about which holes in $\mathcal{N}$ this binding occurrence of $X$ refers to.

In addition to the preceding, the unique arc-naming condition requires that, if a network specification $\mathcal{N}$ mentions $k \geq 2$ copies of the same small network $\mathcal{A}$, then each copy has its own separate set of arc names. Put differently, $\mathcal{N}$ mentions a small network $\mathcal{A}$ at most once, though it may mention several other small networks that are all isomorphic to $\mathcal{A}$.

One binding-occurrence for every hole $X$

For well-formedness we also require that, for every hole $X$, there is at most one let-binding for $X$, i.e., there is at most one binding occurrence of $X$. This condition disallows specifications $\mathcal{N}$ that are of the form:

$$\mathcal{N} = \cdots \left( \text{let } X = \cdots \text{ in } \cdots X \cdots \right) \cdots \left( \text{let } X = \cdots \text{ in } \cdots X \cdots \right) \cdots$$
where there are two let-bindings of $X$ for two disjoint scopes. And it disallows specifications $N$ of the form:

$$N = \cdots \left( \text{let } X = \cdots \text{ in } \cdots X \cdots \left( \text{let } X = \cdots \text{ in } \cdots X \cdots X \cdots \right) \cdots X \cdots \right) \cdots$$

where there are two let-bindings of $X$ for two nested scopes.

We are mostly interested in analyzing closed network specifications and determining their safety properties. Observe that, for a closed network specification $N$, the one binding-occurrence condition disallows the presence of subexpressions in $N$ of the form:

$$\cdots \left( \text{let } X = \cdots \text{ in } \cdots X \cdots X \cdots \right) \cdots$$

where the $X$ indicated by the upward arrow is outside the inner scope of the binding occurrence of $X$.

**Remark 8.** Of the three conditions for well-formedness, the matching-dimensions is the only one required for setting up the topology correctly of larger networks from their smaller components.

The other two conditions, unique arc-naming and one binding-occurrence, are introduced for the purposes of the formal semantics and the typing theory later; and of these two, one binding-occurrence can be omitted, but at the cost of unduly complicating things.

**Example 9.** We illustrate several notions. We use one hole $X$, and four small networks: $F$ (“fork”), $M$ (“merge”), $A$, and $B$. We do not assign capacities to the arcs of $F$, $M$, $A$, and $B$, because they play no role before the formal semantics and the typing theory are introduced. Graphic representations of $F$, $M$, and $X$, are shown in Figure 2, and of $A$ and $B$ in Figure 3. A possible network specification $N$ with two bound occurrences of $X$ may read as follows:

$$N := \text{let } X \in \{A, B\} \text{ in conn}( F, \text{conn}( \text{conn}( X, \text{conn}( X, \text{conn}( X, \text{conn}( X, \theta_3), \theta_2), \theta_1)$$

where

$$\theta_1 = \{\langle c_2, e_1 \rangle, \langle c_3, e_2 \rangle\}$$

$$\theta_2 = \{\langle e_3, c_1 \rangle, \langle e_4, c_2 \rangle\}$$

$$\theta_3 = \{\langle d_3, d_1 \rangle, \langle d_4, d_2 \rangle\}$$

The left superscripts are renaming indices, as mandated by the unique arc-naming condition above. We wrote $N$ using some derived constructors introduced in Section 3.1. We can write $N$ even more succinctly by noting that:
• all output arcs \{c_2, c_3\} of F are connected to all input arcs \{1e_1, 2e_2\} of 1X,
• all output arcs \{1e_3, 1e_4\} of 1X are connected to all input arcs \{2e_1, 2e_2\} of 2X,
• all output arcs \{2e_3, 2e_4\} of 2X are connected to all input arcs \{d_1, d_2\} of M.

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

\[ N := \text{let } X \in \{A, B\} \text{ in } (F \oplus 1X \oplus 2X \oplus M) \]

with now \(\text{in}(N) = \{c_1\}\) and \(\text{out}(N) = \{d_3\}\). The specification \(N\) says that \(A\) or \(B\) can be selected for insertion wherever hole \(X\) occurs. Informally, \(N\) can be viewed as representing two different network configurations:

\[ N_1 := F \oplus 1A \oplus 2A \oplus M \quad \text{and} \quad N_2 := F \oplus 1B \oplus 2B \oplus M \]

We can say nothing here about properties, such as safety, being satisfied or violated by \(N_1\) and \(N_2\). The semantics of our \text{let} constructor later will be equivalent to requiring that both configurations be “safe” to use. By contrast, the constructor \text{try} mentioned in Remark 7 requires only \(N_1\) or \(N_2\), but not necessarily both, to be safe. The constructor \text{mix} requires \(N_1\) and \(N_2\) to be safe, and additionally:

\[ N_3 := F \oplus 1A \oplus 2B \oplus M \quad \text{and} \quad N_4 := F \oplus 1B \oplus 2A \oplus M \]

to be safe. Safe substitution into holes according to \text{mix} implies safe substitution according to \text{let}, which in turn implies safe substitution according to \text{try}.

![Figure 2: Network F (left), network M (middle), and hole X (right), in Example 9.](image)

4. Formal Semantics of Flow Networks

The preceding section explained what we need to write to specify the topology of a flow network \(N\) with interchangeable or replaceable components, the latter being modeled by the presence of holes and \text{let}-bindings. No semantics was
A semantics for $\mathcal{N}$, denoted $[\mathcal{N}]$, is the result of imposing constraints on flows. In this paper, $[\mathcal{N}]$ is the collection of all feasible flows in $\mathcal{N}$ – or, more precisely, all feasible flows in a fully expanded version of $\mathcal{N}$, which we call “normal form” in Definition 10.4.

In general, $[\mathcal{N}]$ is an infinite set. But the whole point of setting up a typing theory is to avoid having to compute $[\mathcal{N}]$ explicitly.

We choose a syntax-directed inductive definition of $[\mathcal{N}]$, which requires that we start from the smallest parts in $\mathcal{N}$, namely holes and the components we identify as small networks.

By well-formedness, every small network $A$ appearing in $\mathcal{N}$ has its own separate set of arc names, and every bound occurrence $iX$ 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 $A$, we associate two sets of functions, its full semantics $[A]$ and its IO-semantics $\langle A \rangle$. Let $A_{\text{in,out}} = \text{in}(A) \cup \text{out}(A)$, and $A = A_{\text{in,out}} \cup \#(A)$.

4When the only constraints are upper-bound capacities, $[\mathcal{N}]$ is never empty, as it always includes the zero flow, which is the flow assigning 0 to all the arcs. We cannot therefore identify the notion of “safe to use” with $[\mathcal{N}] \neq \emptyset$. In this paper, we take that network $\mathcal{N}$ is safe to use if $[\mathcal{N}]$ includes flows that reach a minimum threshold mandated by the application or the user.

