A Different Approach to the Design and Analysis of Network Algorithms

Assaf Kfoury[*]
Boston University
kfoury@bu.edu

Saber Mirzaei[*]
Boston University
smirzaei@bu.edu

March 4, 2013

Abstract

We review elements of a typing theory for flow networks, which we expounded in an earlier report [19]. To illustrate the way in which this typing theory offers an alternative framework for the design and analysis of network algorithms, we here adapt it to the particular problem of computing a maximum-value feasible flow. The result of our examination is a max-flow algorithm which, for particular underlying topologies of flow networks, outperforms other max-flow algorithms. We point out, and leave for future study, several aspects that will improve the performance of our max-flow algorithm and extend its applicability to a wider class of underlying topologies.

[*]Partially supported by NSF award CCF-0820138.

Contents

1 Introduction

Background and motivation. The background to this report is a group effort to develop an integrated enviroment for system modeling and system analysis that are simultaneously: *modular* ("distributed in space"), *incremental* ("distributed in time"), and *order-oblivious* ("components can be modeled and analyzed in *any* order"). These are the three defining properties of what we call a seamlessly *compositional* approach to system modeling and system analysis.[1] Several papers explain how this environment is defined and used, as well as its current state of development and implementation [3, 4, 5, 17, 18, 24]. An extra fortuitous benefit of our work on system modeling has been a fresh perspective on the design and analysis of network algorithms.

To illustrate our methodology, we consider the classical *max-flow problem*. A solution for this problem is an algorithm which, given an arbitrary network $\mathcal{N}$ with one source node s and one sink node t, computes a maximal feasible flow f from s to t in $\mathcal{N}$. That f is a *feasible flow* means f is an assignment of non-negative values to the arcs of $\mathcal{N}$ satisfying *capacity constraints* at every arc and *flow conservation* at every node other than s and t. That f is *maximal* means the net outflow at node s (equivalently, the net inflow at node t) is maximized. A standard assessment of a max-flow algorithm measures its run-time complexity as a function of the size of $\mathcal{N}$. Our methodology is broader, in that it can be applied again to tackle other network-related problems with different measures of what qualifies as an acceptable solution.

Starting with the algorithm of Ford and Fulkerson in the 1950's [9], several different solutions have been found for the max-flow problem, based on the same fundamental concept of *augmenting path*. A refinement of the augmenting-path method is the *blocking flow* method [7], which several researchers have used to devise better-performing algorithms. Another family of max-flow algorithms uses the so-called *preflow push* method (also called *push relabel* method), initiated by Goldberg and Tarjan in the 1980's [10, 14]. A survey of these families of max-flow algorithms to the end of the 1990's can be found in several reports [2, 11]. Further developments introduced variants of the augmenting-path algorithms and the closely related blocking-flow algorithms, variants of the preflow-push algorithms, and algorithms combining different parts of all of these methodologies [12, 13, 21, 22]. More recently, an altogether different approach to the max-flow problem uses the notion of *pseudoflow* [6, 15].

The design and analysis of any of the forementioned algorithms presumes that the given network $\mathcal{N}$ is known in its entirety. No design of an algorithm and its analysis are undertaken until all the pieces (nodes, arcs, and their capacities) are in place. We may therefore qualify such an approach to design and analysis as a *whole-network* approach.

Overview of our methodology. The central concept of our approach is what we call a *network typing*. To make this work, a network (or network component) $\mathcal{N}$ is allowed to have "dangling arcs"; in effect, $\mathcal{N}$ is allowed to have multiple sources or *input arcs* (*i.e.*, arcs whose tails are not incident on any node) and multiple sinks or *output arcs* (*i.e.*, arcs whose heads are not incident on any node). Given a network $\mathcal{N}$, now with multiple input arcs and multiple output arcs, a typing for $\mathcal{N}$ is a formal algebraic characterization of all the feasible flows in $\mathcal{N}$ – including, in particular, all maximal feasible flows.

More precisely, a *valid* typing T for network $\mathcal{N}$ specifies conditions on input/output arcs of $\mathcal{N}$ such that every assignment f of values to the input/output arcs satisfying these conditions can be extended to a feasible flow g in $\mathcal{N}$. Moreover, if the input/output conditions specified by T are satisfied by *every* input/output assignment f extendable to a feasible flow g, then we say that T is not only valid but also *principal* for $\mathcal{N}$.

In our formulation, a typing T for network $\mathcal{N}$ defines a bounded convex polyhedral set (or *polytope*), which we denote $\mathsf{Poly}(T)$, in the vector space $\mathbb{R}^{k+\ell}$, where $\mathbb{R}$ is the set of reals, k the number of input arcs in $\mathcal{N}$, and ℓ the number of output arcs in $\mathcal{N}$. An input/output function f satisfies T if f viewed as a point in the space $\mathbb{R}^{k+\ell}$

[1]This is one of the projects currently in progress under the umbrella of the iBench Initiative at Boston University, co-directed by Azer Bestavros and Assaf Kfoury. The website `https://sites.google.com/site/ibenchbu/` gives further details on this and other research activities.

is inside $\mathsf{Poly}(T)$. Hence, T is a *valid typing* (resp. *principal typing*) for $\mathcal{N}$ if $\mathsf{Poly}(T)$ is *contained in* (resp. *equal to*) the set of all input/output functions that can be extended to feasible flows.[2]

Let T_1 and T_2 be principal typings for networks $\mathcal{N}_1$ and $\mathcal{N}_2$. If we connect $\mathcal{N}_1$ and $\mathcal{N}_2$ by linking some of their output arcs to some of their input arcs, we obtain a new network which we denote (only in this introduction) $\mathcal{N}_1 \oplus \mathcal{N}_2$. One of our results shows that the principal typing of $\mathcal{N}_1 \oplus \mathcal{N}_2$ can be obtained by direct (and relatively easy) algebraic operations on T_1 and T_2, without any need to re-examine the internal details of the two components $\mathcal{N}_1$ and $\mathcal{N}_2$. Put differently, an analysis (to produce a principal typing) for the assembled network $\mathcal{N}_1 \oplus \mathcal{N}_2$ can be directly and easily obtained from the analysis of $\mathcal{N}_1$ and the analysis of $\mathcal{N}_2$.

What we have just described is the counterpart of what type theorists of programming languages call a *modular* (or *syntax-directed*) *analysis* (or *type inference*) – which infers a type for the whole program from the types of its subprograms, and the latter from the types of their respective subprograms, and so on recursively, down to the types of the smallest program fragments.

Because our network typings denote polytopes, we can in fact make our approach not only modular but also *compositional*, in the following sense. If T_1 and T_2 are principal typings for networks $\mathcal{N}_1$ and $\mathcal{N}_2$, then neither T_1 nor T_2 depends on the other; that is, the analysis (to produce T_1) for $\mathcal{N}_1$ and the analysis (to produce T_2) for $\mathcal{N}_2$ can be carried out independently of each other without knowledge that the two will be subsequently assembled together.[3]

Given a network $\mathcal{N}$ partitioned into finitely many components $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3, \ldots$ with respective principal typings $T_1, T_2, T_3, \ldots$, we can then assemble these typings in *any* order – first in pairs, then in sets of four, then in sets of eight, etc. – to obtain a principal typing T for the whole of $\mathcal{N}$. Efficiency in computing the final principal typing T depends on a judicious partitioning of $\mathcal{N}$, which is to decrease as much as possible the number of arcs running between separate components, and again recursively when assembling larger components from smaller components. At the end of this procedure, every input/output function f extendable to a maximal feasible flow g in $\mathcal{N}$ can be directly read off the final typing T – but observe: not g itself.

In contrast to the prevailing *whole-network* approaches, we call ours a *compositional* approach to the design and analysis of max-flow algorithms.

Highlights. Our main contribution is therefore a different framework for the design and analysis of network algorithms, which we here illustrate by presenting a new algorithm for the classical problem of computing a maximum flow. Our final algorithm combines several intermediate algorithms, each of independent interest for computing network typings. We mention some salient features distinguishing our approach from others:

1. As formulated in this report and unlike other approaches, our final algorithm returns only the *value* of a maximum flow, without specifying a set of actual *paths* from source to sink that will carry such a flow. Other approaches handle the two problems simultaneously: Inherent in their operation is that, in order to compute a maximum-flow *value*, they need to determine a set of maximum-flow *paths*; ours does not need to. Though avoided here because of the extra cost and complications for a first presentation, our final algorithm can be adjusted to return a set of maximum-flow paths in addition to a maximum-flow value.

[2]**Note on terminology**: Our choice of the names "type" and "typing" is not coincidental. They refer to notions in our examination which are equivalent to notions by the same names in the study of strongly-typed programming languages. The type system of a strongly-typed language – object-oriented such as Java, or functional such as Standard ML or Haskell – consists of formal logical annotations enforcing safety conditions as invariants across interfaces of program components. In our examination here too, "type" and "typing" will refer to formal annotations (now based on ideas from linear algebra) to enforce safety conditions (now limited to *feasibility* of flows) across interfaces of network components. We take a flow to be *safe* iff it is feasible.

[3]In the study of programming languages, there are syntax-directed, inductively defined, type systems that support modular but not compositional analysis. What is compositional is modular, but not the other way around. A case in point is the so-called Hindley-Milner type system for ML-like functional languages, where the order matters in which types are inferred.

2. We view the uncoupling of the two problems just described as an advantage. It underlies our need to be able to replace components – broken or defective – by other components as long as their principal typings are equal, without regard to how they may direct flow internally from input ports to output ports.

3. As far as run-time complexity is concerned, our final algorithm performs very badly on some networks, *e.g.*, networks whose graphs are dense. However, on other special classes of networks, ours outperforms the best currently available algorithms (*e.g.*, on networks whose graphs are outer-planar or whose graphs are topologically equivalent to some ring graphs).

4. In all cases, our algorithms do not impose any restrictions on flow capacities, in contrast to some of the best-performing algorithms of other approaches. In this report, flow capacities can be arbitrarily large or small, independent of each other, and not restricted to integral values.

5. Our final algorithm, just like all the intermediate algorithms on which it depends, does not rely on any standard linear-programming procedure. More precisely, although our final algorithm carries out some linear optimization, *i.e.*, minimizing or maximizing linear objectives relative to linear constraints, these objectives and constraints are so limited in their form that optimization need not use anything more than addition, subtraction, comparison of numbers, and very simple reasoning of elementary linear algebra.