More generally, what is safe depends on the application modeled by flow networks. For example, if there are greater-than-zero lower-bound thresholds on the arcs, in addition to upper-bound capacities, then the zero flow is not feasible for such an application. In this case, the network $\mathcal{N}$ is safe to use if it allows non-zero flows respecting both lower-bound thresholds and upper-bound capacities.
sets $[A]$ and $⟨A⟩$ are defined thus:

$$[A] = \{ f : A \to \mathbb{R}^+ \mid f \text{ is a feasible flow in } A \}$$

$$⟨A⟩ = \{ f : A_{\text{in,out}} \to \mathbb{R}^+ \mid f \text{ can be extended to a feasible flow } f' \text{ in } A \}$$

We stress that we do not need to compute the infinitely many members of $[A]$ and $⟨A⟩$ explicitly. For example, in the case when the only constraints are upper-bound capacities on arcs, as in this paper, we can use the max-flow min-cut theorem, or an adaptation of it, to decide whether there exists a feasible flow in $A$ and to limit ourselves to computing only one value: the maximum attained by any feasible flow.\(^5\)

Let $X$ be a hole, with $A_{\text{in,out}} = \text{in}(X) \cup \text{out}(X)$ and $A = A_{\text{in,out}}$ because $\#(X) = \emptyset$. The full semantics $[X]$ and IO-semantics $⟨X⟩$ are the same set of functions:

$$[X] = ⟨X⟩ \subseteq \{ f : A_{\text{in,out}} \to \mathbb{R}^+ \mid f \text{ is a bounded function} \}$$

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

Proceeding inductively, we can define $[N]$ and $⟨N⟩$ simultaneously. For conciseness, we prefer to define $[N]$ first, and then define $⟨N⟩$ from $[N]$. Let $M$ and $N$ be network specifications, with:

$$\text{in}(M) \cup \text{out}(M) \cup \#(M) = \{ a_1, \ldots, a_p \}$$

$$\text{in}(N) \cup \text{out}(N) \cup \#(N) = \{ b_1, \ldots, b_q \}.$$  

We have $\{ a_1, \ldots, a_p \} \cap \{ b_1, \ldots, b_q \} = \emptyset$ by the unique arc-naming condition in Section 3.2. If $f \in [M]$ and $g \in [N]$, with $f(a_1) = r_1, \ldots, f(a_p) = r_p$ and $g(b_1) = s_1, \ldots, g(b_q) = s_q$, we may represent $f$ and $g$ by the sequences $\langle r_1, \ldots, r_p \rangle$ and $\langle s_1, \ldots, s_q \rangle$, respectively. We define $f \parallel g$ as follows:

$$(f \parallel g) := \langle r_1, \ldots, r_p \rangle \cdot \langle s_1, \ldots, s_q \rangle = \langle r_1, \ldots, r_p, s_1, \ldots, s_q \rangle$$

where “$\cdot$” is sequence concatenation. The operation “$\parallel$” on flows is associative, but not commutative, just as the constructor “$\parallel$” on network specifications. We\(^5\)

\(^5\) In optimization theory, flow networks are usually considered to have a single input (source) and a single output (sink). The max-flow min-cut theorem asserts that, in a flow network, the maximum amount of flow passing from source to sink is equal to the value of a minimum-capacity cut. Because we consider flow networks with multiple inputs and multiple outputs, the classical max-flow min-cut theorem cannot be applied immediately and has to be adapted appropriately.
define the full semantics $[[N]]$ by induction on the structure of the specification $N$, as shown in Figure 4. Some finer points in the 5 clauses of Figure 4:

**Clause 2.** All bound occurrences $iX$ of the same hole $X$ are assigned the same semantics $[[X]]$, up to renaming of arc names.

**Clause 4.** By matching-dimensions condition in Section 3.2, $\dim(X) \approx \dim(M)$ means that 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 $M$, up to arc renaming. $[[X]] \approx \{[[g]]_A | g \in [[M]]\}$ means that for every $f : \text{in}(X) \cup \text{out}(X) \rightarrow \mathbb{R}^+$, it holds that $f \in [[X]]$ iff there is $g \in [[M]]$ such that $f \approx [[g]]_A$. We write $[[g]]_A$ for the restriction of the function $f$ to the subset $A$.

**Clause 5.** If $f(a) = f(b)$, then it must be that $f(a) \leq \min \{U(a), U(b)\}$. Thus, when we bind output arc $a$ to input arc $b$ and make $\text{head}(a) := \text{tail}(b)$, we update the upper-bound capacity on $a$ by setting $U(a) := \min\{U(a), U(b)\}$.

1. If $N = A$, then $[[N]] := [[A]]$.
2. If $N = iX$, then $[[N]] := i[[X]]$.
3. If $N = (M_1 \mid M_2)$, then $[[N]] := \{ (f_1 \mid f_2) \mid f_1 \in [[M_1]] \text{ and } f_2 \in [[M_2]] \}$.
4. If $N = (\text{let } X = M \text{ in } M')$, then $[[N]] := [[M']]$, provided two conditions:
   (a) $\dim(X) \approx \dim(M)$.
   (b) $[[X]] = \{ [[g]]_A | g \in [[M]] \}$ where $A = \text{in}(M) \cup \text{out}(M)$.
5. If $N = \text{bind } (M, \langle a, b \rangle)$, then $[[N]] := \{ f \mid f \in [[M]] \text{ and } f(a) = f(b) \}$.

**Figure 4:** Formal Semantics of Flow Network Specifications.

We now define the IO-semantics of $N$ as follows:

$$[[N]] = \{ [[f]]_A | f \in [[N]] \}$$

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

**4.1. Canonical Forms and Normal Forms**

We define an intermediate formal syntax which will facilitate the transition from the semantics of network specifications to their typings.
Figure 5 is the formal syntax of what we call network specifications in canonical form. Clearly, every specification according to the BNF in Figure 5 is a specification according to the BNF in Figure 1, but not the other way around. A specification in canonical form places all let-bindings in outermost position. Schematically, a canonical form has the following shape:

```plaintext
let X_1 = R_1 in
let X_2 = R_2 in
...,
let X_k = R_k in bind(\mathcal{P}_1 \parallel \mathcal{P}_2 \parallel ... \parallel \mathcal{P}_{\ell}, \theta)
```

where every \( R_i \) is again a canonical form possibly containing let-bindings, for every \( 1 \leq i \leq k \), and every member of \( \{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_{\ell}\} \) is a small network or a hole. As used here, bind is the derived constructor defined in Section 3.1.

The 3 conditions for the well-formedness of network specifications in general, as discussed in Section 3.2, apply to network specifications in canonical form without any modification.

```
A, B \in \text{SMALL NETWORK}
X, Y \in \text{HOLE NAME}
\mathcal{P}, \mathcal{Q} \in \text{LET FREE}
::= A \quad \text{small network name}
| \ X \quad \text{hole name}
| \mathcal{P} \parallel \mathcal{Q} \quad \text{parallel connection}
| \operatorname{bind}(\mathcal{P}, \langle a, b \rangle) \quad \text{bind head}(a) \text{ to tail}(b), \text{ where }
\quad \langle a, b \rangle \in \text{out}(\mathcal{P}) \times \text{in}(\mathcal{P})
\mathcal{R}, \mathcal{S} \in \text{CANONICAL FORM} ::= \mathcal{P} \quad \text{let-free network specification}
| \text{let } X = \mathcal{R} \text{ in } \mathcal{S} \quad \text{let-binding of hole } X
```

**Figure 5:** Formal Syntax of Canonical Network Specifications.

**Definition 10.** A network specification is in normal form if it is a closed let-free specification in canonical form.

Note carefully our metavariable conventions: A network specification in general is denoted by the letter \( \mathcal{M} \) or \( \mathcal{N} \) (possibly decorated), while a let-free specification is denoted by the letter \( \mathcal{P} \) or \( \mathcal{Q} \) (possibly decorated), and a specification
in canonical form is denoted by the letter $R$ or $S$ (possibly decorated). Since a normal form is a closed let-free specification, we also use $P$ and $Q$ to denote normal-form specifications.