Organization of the report. Sections 2, 3, and 4, are background material, where we fix our notation regarding standard notions of flow networks as well as introduce new notions regarding typings.

Section 5 presents a simple, but expensive, algorithm for computing the principal typing of an arbitrary flow network $\mathcal{N}$, which we call WholePT. It provides a point of comparison for algorithms later in the report. WholePT is our only algorithm that operates in "whole-network" mode, in the sense explained above, and that produces its result using standard linear-programming procedures.

In Sections 6, 7, and 8, we present our methodology for breaking up a flow network $\mathcal{N}$ into one-node components at an initial stage, and then gradually re-assembling $\mathcal{N}$ from these components. This part of the report includes algorithms for producing principal typings of one-node networks, and then producing the principal typings of intermediate network components, each obtained by re-connecting an arc that was disconnected at the initial stage.

Section 9 presents algorithm CompPT which combines the algorithms of the preceding three sections and computes the principal typing of a flow network $\mathcal{N}$ in "compositional" mode. In addition to $\mathcal{N}$, algorithm CompPT takes a second argument, which we call a *binding schedule*; a binding schedule σ dictates the order in which initially disconnected arcs are re-connected and, as a result, determines the run-time complexity of CompPT which, if σ is badly selected, can be excessive.

Algorithm CompMaxFlow in Section 10 calls CompPT as a subroutine to compute a maximum-flow value. The run-time complexity of CompMaxFlow therefore depends on the binding schedule σ that is used as the second argument in the call to CompPT.

Acknowledgments. The work reported herein 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 Assaf Kfoury. The website `https://sites.google.com/site/ibenchbu/` gives a list of current and past participants, and research activities. Several iBench participants were a captive audience for partial presentations of the included material, in several sessions over the last two years. Special thanks are due to them all.

2 Flow Networks

We repeat standard notions of flow networks [1] using our notation and terminology. We take a *flow network* $\mathcal{N}$ as a pair $\mathcal{N} = (\mathbf{N}, \mathbf{A})$, where $\mathbf{N}$ is a finite set of nodes and $\mathbf{A}$ a finite set of directed arcs, with each arc connecting two distinct nodes (no self-loops). We write $\mathbb{R}$ and $\mathbb{R}_+$ for the sets of reals and non-negative reals, respectively. Such a flow network $\mathcal{N}$ is supplied with *capacity functions* on the arcs:

- Lower-bound capacity $\underline{\boldsymbol{c}} : \mathbf{A} \to \mathbb{R}_+$.
- Upper-bound capacity $\overline{\boldsymbol{c}} : \mathbf{A} \to \mathbb{R}_+$.

We assume $0 \leqslant \underline{\boldsymbol{c}}(a) \leqslant \overline{\boldsymbol{c}}(a)$ and $\overline{\boldsymbol{c}}(a) \neq 0$ for every $a \in \mathbf{A}$. We identify the two ends of an arc $a \in \mathbf{A}$ by writing *tail*(a) and *head*(a), with the informal understanding that flow "moves" from *tail*(a) to *head*(a). The set $\mathbf{A}$ of arcs is the disjoint union – written "$\uplus$" whenever we want to make it explicit – of three sets: the set $\mathbf{A}_\#$ of internal arcs, the set $\mathbf{A}_{\text{in}}$ of input arcs, and the set $\mathbf{A}_{\text{out}}$ of output arcs:

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

The tail of any input arc is not attached to any node, and the head of an output arc is not attached to any node. Since there are no self-loops, *head*$(a) \neq$ *tail*(a) for all $a \in \mathbf{A}_\#$.

We assume that $\mathbf{N} \neq \varnothing$, *i.e.*, there is at least one node in $\mathbf{N}$, without which there would be no input arc, no output arc, and nothing to say. We do not assume $\mathcal{N}$ is connected as a directed graph – an assumption often made in studies of network flows, which is sensible when there is only one input arc (or "source node") and only one output arc (or "sink node").[4]

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

We call a bounded, closed interval $[r, r']$ of real numbers (possibly negative) a *type*, and we call a *typing* a partial map T (possibly total) that assigns types to subsets of the input and output arcs. Formally, T is of the following form, where $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \cup \mathbf{A}_{\text{out}}$:

$$T \;:\; \mathscr{P}(\mathbf{A}_{\text{in,out}}) \;\to\; \mathcal{I}(\mathbb{R})$$

where $\mathscr{P}(\;)$ is the power-set operator, $\mathscr{P}(\mathbf{A}_{\text{in,out}}) = \{A \mid A \subseteq \mathbf{A}_{\text{in,out}}\}$, and $\mathcal{I}(\mathbb{R})$ is the set of bounded, closed intervals of reals:

$$\mathcal{I}(\mathbb{R}) \;:=\; \Big\{ [r, r'] \;\Big|\; r, r' \in \mathbb{R} \text{ and } r \leqslant r' \Big\}.$$

As a function, T is not totally arbitrary and satisfies certain conditions, discussed in Section 3, which qualify it as a *network typing*. Henceforth, we use the term "network" to mean "flow network" in the sense just defined.

Flow Conservation, Capacity Constraints, Type Satisfaction. Though obvious and entirely standard, we precisely state fundamental concepts to fix our notation for the rest of the report, in Definitions 1, 2, 3, and 4.

Definition 1 (*Flow Conservation*)**.** If A is a subset of arcs in $\mathcal{N}$ and f a flow in $\mathcal{N}$, we write $\sum f(A)$ to denote the sum of the flows assigned to all the arcs in A:

$$\textstyle\sum f(A) \;:=\; \sum\{\, f(a) \mid a \in A \,\}.$$

[4] Presence of multiple sources and multiple sinks is not incidental, but crucial to the way we develop and use our typing theory.

By convention, $\sum \varnothing = 0$. If $A = \{a_1, \dots, a_p\}$ is the set of arcs entering a node ν, and $B = \{b_1, \dots, b_q\}$ the set of arcs exiting ν, conservation of flow at ν is expressed by the linear equation:

$$(1) \quad \sum f(A) \; = \; \sum f(B).$$

There is one such equation E_ν for every node $\nu \in \mathbf{N}$ and $\mathscr{E} = \{\, E_\nu \mid \nu \in \mathbf{N} \,\}$ is the collection of all equations enforcing flow conservation in $\mathcal{N}$. □

Note that we do not distinguish some nodes as "sources" and some other nodes as "sinks". The role of a "source" (resp. "sink") is assumed by an input arc (resp. output arc). Thus, flow conservation must be satisfied at all the nodes, with no distinction between them.

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

$$(2) \quad \underline{c}(a) \;\leqslant\; f(a) \;\leqslant\; \overline{c}(a).$$

There are two such inequalities C_a for every arc $a \in \mathbf{A}$ and $\mathscr{C} = \{\, C_a \mid a \in \mathbf{A} \,\}$ is the collection of all inequalities enforcing capacity constraints in $\mathcal{N}$. □

Definition 3 (*Feasible Flows*)**.** A flow f is *feasible* iff two conditions:

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

following standard definitions of network flows. □

Definition 4 (*Type Satisfaction*)**.** Let $\mathcal{N}$ be a network with input/output arcs $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$, and let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a typing over $\mathbf{A}_{\text{in,out}}$. We say the flow f *satisfies* T if, for every $A \in \mathscr{P}(\mathbf{A}_{\text{in,out}})$ for which $T(A)$ is defined with $T(A) = [r, r']$, it is the case that:

$$(3) \quad r \;\leqslant\; \sum f(A \cap \mathbf{A}_{\text{in}}) \;-\; \sum f(A \cap \mathbf{A}_{\text{out}}) \;\leqslant\; r'.$$

We often denote a typing T for $\mathcal{N}$ by simply writing $\mathcal{N} : T$. □

3 Network Typings

Let $\mathbf{A} = \mathbf{A}_{\#} \uplus \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$ be the set of arcs in a network $\mathcal{N}$, with $\mathbf{A}_{\text{in}} = \{a_1, \dots, a_k\}$ and $\mathbf{A}_{\text{out}} = \{a_{k+1}, \dots, a_{k+\ell}\}$, where $k, \ell \geqslant 1$. Throughout this section, we make no mention of the network $\mathcal{N}$ and it internal arcs $\mathbf{A}_{\#}$, and only deal with functions from $\mathscr{P}(\mathbf{A}_{\text{in,out}})$ to $\mathcal{I}(\mathbb{R})$ where, as in Section 2, we pose $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$.[5] We always call a map T, possibly partial, of the form:

$$T \;:\; \mathscr{P}(\mathbf{A}_{\text{in,out}}) \;\to\; \mathcal{I}(\mathbb{R})$$

a *typing* over $\mathbf{A}_{\text{in,out}}$. Such a typing T defines a convex *polyhedral set*, which we denote $\text{Poly}(T)$, in the Euclidean hyperspace $\mathbb{R}^{k+\ell}$, as we explain next. We think of the $k + \ell$ arcs in $\mathbf{A}_{\text{in,out}}$ as the dimensions of the space $\mathbb{R}^{k+\ell}$, and we use the arc names as variables to which we assign values in $\mathbb{R}$. $\text{Poly}(T)$ is the intersection of at most $2 \cdot (2^{k+\ell} - 1)$ halfspaces, because there are $(2^{k+\ell} - 1)$ non-empty subsets in $\mathscr{P}(\mathbf{A}_{\text{in,out}})$ and each induces two inequalities, as follows. Let $\varnothing \neq A \subseteq \mathbf{A}_{\text{in,out}}$ with:

$$A \cap \mathbf{A}_{\text{in}} = \{b_1, \dots, b_p\} \quad \text{and} \quad A \cap \mathbf{A}_{\text{out}} = \{b_{p+1}, \dots, b_{p+q}\},$$

[5] The notation "$\mathbf{A}_{\text{in,out}}$" is ambiguous, because it does not distinguish between input and output arcs. We use it nonetheless for succinctness. The context will always make clear which members of $\mathbf{A}_{\text{in,out}}$ are input arcs and which are output arcs.