By the preceding definition, a normal form is assembled from small networks – and no holes, since it is closed – using the constructors $\parallel$ and bind only. Hence, the only thing distinguishing small networks from normal forms is size, in case there is a size limit on small networks (typically for reasons of practical implementation). Otherwise, if there is no limit on their size, we can identify small networks with normal form specifications.

**Lemma 11.** Let $P$ be a normal-form specification assembled from the small networks $A_1,\ldots,A_k$ for some $k \geq 1$. We then have:

$$\llbracket P \rrbracket = \begin{cases} f : \text{in}(P) \cup \text{out}(P) \cup \#(P) \to \mathbb{R}^+ \mid f \text{ satisfies } \bigcup_{1 \leq i \leq k} (\mathcal{E}(A_i) \cup \mathcal{C}(A_i)) \end{cases}$$

where $\text{in}(P)$, $\text{out}(P)$, and $\#(P)$, are the input, output, and internal arcs, of $P$ as defined in Section 3, and $\mathcal{E}(A_i) \cup \mathcal{C}(A_i)$ are the equations and inequalities enforcing flow conservation and capacity constraints in $A_i$, as given in Definitions 1 and 2.

**Proof.** This is straightforward by the observations preceding the lemma. A formal proof is by induction on $k \geq 1$. \qed

By Lemma 11, we now have a direct formula to define the full semantics of a normal-form specification $P$ built up from the small networks $A_1,\ldots,A_k$:

$$\mathcal{E}(P) := \bigcup_{1 \leq i \leq k} \mathcal{E}(A_i),$$
$$\mathcal{C}(P) := \bigcup_{1 \leq i \leq k} \mathcal{C}(A_i),$$
$$\llbracket P \rrbracket = \{ f : \text{in}(P) \cup \text{out}(P) \cup \#(P) \to \mathbb{R}^+ \mid f \text{ satisfies } \mathcal{E}(P) \cup \mathcal{C}(P) \}.$$  

**Lemma 12.** Let $R$ be a closed canonical-form specification with exactly one let-binding. We can write a normal-form specification $P$ such that $\llbracket P \rrbracket = \llbracket R \rrbracket$.

Note that $\llbracket P \rrbracket = \llbracket R \rrbracket$ implies both $\text{in}(P) = \text{in}(R)$ and $\text{out}(P) = \text{out}(R)$. We cannot write $\llbracket P \rrbracket = \llbracket R \rrbracket$ in the lemma statement, because $\#(P)$ is not necessarily the same as $\#(R)$, as the proof makes clear.

**Proof.** Suppose the let-binding in $R$ is $(\text{let } X = Q_1 \text{ in } Q_2)$, the latter being necessarily closed because there are no other let-binding in $R$. This also implies
that both $Q_1$ and $Q_2$ must be let-free, and $Q_1$ is necessarily closed and therefore a normal-form specification. While the specification $R$ may be larger than $(\text{let } X = Q_1 \text{ in } Q_2)$, i.e., schematically:

$$R = \cdots (\text{let } X = Q_1 \text{ in } Q_2) \cdots$$

it suffices to show that there is a normal-form specification $Q_3$ such that $[Q_3] = [\text{let } X = Q_1 \text{ in } Q_2]$ and then proceed as in the proof of Lemma 11 to conclude this proof. The desired $Q_3$ is simply obtained by substituting $Q_1$ for every occurrence of $X$ in $Q_2$, with a different renaming index for each occurrence of $X$.

**Theorem 13.** Let $N$ be a closed network specification. We can write a normal-form network specification $P$ such that $\llbracket N \rrbracket = \llbracket P \rrbracket$ with $\text{in}(N) = \text{in}(P)$ and $\text{out}(N) = \text{out}(P)$.

As in the statement of Lemma 12, the fact that $\llbracket N \rrbracket = \llbracket P \rrbracket$ implies both $\text{in}(N) = \text{in}(P)$ and $\text{out}(N) = \text{out}(P)$. We cannot write $[N] = [P]$ because, in general, $\#(N)$ is not the same as $\#(P)$.

**Proof.** There are different approaches to this result. A perspicuous proof considers the abstract syntax tree of $N$, call it $\text{AST}(N)$, and then proceed by induction on the number $k \geq 0$ of let-bindings in $N$. If $k = 0$, then $N$ is already a normal-form specification. If $k \geq 1$, it suffices to show how we can find an equivalent closed network specification $N'$ with $(k - 1)$ let-bindings.

Following the original syntax in Figure 1, we think of $\text{AST}(N)$ as a tree with its root at the top and where every internal node is labelled:

- “$\parallel$” with two branches $M$ and $M'$ corresponding to $(M \parallel M')$, or
- “bind $(a, b)$” with one branch $M$ corresponding to $\text{bind}(M, (a, b))$, or
- “@” with a left branch whose root is “let $X$” and below which is $M'$ and a right branch $M$ corresponding to $(\text{let } X = M \text{ in } M')$.

In the case of the let-binding for $X$, we put $M'$ (not $M$) under it, because $M$ cannot contain occurrences of $X$. Every leaf node in $\text{AST}(N)$ is labelled with a hole name or a small-network name.

Because $\text{AST}(N)$ is finite and $N$ is closed, there must be a subexpression in $N$ of the form $(\text{let } X = Q_1 \text{ in } Q_2)$ where both $Q_1$ and $Q_2$ are let-free, $Q_1$ is closed, and $Q_2$ contains no free occurrences of holes other than $X$. Such a subexpression can be found by starting at the root of $\text{AST}(N)$ and traversing it in reverse post-order, looking for the innermost and rightmost occurrence of @.

Once such a subexpression $(\text{let } X = Q_1 \text{ in } Q_2)$ is identified, we proceed as in Lemma 12 to reduce the number of let-bindings by one.
Remark 14. The expansions of \texttt{let}-bindings in the proofs of Lemma 12 and Theorem 13 do not yet define a \textit{reduction} (or \textit{rewrite}) system for network specifications. Strictly speaking, these proofs only establish the existence of normal-form specifications, without explicitly specifying reduction rules. Further elaboration is required to set up such rules and provide the basis of an \textit{operational semantics} for our DSL.

This can be done (not in this report), which will in turn call for a proof of \textit{soundness} of the operational semantics relative to the denotational semantics in Figure 4. Soundness will show that the denotational semantics of network specifications are an \textit{invariant} of the reduction.

We avoid presenting an operational semantics and the underlying reduction rules. For our purposes, a denotational semantics is more flexible and more closely mimicked by the typing theory (as a sound “approximation” of the exact semantics) in Section 5 and later.

4.2. Flow Conservation, Capacity Constraints, Type Satisfaction (Continued)

We extend the fundamental concepts stated in relation to small networks $\mathcal{A}$ in Definitions 1, 2, and 3, to arbitrary network specifications $\mathcal{N}$.

Let $\mathcal{N}$ be a closed network specification and let $\mathcal{P}$ be the normal-form specification obtained from $\mathcal{N}$ according to Theorem 13. We have $\text{in}(\mathcal{N}) = \text{in}(\mathcal{P})$ and $\text{out}(\mathcal{N}) = \text{out}(\mathcal{P})$. Let

\[ A_{\text{in,out}} := \text{in}(\mathcal{N}) \cup \text{out}(\mathcal{N}) = \text{in}(\mathcal{P}) \cup \text{out}(\mathcal{P}). \]

As noted earlier, $\#(\mathcal{N})$ is not necessarily equal to $\#(\mathcal{P})$. Let $A := A_{\text{in,out}} \cup \#(\mathcal{N})$ and $A' := A_{\text{in,out}} \cup \#(\mathcal{P})$. We can define a flow in $\mathcal{N}$ as a map from $A$ to $\mathbb{R}^+$, or else as a map from $A'$ to $\mathbb{R}^+$. These are two different definitions, but which coincide when we consider flows restricted to $A_{\text{in,out}}$. It is a little more convenient to take a flow in $\mathcal{N}$ as a map from $A'$ to $\mathbb{R}^+$.