where $1 \leqslant p + q \leqslant k + \ell$. Suppose $T(A)$ is defined and let $T(A) = [r, r']$. Corresponding to A, there are two linear inequalities in the variables $\{b_1, \ldots, b_{p+q}\}$, denoted $T^{\min}_{\geqslant}(A)$ and $T^{\max}_{\leqslant}(A)$:

$$
\begin{array}{llll}
(4) & T^{\min}_{\geqslant}(A): & b_1 + \cdots + b_p - b_{p+1} - \cdots - b_{p+q} \geqslant r & \text{or, more succintly, } \sum(A \cap \mathbf{A}_{\text{in}}) - \sum(A \cap \mathbf{A}_{\text{out}}) \geqslant r \\
 & T^{\max}_{\leqslant}(A): & b_1 + \cdots + b_p - b_{p+1} - \cdots - b_{p+q} \leqslant r' & \text{or, more succintly, } \sum(A \cap \mathbf{A}_{\text{in}}) - \sum(A \cap \mathbf{A}_{\text{out}}) \leqslant r'
\end{array}
$$

and, therefore, two halfspaces $\mathsf{Half}(T^{\min}_{\geqslant}(A))$ and $\mathsf{Half}(T^{\max}_{\leqslant}(A))$ in $\mathbb{R}^{k+\ell}$ defined by:

$$
\begin{array}{lll}
(5) & \mathsf{Half}(T^{\min}_{\geqslant}(A)) & := \{\, \boldsymbol{r} \in \mathbb{R}^{k+\ell} \mid \boldsymbol{r} \text{ satisfies } T^{\min}_{\geqslant}(A) \,\}, \\
 & \mathsf{Half}(T^{\max}_{\leqslant}(A)) & := \{\, \boldsymbol{r} \in \mathbb{R}^{k+\ell} \mid \boldsymbol{r} \text{ satisfies } T^{\max}_{\leqslant}(A) \,\}.
\end{array}
$$

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

$$
\boxed{\mathsf{Poly}(T) \;:=\; \bigcap \Big\{ \mathsf{Half}(T^{\min}_{\geqslant}(A)) \cap \mathsf{Half}(T^{\max}_{\leqslant}(A)) \;\Big|\; \varnothing \neq A \subseteq \mathbf{A}_{\text{in,out}} \text{ and } T(A) \text{ is defined} \Big\}}
$$

For later reference, we write $\mathsf{Constraints}(T)$ for the set of all inequalities/constraints that define $\mathsf{Poly}(T)$:

$$
\begin{array}{lll}
(6) & \mathsf{Constraints}(T) & := \{\, T^{\min}_{\geqslant}(A) \mid \varnothing \neq A \subseteq \mathbf{A}_{\text{in,out}} \text{ and } T(A) \text{ is defined} \,\} \\
 & & \quad \cup \; \{\, T^{\max}_{\leqslant}(A) \mid \varnothing \neq A \subseteq \mathbf{A}_{\text{in,out}} \text{ and } T(A) \text{ is defined} \,\}.
\end{array}
$$

We sometimes write $\mathsf{Poly}(\mathsf{Constraints}(T))$ instead of $\mathsf{Poly}(T)$ if we need to make explicit reference to the inequalities induced by T. A $(k + \ell)$-dimensional point $\boldsymbol{r} = \langle r_1, \ldots, r_{k+\ell} \rangle$ defines a function $f : \mathbf{A}_{\text{in,out}} \to \mathbb{R}$ with $f(a_1) = r_1, \ldots, f(a_{k+\ell}) = r_{k+\ell}$. By a slight abuse of notation, we can therefore write $f \in \mathsf{Poly}(T)$ to mean that $\boldsymbol{r} = \langle r_1, \ldots, r_{k+\ell} \rangle \in \mathsf{Poly}(T)$. If $A \subseteq \mathbf{A}_{\text{in,out}}$, we write $[\mathsf{Poly}(T)]_A$ for the projection of $\mathsf{Poly}(T)$ on the subset A of the $(k + \ell)$ arcs/coordinates:

$$
[\mathsf{Poly}(T)]_A \;:=\; \Big\{ [f]_A \;\Big|\; f \in \mathsf{Poly}(T) \Big\}
$$

where $[f]_A$ is the restriction of f to the subset A.

We can view a typing T as a syntactic expression, with its semantics $\mathsf{Poly}(T)$ being a polytope in Euclidean hyperspace. As in other situations connecting syntax and semantics, there are generally distinct typings T and T' such that $\mathsf{Poly}(T) = \mathsf{Poly}(T')$. This is an obvious consequence of the fact that the same polytope can be defined by many different equivalent sets of linear inequalities, which is the source of some complications when we combine two typings to produce a new one.

If T and U are typings over $\mathbf{A}_{\text{in,out}}$, we write $T \equiv U$ whenever $\mathsf{Poly}(T) = \mathsf{Poly}(U)$, in which case we say that T and U are *equivalent*.