Definition 15. Let $\mathcal{N}$ be a closed network specification and $\mathcal{P}$ be a normal-form specification obtained from $\mathcal{N}$ according to Theorem 13. As in the preceding paragraph, let:

\[ A_{\text{in,out}} := \text{in}(\mathcal{N}) \cup \text{out}(\mathcal{N}) = \text{in}(\mathcal{P}) \cup \text{out}(\mathcal{P}) \]

and $A' = A_{\text{in,out}} \cup \#(\mathcal{P})$. We identify the set of flows in $\mathcal{N}$ with those in $\mathcal{P}$, i.e., a flow is a function $f : A' \to \mathbb{R}^+$, and not a function $f : A \to \mathbb{R}^+$.

We make a distinction between “flows” and “input-output functions”. An input-output function, or just \texttt{IO} function, in $\mathcal{N}$ (and $\mathcal{P}$) is a map $g : A_{\text{in,out}} \to \mathbb{R}^+$.  

22
The two next lemmas establish the necessary connections between \( N \) and \( P \). Their proofs are immediate from the definitions and therefore omitted. Lemma 17 also extends the notion of feasibility to IO functions.

**Lemma 16.** Let \( N \) be closed network specification and \( P \) its normal form as in Definition 15. For every flow \( f : A' \rightarrow \mathbb{R}^+ \), the three following assertions are equivalent:

1. \( f \) is feasible in \( P \).
2. \( f \in \llbracket P \rrbracket \).
3. \( f \) satisfies \( \mathcal{E}(P') \cup \mathcal{E}(P) \).

**Lemma 17.** Let \( N \) be closed network specification and \( P \) its normal form as in Definition 15. For every IO function \( g : A_{\text{in,out}} \rightarrow \mathbb{R}^+ \), the four following assertions are equivalent:

1. \( g \in \llbracket N \rrbracket \).
2. \( g \in \llbracket P \rrbracket \).
3. There is a flow \( f \in \llbracket N \rrbracket \) such that \( g = [f]_{A_{\text{in,out}}} \).
4. There is a flow \( f \in \llbracket P \rrbracket \) such that \( g = [f]_{A_{\text{in,out}}} \).

If any of these four assertions holds, we will say that the IO function \( g \) is feasible (in both \( N \) and \( P \)).

The following definition extends Definition 4 to flows in network specifications in general and also adapts it to IO functions.

**Definition 18.** Let \( N \) be closed network specification and \( P \) its normal form as in Definition 15. A typing \( T : \mathcal{P}(A_{\text{in,out}}) \rightarrow \mathbb{R} \times \mathbb{R} \) for \( N \) is defined independently of the internal arcs. Hence, \( T \) is also a typing for its normal form \( P \).

We say the flow \( f : A' \rightarrow \mathbb{R}^+ \), resp. the IO function \( g : A_{\text{in,out}} \rightarrow \mathbb{R}^+ \), satisfies \( T \) iff for every \( A \in \mathcal{P}(A_{\text{in,out}}) \) for which \( T(A) \) is defined and \( T(A) = [r, r'] \), it is the case that:

\[
    r \leq \sum f(A \cap A_{\text{in}}) - \sum f(A \cap A_{\text{out}}) \leq r'
\]

resp.

\[
    r \leq \sum g(A \cap A_{\text{in}}) - \sum g(A \cap A_{\text{out}}) \leq r'
\]
5. Typings Are Polytopes

Let \( \mathcal{N} \) be a network specification, and let \( A_{\text{in}} := \text{in}(\mathcal{N}) \), \( A_{\text{out}} := \text{out}(\mathcal{N}) \), and \( A_{\text{in, out}} := A_{\text{in}} \cup A_{\text{out}} \). Let \( A_{\text{in}} = \{a_1, \ldots, a_m\} \) and \( A_{\text{out}} = \{a_{m+1}, \ldots, a_{m+n}\} \) for some \( m \geq 1 \) and \( n \geq 1 \). As usual, there is a fixed ordering on the arcs in \( A_{\text{in}} \) and again on the arcs in \( A_{\text{out}} \).

Let \( T \) be a typing for \( \mathcal{N} \) that assigns an interval \([r, r']\) to \( A \subseteq A_{\text{in, out}} \). With no loss of generality, suppose:

\[
A \cap A_{\text{in}} = \{a_1, \ldots, a_k\} \quad \text{and} \quad A \cap A_{\text{out}} = \{a_{m+1}, \ldots, a_{m+\ell}\},
\]

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

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

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

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

\[
(4) \quad T_\geq(A) : \quad a_1 + \cdots + a_k - a_{m+1} - \cdots - a_{m+\ell} \geq r \\
T_\leq(A) : \quad a_1 + \cdots + a_k - a_{m+1} - \cdots - a_{m+\ell} \leq r'
\]

and, therefore, two halfspaces \( \text{Half}(T_\geq(A)) \) and \( \text{Half}(T_\leq(A)) \) in \( \mathbb{R}^{m+n} \):

\[
(5) \quad \text{Half}(T_\geq(A)) = \{r \in \mathbb{R}^{m+n} | r \text{ satisfies } T_\geq(A)\} \\
\text{Half}(T_\leq(A)) = \{r \in \mathbb{R}^{m+n} | r \text{ satisfies } T_\leq(A)\}
\]

We can therefore define \( \text{Poly}(T) \) formally as follows:

\[
\text{Poly}(T) = \bigcap \left\{ \text{Half}(T_\geq(A)) \cap \text{Half}(T_\leq(A)) : \emptyset \neq A \subseteq A_{\text{in, out}} \text{ and } T(A) \text{ is defined} \right\}
\]

Generally, many of the inequalities induced by the typing \( T \) will be redundant, and the induced \( \text{Poly}(T) \) will be defined by far fewer than \( 2 \cdot (2^{m+n} - 1) \) halfspaces.

**Remark 19.** We agree that, in order for \( T : \mathcal{P}(A_{\text{in, out}}) \to \mathbb{R} \times \mathbb{R} \) to be a network typing, three requirements are satisfied:
1. \( T(\emptyset) = T(A_{\text{in,out}}) = [0, 0] = \{0\} \). Informally, this corresponds to \textit{global flow conservation}: The total amount entering a flow network must equal the total amount exiting it.

2. Poly\((T)\) must be a \textit{bounded} subspace of \( \mathbb{R}^{m+n} \) and therefore a polytope, and not just a polyhedron. That is, for every \( 1 \leq i \leq m + n \), there is an interval \([s, s']\), such that for every \( \langle r_1, \ldots, r_i, \ldots, r_{m+n} \rangle \in \text{Poly}(T) \), it must be that \( s \leq r_i \leq s' \). This is a mild restriction, obviating the need to deal separately with cases of unboundedly large flows. We therefore take “Poly()” to denote “polytope”, not “polyhedron”.

3. Poly\((T)\) is entirely contained within the \textit{first orthant} of the hyperspace \( \mathbb{R}^{m+n} \), \textit{i.e.}, the subspace \( (\mathbb{R}^+)^{m+n} \). This means that if \( \langle r_1, \ldots, r_{m+n} \rangle \in \text{Poly}(T) \) then every component \( r_i \) is non-negative, corresponding to the fact that if an IO function \( f : A_{\text{in,out}} \rightarrow \mathbb{R}^+ \) satisfies \( T \), then every entry in \( \langle f(a_1), \ldots, f(a_{m+n}) \rangle \) is non-negative.

Even assuming that the three preceding requirements are satisfied, not all network typings are “inhabited”, \textit{i.e.}, some are not typings of any flow networks. In a subsequent report [16], we characterize network typings \( T \) which are inhabited.

5.1. \textit{Uniqueness and Redundancy in Typings}

We can view a network typing \( T \) as a syntactic expression, with its semantics 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 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 set Constraints\((T)\) defined by:

\[
\text{Constraints}(T) := \{ T_\leq(A) \mid \emptyset \neq A \in \mathcal{P}(A_{\text{in,out}}) \text{ and } T(A) \text{ is defined} \} \\
\cup \{ T_\approx(A) \mid \emptyset \neq A \in \mathcal{P}(A_{\text{in,out}}) \text{ and } T(A) \text{ is defined} \}
\]

where \( T_\leq(A) \) and \( T_\approx(A) \) are as in (4) 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.

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 Poly\((T_1) \approx \text{Poly}(T_2)\), in which case we say that \( T_1 \) and \( T_2 \) are \textit{equivalent}.\(^6\) If \( \mathcal{N}_1 = \mathcal{N}_2 \), then \( T_1 \equiv T_2 \) (which

\(^6\)Recall the meaning of “\( \approx \)” for matching-dimensions in Section 3.2: Poly\((T_1) \approx \text{Poly}(T_2)\) means that Poly\((T_1)\) and Poly\((T_2)\) are the same up to the renaming of variables/arc names.
Definition 20. Let $T$ be a network typing over $A_{\text{in,out}}$. $T$ is tight if, for every $A \in \mathcal{P}(A_{\text{in,out}})$ for which $T(A)$ is defined and for every $r \in T(A)$, there is an IO function $f \in \text{Poly}(T)$ such that

$$r = \sum f(A \cap A_{\text{in}}) - \sum f(A \cap A_{\text{out}}).$$

Informally, $T$ is tight if none of the intervals/types assigned by $T$ to members of $\mathcal{P}(A_{\text{in,out}})$ contains redundant information.

Propositions 21 and 22 are simple facts about network typings that will be invoked in Section 7.2. Proposition 23 makes explicit connections between notions introduced earlier in the section.

Proposition 21. There is an algorithm $\text{TTight()}$ which, given a network typing $T$ as input, always terminates and returns an equivalent total and tight typing $\text{TTight}(T)$. Moreover, $\text{TTight}(T)$ is uniquely determined, i.e., every other network typing which is tight, total, and equivalent to $T$ is the same as $\text{TTight}(T)$.

Proof. Starting from the given typing $T : \mathcal{P}(A_{\text{in,out}}) \rightarrow \mathbb{R} \times \mathbb{R}$, we can compute $\text{Poly}(T)$ by specifying the corresponding set of linear inequalities/constraints $\text{Constraints}(T)$ as defined in (4) above. We compute a total and tight typing $T' : \mathcal{P}(A_{\text{in,out}}) \rightarrow \mathbb{R} \times \mathbb{R}$ by assigning an appropriate interval/type $T'(A)$ to every $A \in \mathcal{P}(A_{\text{in,out}})$ as follows. For such a set $A$ of input and output arcs, let $\theta_A$ be the objective function:

$$\theta_A = \sum A \cap A_{\text{in}} - \sum A \cap A_{\text{out}}.$$ 

Relative to $\text{Constraints}(T)$, using standard procedures of linear programming, we minimize and maximize the objective $\theta_A$ to obtain two values $r_1$ and $r_2$, respectively. The desired type $T'(A)$ is $[r_1, r_2]$ and the desired $\text{TTight}(T)$ is $T'$.

Proposition 22. Let $\mathcal{C}$ denote a finite set of linear constraints, over a set of input and output arcs $A_{\text{in,out}} = \{a_1, \ldots, a_{m+n}\}$, which defines a polytope $\Pi$ in the hyperspace $(\mathbb{R}^+)^{m+n}$ for some $m, n \geq 1$.

There is an algorithm $\text{TTyping()}$ of two arguments which, given any such constraint set $\mathcal{C}$ and non-empty $B \subseteq A_{\text{in,out}}$ as input, returns a total and tight

---

7Typically, $\mathcal{C}$ will be $\text{Constraints}(T)$ for some typing $T : \mathcal{P}(A_{\text{in,out}}) \rightarrow \mathbb{R} \times \mathbb{R}$ which may or may not be total and tight.
network typing $\mathbb{T} \text{Typing}(\mathcal{C}, B) : \mathcal{P}(B) \to \mathbb{R} \times \mathbb{R}$ which defines the least polytope
such that $\text{Poly}(\mathbb{T} \text{Typing}(\mathcal{C}, B)) \supseteq [\Pi]_B$ where

$$[\Pi]_B := \{ [r]_B \mid r \in \Pi \subseteq (\mathbb{R}^+)^{m+n} \}$$

where $[r]_B$ is the restriction of the $(m+n)$-dimensional vector $r$ to the coordinates appearing in $B$. Thus, $[\Pi]_B$ is just the projection of $\Pi$ on the subspace defined by the arcs/coordinates in $B$. Moreover, $\mathbb{T} \text{Typing}(\mathcal{C}, B)$ is uniquely determined.

Proof. Similar to the proof of Proposition 21, but simpler. Since the projection of a polytope is again a polytope, it suffices to prove the proposition for the case $B = A_{\text{in,out}}$. The desired typing $\mathbb{T} \text{Typing}(\mathcal{C}, A_{\text{in,out}}) : \mathcal{P}(A_{\text{in,out}}) \to \mathbb{R} \times \mathbb{R}$, which is both total and tight, is obtained by assigning an appropriate interval/type $\mathbb{T} \text{Typing}(\mathcal{C}, A_{\text{in,out}})(A)$ to every $A \in \mathcal{P}(A_{\text{in,out}})$ as follows. For such a set $A$ of input and output arcs, let $\theta_A$ be the objective function:

$$\theta_A = \sum A \cap A_{\text{in}} - \sum A \cap A_{\text{out}}.$$

Relative to $\mathcal{C}$, using standard procedures of linear programming, we minimize and maximize the objective $\theta_A$ to obtain two values $r_1$ and $r_2$, respectively. The desired type $\mathbb{T} \text{Typing}(\mathcal{C}, A_{\text{in,out}})(A)$ is $[r_1, r_2]$. □

Proposition 23. Let $T$ be a typing over the set $A_{\text{in,out}}$ of input and output arcs. Let $\text{Constraints}(T)$ be the corresponding set inequalities as defined in (6) above. Then $T$ is total and tight iff $\mathbb{T} \text{Typing}(\text{Constraints}(T)) = T$.