Definition 5 (*Tight Typings*)**.** Let T be a typing over $\mathbf{A}_{\text{in,out}}$. T is *tight* if, for every $A \in \mathscr{P}(\mathbf{A}_{\text{in,out}})$ for which $T(A)$ is defined and for every $r \in T(A)$, there is an IO function $f \in \mathsf{Poly}(T)$ such that

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

Informally, T is tight if none of the intervals/types assigned by T to members of $\mathscr{P}(\mathbf{A}_{\text{in,out}})$ contains redundant information.[6] □

[6]There are different equivalent ways of defining "tightness". Let $\mathsf{Constraints}(T)$ be the set of inequalities induced by T, as in (6) above. Let $T^{\min}_{=}(A)$ and $T^{\max}_{=}(A)$ be the equations obtained by turning "$\geqslant$" and "$\leqslant$" into "=" in the inequalities $T^{\min}_{\geqslant}(A)$ and $T^{\max}_{\leqslant}(A)$ in $\mathsf{Constraints}(T)$. Using the terminology of [23], pp 327, we say $T^{\min}_{=}(A)$ is *active* for $\mathsf{Poly}(T)$ if $T^{\min}_{=}(A)$ defines a *face* of $\mathsf{Poly}(T)$, and similarly for $T^{\max}_{=}(A)$. We can then say that T is tight if, for every $T^{\min}_{\geqslant}(A)$ and every $T^{\max}_{\leqslant}(A)$, the corresponding $T^{\min}_{=}(A)$ and $T^{\max}_{=}(A)$ are active for $\mathsf{Poly}(T)$.

Let T be a typing over $\mathbf{A}_{\text{in,out}}$ and $A \subseteq \mathbf{A}_{\text{in,out}}$. If $T(A)$ is defined with $T(A) = [r_1, r_2]$ for some $r_1 \leqslant r_2$, we write $T^{\min}(A)$ and $T^{\max}(A)$ to denote the endpoints of $T(A)$:

$$T^{\min}(A) = r_1 \text{ and } T^{\max}(A) = r_2.$$

The following is sometimes an easier-to-use characterization of tight typings.

Proposition 6 (Tightness Defined Differently)**.** *Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a typing. T is tight iff, for every $A \subseteq \mathbf{A}_{\text{in,out}}$ for which $T(A)$ is defined, there are $f_1, f_2 \in \mathsf{Poly}(T)$ such that:*

$$\begin{aligned} T^{\min}(A) &= \sum f_1(A \cap \mathbf{A}_{\text{in}}) - \sum f_1(A \cap \mathbf{A}_{\text{out}}), \\ T^{\max}(A) &= \sum f_2(A \cap \mathbf{A}_{\text{in}}) - \sum f_2(A \cap \mathbf{A}_{\text{out}}). \end{aligned}$$

Proof. The left-to-right implication follows immediately from Definition 5. The right-to-left implication is a staightforward consequence of the linearity of the constraints that define T. □

Proposition 7 (Every Typing Is Equivalent to a Tight Typing)**.** *There is an algorithm* Tight *which, given a typing T as input, always terminates and returns an equivalent tight (and total) typing* $\mathsf{Tight}(T)$.

Proof. Starting from the given typing $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$, we first determine the set of linear inequalities $\mathsf{Constraints}(T)$ that defines $\mathsf{Poly}(T)$, as given in (6) above. We compute a total and tight typing $T' : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ by assigning an appropriate interval/type $T'(A)$ to every $A \in \mathscr{P}(\mathbf{A}_{\text{in,out}})$ as follows. For such a set A of input/output arcs, let $\theta(A)$ be the objective function:

$$\theta(A) := \sum A \cap \mathbf{A}_{\text{in}} - \sum A \cap \mathbf{A}_{\text{out}}.$$

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

Proposition 8 (Tightness Inherited Downward)**.** *Let $T, U : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be typings such that $T \subseteq U$, i.e., U extends T. If U is tight, then so is T tight.*

Proof. Two preliminary observations, both following from $T \subseteq U$:

1. For every $A \in \mathscr{P}(\mathbf{A}_{\text{in,out}})$, if $T(A)$ is defined, so is $U(A)$ defined with $T(A) = U(A)$.

2. $\mathsf{Poly}(T) \supseteq \mathsf{Poly}(U)$, because $\mathsf{Constraints}(T) \subseteq \mathsf{Constraints}(U)$.

We need to show that for every $A \in \mathscr{P}(\mathbf{A}_{\text{in,out}})$ for which $T(A)$ is defined and for every $r \in T(A)$, there is an IO function $f \in \mathsf{Poly}(T)$ such that the following equation holds:

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

If $T(A)$ is defined, then $U(A)$ is defined, and if $r \in T(A)$, then $r \in U(A)$, by observation 1. Because U is tight, there is $f \in \mathsf{Poly}(U)$ such that the preceding equation holds. But $f \in \mathsf{Poly}(U)$ implies $f \in \mathsf{Poly}(T)$, by observation 2, from which the desired conclusion follows. □

4 Valid Typings and Principal Typings

We relate typings, as defined in Section 3, to networks.

Definition 9 (*Input-Output Functions*)**.** Let $\mathbf{A} = \mathbf{A}_{\#} \uplus \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$ be the set of arcs in a network $\mathcal{N}$, with $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$ its set of input/output arcs. We call a function $f : \mathbf{A}_{\text{in,out}} \to \mathbb{R}_+$ an *input-output function*, or just *IO function*, for $\mathcal{N}$.

If $g : \mathbf{A} \to \mathbb{R}_+$ is a flow in $\mathcal{N}$, then $[g]_{\mathbf{A}_{\text{in,out}}}$, the restriction of g to $\mathbf{A}_{\text{in,out}}$, is an IO function. We say that an IO function $f : \mathbf{A}_{\text{in,out}} \to \mathbb{R}_+$ is *feasible* if there is a feasible flow $g : \mathbf{A} \to \mathbb{R}_+$ such that $f = [g]_{\mathbf{A}_{\text{in,out}}}$.