Proof. This is a straightforward consequence of the definitions, invoking also Propositions 21 and 22. □

5.2. Valid Typings and Principal Typings

Let $\mathcal{N}$ be a network specification with outer arcs $A_{\text{in,out}}$. Review Lemmas 16 and 17 and Definition 18. We say the typing $T : \mathcal{P}(A_{\text{in,out}}) \to \mathbb{R} \times \mathbb{R}$ is valid for $\mathcal{N}$ iff $T$ satisfies a soundness condition:

(soundness) Every IO function $f : A_{\text{in,out}} \to \mathbb{R}^+$ satisfying $T$ is feasible.

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

(completeness) Every feasible IO function $f : A_{\text{in,out}} \to \mathbb{R}^+$ satisfies $T$. 27
More succintly, the typing \( T \) is valid for \( \mathcal{N} \) iff \( \text{Poly}(T) \subseteq \llbracket \mathcal{N} \rrbracket \), and it is principal for \( \mathcal{N} \) iff \( \text{Poly}(T) = \llbracket \mathcal{N} \rrbracket \).

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:

\[
\text{(subtyping)} \quad T_1 <: T_2 \quad \text{iff} \quad \text{Poly}(T_2) \subseteq \text{Poly}(T_1).
\]

Our subtyping relation is contravariant w.r.t. the subset relation, i.e., the typing \( T_2 \) appears on the right of “\(<:"” but \( \text{Poly}(T_2) \) appears on the left of “\("\)”. In words, subtype \( T_1 \) is less restrictive than supertype \( T_2 \). The following is an immediate consequence of the definitions and we record it as a proposition for later reference.

**Proposition 24.** Let \( \mathcal{N} \) be a network specification. If \( T_1 \) is a principal typing for \( \mathcal{N} \) and \( T_2 \) is a valid typing for \( \mathcal{N} \), then \( T_1 <: T_2 \).

Any two principal typings \( T_1 \) and \( T_2 \) for the same network are not necessarily identical, but they always denote the same polytope, as formally stated in the next proposition. First, a lemma of more general interest.

**Lemma 25.** Let \( T_1 \) and \( T_2 \) be typings for the same flow network \( \mathcal{N} \). If \( T_1 \) and \( T_2 \) are tight and total, and \( \text{Poly}(T_1) = \text{Poly}(T_2) \), then \( T_1 = T_2 \).

**Proof.** This is a straightforward consequence of Proposition 21, where \( \text{Tight}(T) \) returns a typing which is total, tight, and equivalent to \( T \). \qed

**Proposition 26.** If \( T_1 \) and \( T_2 \) are two principal typings for the same network \( \mathcal{N} \), then \( T_1 \equiv T_2 \). Moreover, if \( T_1 \) and \( T_2 \) are tight and total, then \( T_1 = T_2 \).

**Proof.** Because they are principal, both \( T_1 \) and \( T_2 \) are valid. Hence, by Proposition 24, both \( T_1 <: T_2 \) and \( T_2 <: T_1 \). This implies that \( T_1 \equiv T_2 \). When \( T_1 \) and \( T_2 \) are tight and total, then the equality \( T_1 = T_2 \) follows from Lemma 25. \qed

Based on the facts so far, a total and tight typing \( T \), which is principal for a network specification \( \mathcal{N} \), plays a special role among all valid typings for \( \mathcal{N} \). It is the smallest element in a distributive lattice which we examine more carefully in a subsequent report [16].

6. Inferring Typings for Small Networks

The proof of the next theorem gives one method for computing a principal typing for a small network \( \mathcal{A} \), based on linear programming.
Theorem 27. Let $A$ be a small network. We can compute a principal typing $T$ for $A$, which is additionally tight and total.

Proof. Let the external arcs of $A$ be $A_{\text{in, out}} = A_{\text{in}} \cup A_{\text{out}}$, with $A_{\text{in}} = \{a_1, \ldots, a_m\}$ and $A_{\text{out}} = \{a_{m+1}, \ldots, a_{m+n}\}$, for some $m, n \geq 1$. The set of both internal and external arcs is $A = A_{\text{in}} \cup A_{\text{out}}$. Let $\mathcal{E}(A)$ be the collection of all equations enforcing flow conservation and $\mathcal{C}(A)$ the collection of all inequalities enforcing capacity constraints in $A$, as given in Definitions 1 and 2. $\mathcal{E}(A) \cup \mathcal{C}(A)$ are written over $A$, where arc names are used as variables.

We define the desired typing $T : \mathcal{P}(A_{\text{in, out}}) \rightarrow \mathbb{R} \times \mathbb{R}$ as follows. For every non-empty $A \in \mathcal{P}(A_{\text{in, out}})$, relative to the equations and inequalities in $\mathcal{E}(A) \cup \mathcal{C}(A)$, we use linear programming to minimize and maximize the same objective function:

$$\theta_A = \sum \{ a \mid a \in A \cap A_{\text{in}} \} - \sum \{ a \mid a \in A \cap A_{\text{out}} \}$$

Relative to $\mathcal{E}(A) \cup \mathcal{C}(A)$, the determination of the type/interval assigned to $T(A)$ is in three steps:

1. Compute the minimum possible value $r_1 \in \mathbb{R}$ for the objective $\theta_A$.
2. Compute the maximum possible value $r_2 \in \mathbb{R}$ for the objective $\theta_A$.
3. Assign to $T(A)$ the interval $[r_1, r_2]$.

Trivially, we also assign the empty type/interval to $T(\emptyset)$. The resulting $T$ is total because it assigns a type to every $A \in \mathcal{P}(A_{\text{in, out}})$. Moreover, $T$ is tight because $T(A)$ does not exceed the minimum and the maximum of $\theta_A$ allowed by $\mathcal{E}(A) \cup \mathcal{C}(A)$, for $A \in \mathcal{P}(A_{\text{in, out}})$. And $T$ is principal because every feasible flow in $A$ is such that $[f]_{A_{\text{in, out}}}$ is a “point” inside the polytope $\text{Poly}(T)$. \qed

Example 28. Consider the two small networks $A$ and $B$ from Example 9. We assign capacities to their arcs and compute their respective principal typings. The sets of arcs in $A$ and $B$ are, respectively: $A = \{a_1, \ldots, a_{11}\}$ and $B = \{b_1, \ldots, b_{16}\}$. The upper-bound capacity on every arc is a “very large number”, unless indicated otherwise in Figure 6 by the numbers in rectangular boxes, namely:

$$U(a_5) = 5, \quad U(a_8) = 10, \quad U(a_{11}) = 15, \quad \text{in } A,$n
$$U(b_5) = 3, \quad U(b_8) = 2, \quad U(b_{11}) = 10, \quad \text{in } B,$n
$$U(b_{13}) = 8, \quad U(b_{15}) = 10, \quad U(b_{16}) = 7, \quad \text{in } B.$n

We compute the principal typings $T_A$ and $T_B$, 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. Together with a standard package for linear programming (such as Matlab), we use the construction in the proof of Theorem 27 to compute $T_A$ and $T_B$.

$T_A$ assignments:

<table>
<thead>
<tr>
<th>$a_1 : [0, 15]$</th>
<th>$a_2 : [0, 25]$</th>
<th>$-a_3 : [-15, 0]$</th>
<th>$-a_4 : [-25, 0]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1 + a_2 : [0, 30]$</td>
<td>$a_1 - a_3 : [-10, 10]$</td>
<td>$a_1 - a_4 : [-25, 15]$</td>
<td>$a_2 - a_3 : [-15, 25]$</td>
</tr>
<tr>
<td>$a_2 - a_3 : [-15, 25]$</td>
<td>$a_2 - a_4 : [-10, 10]$</td>
<td>$-a_3 - a_4 : [-30, 0]$</td>
<td></td>
</tr>
</tbody>
</table>

$T_B$ assignments:

<table>
<thead>
<tr>
<th>$b_1 : [0, 15]$</th>
<th>$b_2 : [0, 25]$</th>
<th>$-b_3 : [-15, 0]$</th>
<th>$-b_4 : [-25, 0]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1 + b_2 : [0, 30]$</td>
<td>$b_1 - b_3 : [-10, 12]$</td>
<td>$b_1 - b_4 : [-23, 15]$</td>
<td>$b_2 - b_3 : [-15, 23]$</td>
</tr>
<tr>
<td>$b_2 - b_3 : [-15, 23]$</td>
<td>$b_2 - b_4 : [-12, 10]$</td>
<td>$-b_3 - b_4 : [-30, 0]$</td>
<td></td>
</tr>
</tbody>
</table>

The types in rectangular boxes are those of $[T_A]_{\text{in}}$ and $[T_B]_{\text{in}}$ which are equivalent, and those of $[T_A]_{\text{out}}$ and $[T_B]_{\text{out}}$ which are equivalent. Thus, $[T_A]_{\text{in}} \equiv [T_B]_{\text{in}}$ and $[T_A]_{\text{out}} \equiv [T_B]_{\text{out}}$.

Nevertheless, $T_A \neq T_B$, the difference being in the (underlined) types assigned to some subsets mixing input and output arcs. As a result, there are feasible flows in one which are not feasible in the other. For example, if we set:

- $f_0(a_1) = f_0(b_1) = 15$
- $f_0(a_2) = f_0(b_2) = 0$
- $f_0(a_3) = f_0(b_3) = 3$
- $f_0(a_4) = f_0(b_4) = 12$

it is easy to see that $f_0$ can be extended to a feasible flow $f$ in $B$ but not in $A$.

To conclude this example, every typing $T$ such that $\text{Poly}(T) \subseteq \text{Poly}(T_A)$ (resp. $\text{Poly}(T) \subseteq \text{Poly}(T_B)$) is valid for $A$ (resp. $B$). In particular, if we take the intersection of $\text{Poly}(T_A)$ and $\text{Poly}(T_B)$, with appropriate renaming of arc names/variables, we obtain:

$T$ assignments:

<table>
<thead>
<tr>
<th>$a_1 : [0, 15]$</th>
<th>$a_2 : [0, 25]$</th>
<th>$-a_3 : [-15, 0]$</th>
<th>$-a_4 : [-25, 0]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1 + a_2 : [0, 30]$</td>
<td>$a_1 - a_3 : [-10, 10]$</td>
<td>$a_1 - a_4 : [-23, 15]$</td>
<td>$a_2 - a_3 : [-15, 23]$</td>
</tr>
<tr>
<td>$a_2 - a_3 : [-15, 23]$</td>
<td>$a_2 - a_4 : [-10, 10]$</td>
<td>$-a_3 - a_4 : [-30, 0]$</td>
<td></td>
</tr>
</tbody>
</table>
where we again omit assignments by $T$ that are implied by those already listed. The underlined types in $T$ are those that are different from the corresponding types in $T_A$ or $T_B$. The resulting $T$ is valid for both $A$ and $B$, and is the “most general” (i.e., least restrictive) because it is the intersection of the principal $T_A$ and $T_B$.

Figure 6: An assignment of arc capacities for small networks $A$ and $B$ in Example 28.

7. Inferring Typings for Flow Networks in General

We set up a formal system for assigning typings to network specifications in general. Typings are polytopes and operations on them are adaptations of standard operations on polytopes in vector spaces.

Starting from a set of small networks, each assigned a total and tight typing, the typing rules will assign a total and tight typing to any network specification assembled from these small networks. In situations where the typing $T$ assigned to a small network is not total or tight, we first apply algorithm $TTight(\cdot)$ from Proposition 21 to obtain an equivalent total and tight $TTight(T)$ before we use the typing rules.

7.1. Operations on Typings

Let $T_1$ and $T_2$ be two total and tight typings for $N_1$ and $N_2$, respectively. The four arc sets: $\text{in}(N_1), \text{out}(N_1), \text{in}(N_2)$, and $\text{out}(N_2)$, are pairwise disjoint. By our inductive definition in Section 3, the set $\text{in}(N_1) \cup \text{in}(N_2)$ are the input arcs, and the set $\text{out}(N_1) \cup \text{out}(N_2)$ are the output arcs, of the network specification $(N_1 \parallel N_2)$. We define the typing $(T_1 \parallel T_2)$ for the specification $(N_1 \parallel N_2)$ as
This is immediate from the definitions. We omit the straightforward formal details. Informally, the specification \( \equiv (G \subseteq N) \)

follows:

\[
(T_1 \parallel T_2)(A) = \begin{cases} T_1(A) & \text{if } A \subseteq \text{in}(N_1) \cup \text{out}(N_1), \\ T_2(A) & \text{if } A \subseteq \text{in}(N_2) \cup \text{out}(N_2), \\ T_1(A_1) \oplus T_2(A_2) & \text{if } A = A_1 \cup A_2 \text{ where} \\ A_1 \subseteq \text{in}(N_1) \cup \text{out}(N_1) \text{ and} \\ A_2 \subseteq \text{in}(N_2) \cup \text{out}(N_2). 
\end{cases}
\]

where the operation “\( \oplus \)” on intervals/types is defined as follows:

\[
[r_1, r_2] \oplus [r_1', r_2'] = [r_1 + r_1', r_2 + r_2'].
\]

**Lemma 29.** Let \( T_1 \) and \( T_2 \) be total and tight typings for network specifications \( N_1 \) and \( N_2 \), respectively. If \( T_1 \) and \( T_2 \) are principal (resp. valid) for \( N_1 \) and \( N_2 \), then the typing \((T_1 \parallel T_2)\) is total, tight, and principal (resp. valid) for the network specification \((N_1 \parallel N_2)\). 

**Proof.** This is immediate from the definitions. We omit the straightforward formal details. Informally, the specification \((N_1 \parallel N_2)\) places \( N_1 \) and \( N_2 \) next to each other without making any connection between the two. All the mentioned properties – total, tight, and principal (or valid) – lift therefore from the components \( N_1 \) and \( N_2 \) to their non-interacting assembly \((N_1 \parallel N_2)\), which implies that \((T_1 \parallel T_2)\) is total, tight, and principal (or valid) for the latter. \( \square \)

Let \( T : \mathcal{P}(A_{\text{in,out}}) \to \mathbb{R} \times \mathbb{R} \) be a total and tight typing for specification \( N \), where \( A_{\text{in,out}} = A_{\text{in}} \cup A_{\text{out}} \) is the set of outer arcs in \( N \), with \( A_{\text{in}} = \{a_1, \ldots, a_m\} \) and \( A_{\text{out}} = \{a_{m+1}, \ldots, a_{m+n}\} \) for some \( m, n \geq 1 \). Let \( a \in A_{\text{out}} \) and \( b \in A_{\text{in}} \). We use algorithm \( \text{TTyping}(\cdot) \) from Proposition 22 to define the typing \( \text{bind}(T, \langle a, b \rangle) \):

\[
(8) \quad \text{bind}(T, \langle a, b \rangle) := \text{TTyping}(\text{Constraints}(T) \cup \{a = b\}, A_{\text{in,out}} - \{a, b\})
\]