A typing $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ for $\mathcal{N}$ is defined independently of the internal arcs $\mathbf{A}_{\#}$. Hence, the notion of *satisfaction* of T by a flow g as in Definition 4 directly applies to an IO function f, with no change. More succinctly, the flow g satisfies T iff $[g]_{\mathbf{A}_{\text{in,out}}} \in \mathsf{Poly}(T)$ whereas the IO function f satisfies T iff $f \in \mathsf{Poly}(T)$. ☐

Let $\mathcal{N}$ and T be a network and a typing as in Definition 9. We say T is a *valid typing* for $\mathcal{N}$, sometimes denoted $(\mathcal{N} : T)$, if it is sound in the following sense:

(soundness) Every IO function $f : \mathbf{A}_{\text{in,out}} \to \mathbb{R}_+$ satisfying T can be extended to a feasible flow $g : \mathbf{A} \to \mathbb{R}_+$.

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

(completeness) Every feasible flow $g : \mathbf{A} \to \mathbb{R}_+$ satisfies T.

Any two principal typings T and U 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 10. *Let $(\mathcal{N} : T)$ and $(\mathcal{N} : T')$ be typings for the same $\mathcal{N}$. If T and T' are tight, total, and $T \equiv T'$, then $T = T'$.*

Proof. This follows from the construction in the proof of Proposition 7, where $\mathsf{Tight}(T)$ returns a typing which is both total and tight (and equivalent to T). ☐

Proposition 11 (Principal Typings Are Equivalent)**.** *If $(\mathcal{N} : T)$ and $(\mathcal{N} : U)$ are two principal typings for the same network $\mathcal{N}$, then $T \equiv U$. Moreover, if T and U are tight and total, then $T = U$.*

Proof. If both $(\mathcal{N} : T)$ and $(\mathcal{N} : U)$ are principal typing, then $\mathsf{Poly}(T) = \mathsf{Poly}(U)$, so that also $T \equiv U$. When T and U are tight and total, then the equality $T = U$ follows from Lemma 10. ☐

Based on the preceding, we can re-state the definition of principal typing as follows. Typing T is principal for network $\mathcal{N}$ if both:

$$\begin{aligned} \mathsf{Poly}(T) &\subseteq \{\, f : \mathbf{A}_{\text{in,out}} \to \mathbb{R}_+ \mid f \text{ feasible in } \mathcal{N} \,\} \quad \text{(soundness)},\\ \mathsf{Poly}(T) &\supseteq \{\, f : \mathbf{A}_{\text{in,out}} \to \mathbb{R}_+ \mid f \text{ feasible in } \mathcal{N} \,\} \quad \text{(completeness)}. \end{aligned}$$

Restriction 12. In the rest of this report, every typing $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ will be equivalent to a typing $T' : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$, including the possibility that $T = T'$, such that two requirements are satisfied:

1. $T'(\varnothing) = T'(\mathbf{A}_{\text{in,out}}) = [0, 0] = \{0\}$. Informally, $T'(\mathbf{A}_{\text{in,out}}) = \{0\}$ expresses *global flow conservation*: The total amount entering a network must equal the total amount exiting it.

2. T' is defined for every singleton subset $A \subseteq \mathbf{A}_{\text{in,out}}$. Moreover, there is a "very large" number K such that for every singleton $A \subseteq \mathbf{A}_{\text{in}}$ (resp. $A \subseteq \mathbf{A}_{\text{out}}$), it holds that $T'(A) \subseteq [0, K]$ (resp. $T'(A) \subseteq [-K, 0]$), *i.e.*, the value of a feasible flow on any outer arc is between 0 and K.

A consequence of the second requirement is that $\mathsf{Poly}(T)$ is inside the $(k+\ell)$-dimensional hypercube $[0,K]^{k+\ell}$, thus entirely contained in a bounded part of the first orthant of the hyperspace $\mathbb{R}^{k+\ell}$. $\mathsf{Poly}(T)$ is thus a *bounded* subset of $\mathbb{R}^{k+\ell}$, and therefore a convex *polytope*, rather than just a convex *polyhedral set*. □

We include a few facts about typings that we use in later sections. These are solely about typings and make no mention of a network $\mathcal{N}$ and its set $\mathbf{A}_{\#}$ of internal arcs.

If $[r,s]$ is an interval of real numbers for some $r \leqslant s$, we write $-[r,s]$ to denote the interval $[-s,-r]$. This implies the following:

$$-[r,s] \;=\; \{\, t \in \mathbb{R} \mid -s \leqslant t \leqslant -r \,\} \;=\; \{\, -t \in \mathbb{R} \mid t \in [r,s] \,\}$$

Recall that $\mathsf{Constraints}(T)$ denote the set of linear inequalities induced by a typing T, as in (6) in Section 3.