Note that \( \text{Constraints}(T) \) defines the polytope, call it \( \Pi \), consisting of all “points” (i.e., IO functions) in the hyperspace \((\mathbb{R}^+)^{m+n}\) satisfying \( T \). By Proposition 23, \( \Pi = \text{Poly}(T) \). The constraint \( \{a = b\} \) defines a hyperplane cutting across \( \Pi \), implying that the polytope defined by \( \text{Constraints}(T) \cup \{a = b\} \), call it \( \Pi' \), is related to \( \Pi \) according to:

\[
\Pi' = \{ f : A_{\text{in,out}} \to \mathbb{R}^+ \mid f \in \Pi \text{ and } f(a) = f(b) \}
\]

Hence, the projection of the polytope \( \Pi' \) on the subset \( A_{\text{in,out}} - \{a, b\} \), call it \( \Pi'' \), consists of all the “points” (i.e., IO functions) satisfying \( \text{bind}(T, \langle a, b \rangle) \) as defined
in (8). Hence, \( \Pi'' = \text{Poly}(\text{bind}(T, \langle a, b \rangle)) \) and if \( T \) is principal (resp. valid) for network specification \( \mathcal{N} \), then \( \text{bind}(T, \langle a, b \rangle) \) is principal (resp. valid) for network specification \( \text{bind}(\mathcal{N}, \langle a, b \rangle) \).

**Lemma 30.** Let \( T \) be a total and tight typing for \( \mathcal{N} \), whose input set and output set are \( A_{\text{in}} \) and \( A_{\text{out}} \). For every pair \( \langle a, b \rangle \in A_{\text{out}} \times A_{\text{in}} \), if \( T \) is principal (resp. valid) for \( \mathcal{N} \), then \( \text{bind}(T, \langle a, b \rangle) \) is principal (resp. valid) for \( \text{bind}(\mathcal{N}, \langle a, b \rangle) \).

**Proof.** Straightforward consequence of the discussion preceding the lemma. \( \square \)

### 7.2. Typing Rules

The system is in Figure 7, 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 \( \text{in}(X) = A_{\text{in}} \) and \( \text{out}(X) = A_{\text{out}} \), then \( T : \mathcal{P}(A_{\text{in},\text{out}}) \rightarrow \mathbb{R} \times \mathbb{R} \).

In the rule **LET**, assumptions are discharged from the context \( \Gamma \). This is not essential, because we assume there is at most one binding occurrence for every hole. We discharge assumptions in the rule **LET** for conciseness and only to indicate which holes in a network specification remain unbound.

If a typing \( T \) is derived for a network specification \( \mathcal{N} \) according to the rules in Figure 7, 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. The side conditions in Figure 7 must be satisfied in order that the corresponding rules can be applied.

**Theorem 31.** Let \( \mathcal{N} \) be a closed network specification and \( T \) a typing for \( \mathcal{N} \) derived according to the rules in Figure 7, i.e., the judgment \( \vdash \mathcal{N} : T \) is derivable according to the rules. If the typing of every small network \( A \) in \( \mathcal{N} \) is total, tight, and principal (resp., valid) for \( A \), then \( T \) is total, tight, and principal (resp., valid) typing for \( \mathcal{N} \).

**Proof.** By induction on the number of rules used to derive the judgment \( \vdash \mathcal{N} : T \), invoking Lemma 29 every time the rule **PAR** is used and Lemma 30 every time the rule **BIND** is used. \( \square \)

Typing-inference algorithms can be set up based on the rules in Figure 7, which we outlined in different degrees of details in separate technical reports, some already presented in workshops [17, 18, 19].
\[ (X : T) \in \Gamma \quad \Gamma \vdash X : !T \quad i \geq 1 \text{ is the smallest available renaming index} \]

**HOLE**

\[ \begin{array}{ll}
\hline
\text{SMALL} & T \text{ is a total typing for small network } A \\

\hline
\Gamma \vdash A : T \\
\hline
\text{PAR} & \Gamma \vdash N_1 : T_1 \quad \Gamma \vdash N_2 : T_2 \\
\Gamma \vdash (N_1 \parallel N_2) : (T_1 \parallel T_2) \\
\hline
\text{BIND} & \Gamma \vdash N : T \\
\Gamma \vdash \text{bind}(N ; \langle a, b \rangle) : \text{bind}(T ; \langle a, b \rangle) \\
\langle a, b \rangle \in \text{out}(N) \times \text{in}(N) \\
\hline
\text{LET} & \Gamma \vdash M : T_1 \\
\Gamma \cup \{(X : T_2)\} \vdash N : T \\
\Gamma \vdash (\text{let } X = M \text{ in } N) : T \\
T_1 \approx T_2 \\
\hline
\end{array} \]

**Figure 7:** Typing Rules. The operations on typings, \((T_1 \parallel T_2)\) and \(\text{bind}(T, \langle a, b \rangle)\), are defined in Section 7.1.

8. Future Work and Conclusion

In several places earlier in the paper, we indicated possible extensions, in both the syntax and the semantics of our DSL. We elaborate a little further on this, in Section 8.1 below. The final Section 8.2 suggests future work of a different kind, by making connections with research by others which, though motivated by different concerns, shares some of the underlying formalisms and methodologies.

8.1. Extensions

For illustrative purposes in this paper, we examined only one kind of constraint on flows, namely, an upper-bound function \(U : A \to \mathbb{R}\) that assigns a maximum capacity to every arc \(a \in A\) in a network. A natural extension is to add a lower-bound function \(L : A \to \mathbb{R}\) that assigns a minimum threshold to every arc \(a \in A\). A “safe” flow is now a flow that must remain within the interval between the minimum \(L(a)\) and the maximum \(U(a)\) on every arc \(a \in A\). The semantics must be adjusted accordingly and the typings formulated to enforce this kind of safety across interfaces.

More interesting and challenging is to require that a flow, in order to be safe, 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 [17], all inspired by studies in the area of “traffic engineering” (see, e.g., [20] and references therein).

Another extension has to do with the typing system. As set up in Section 7.2, the typing rules are syntax-directed, and therefore modular, as they infer or assign typings to specifications 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 typing system is seamlessly compositional. We add the qualifier “seamlessly” to distinguish our notion of compositionality from similar, but different, notions in other areas of computer science. A direct typing-inference algorithm based on the rules of Section 7.2 is only modular, because in the process of inferring a typing for an expression of the form \((\text{let } X = M \text{ in } N)\), the most natural approach is to (1) infer a typing \(T_1\) for \(M\) first, and (2) assuming a typing \(T_2 \approx T_1\) for \(X\), infer a typing \(T\) for \(N\) second. Inferring a typing for \(N\) must wait until a typing for \(M\) is determined. Hence, even though syntax-directed and modular, this approach follows a strict order in which typings are inferred for the constituent parts. Appropriate adjustments must be introduced in the typing system in Section 7.2, and the inference algorithms based on it, in order to support seamless compositionality.

8.2. Related 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 [21] 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 [22].

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 [22, 23], 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 [24], i.e., linear programming applied to problems of network flows. There is still on-going research to improve network-simplex algorithms.

---

\[8\] Adding to the imprecision of the word, “compositional” in the literature is sometimes used in the more restrictive sense of “modular” in our sense.
(e.g., [25]), which will undoubtedly have a bearing on the efficiency of typing inference for our DSL.

As alluded earlier in Remarks 6 and 14, we (mostly) avoided getting involved in the details of an operational semantics for our DSL in this paper, primarily 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 explicit 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 7 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 [26, 27, 28, 29, 30] 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