Proposition 13. *Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a tight typing such that $T(\varnothing) = T(\mathbf{A}_{\text{in,out}}) = [0,0]$.*
***Conclusion:** For every two-part partition $A \uplus B = \mathbf{A}_{\text{in,out}}$, if $T(A)$ and $T(B)$ are defined, then $T(A) = -T(B)$.*

Proof. One particular case in the conclusion is when $A = \varnothing$ and $B = \mathbf{A}_{\text{in,out}}$, so that trivially $A \uplus B = \mathbf{A}_{\text{in,out}}$, which also implies $T(A) = -T(B)$. Because $T(\mathbf{A}_{\text{in,out}}) = [0,0]$ and T is tight, we have that:

$$0 \;\leqslant\; \sum \{\, a \mid a \in \mathbf{A}_{\text{in}} \,\} \;-\; \sum \{\, a \mid a \in \mathbf{A}_{\text{out}} \,\} \;\leqslant 0$$

are among the inequalities in $\mathsf{Constraints}(T)$. Consider arbitrary $\varnothing \neq A, B \subsetneq \mathbf{A}_{\text{in,out}}$ such that $A \uplus B = \mathbf{A}_{\text{in,out}}$ and both $T(A)$ and $T(B)$ are defined. For every $f \in \mathsf{Poly}(T)$, we can therefore write the equation:

$$\sum f(A \cap \mathbf{A}_{\text{in}}) \;+\; \sum f(B \cap \mathbf{A}_{\text{in}}) \;-\; \sum f(A \cap \mathbf{A}_{\text{out}}) \;\;-\; \sum f(B \cap \mathbf{A}_{\text{out}}) \;=\; 0$$

or, equivalently:

$$(\ddagger) \qquad \sum f(A \cap \mathbf{A}_{\text{in}}) \;-\; \sum f(A \cap \mathbf{A}_{\text{out}}) = -\sum f(B \cap \mathbf{A}_{\text{in}}) \;+\; \sum f(B \cap \mathbf{A}_{\text{out}})$$

Hence, relative to $\mathsf{Constraints}(T)$, f maximizes (resp. minimizes) the left-hand side of equation $(\ddagger)$ iff f maximizes (resp. minimizes) the right-hand side of $(\ddagger)$. Negating the right-hand side of $(\ddagger)$, we also have:

f maximizes (resp. minimizes) $\sum f(A \cap \mathbf{A}_{\text{in}}) - \sum f(A \cap \mathbf{A}_{\text{out}})$ if and only if
f minimizes (resp. maximizes) $\sum f(B \cap \mathbf{A}_{\text{in}}) - \sum f(B \cap \mathbf{A}_{\text{out}})$.

Because T is tight, by Proposition 6, every point $f \in \mathsf{Poly}(T)$ which maximizes (resp. minimizes) the objective function:

$$\theta(A) \;:=\; \sum A \cap \mathbf{A}_{\text{in}} \;-\; \sum A \cap \mathbf{A}_{\text{out}}$$

must be such that:

$$T^{\max}(A) \;=\; \sum f(A \cap \mathbf{A}_{\text{in}}) \;-\; \sum f(A \cap \mathbf{A}_{\text{out}})$$

$$\Big(\text{resp. } T^{\min}(A) \;=\; \sum f(A \cap \mathbf{A}_{\text{in}}) \;-\; \sum f(A \cap \mathbf{A}_{\text{out}})\Big)$$

We can repeat the same reasoning for B. Hence, if $f \in \mathsf{Poly}(T)$ maximizes both sides of $(\ddagger)$, then:

$$\begin{aligned} T^{\max}(A) \;&=\; +\sum f(A \cap \mathbf{A}_{\text{in}}) \;-\; \sum f(A \cap \mathbf{A}_{\text{out}}) \\ &=\; -\sum f(B \cap \mathbf{A}_{\text{in}}) \;+\; \sum f(B \cap \mathbf{A}_{\text{out}}) \\ &=\; -T^{\min}(B) \end{aligned}$$

and, respectively, if $f \in \mathsf{Poly}(T)$ minimizes both sides of (‡), then:

$$\begin{aligned} T^{\min}(A) &= +\sum f(A \cap \mathbf{A}_{\text{in}}) \;-\; \sum f(A \cap \mathbf{A}_{\text{out}}) \\ &= -\sum f(B \cap \mathbf{A}_{\text{in}}) \;+\; \sum f(B \cap \mathbf{A}_{\text{out}}) \\ &= -\,T^{\max}(B) \end{aligned}$$

The preceding implies $T(A) = -T(B)$ and concludes the proof. □

Proposition 14. *Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a tight typing such that $T(\varnothing) = T(\mathbf{A}_{\text{in,out}}) = [0,0]$.*
***Conclusion**: For every two-part partition $A \uplus B = \mathbf{A}_{\text{in,out}}$, if $T(A)$ is defined and $T(B)$ is undefined, then:*

$$\min\ \theta(B) = -T^{\max}(A) \quad \textit{and} \quad \max\ \theta(B) = -T^{\min}(A),$$

where $\theta(B) := \sum(B \cap \mathbf{A}_{\text{in}}) - \sum(B \cap \mathbf{A}_{\text{out}})$ is minimized and maximized, respectively, w.r.t. $\mathsf{Constraints}(T)$.

Hence, if we extend the typing T to a typing T' that includes the type assignment $T'(B) := -T(A)$, then T' is a tight typing equivalent to T.

Proof. If $T(A) = [r,s]$, then $r = \min\theta(A)$ and $s = \max\theta(A)$ where $\theta(A) := \sum(A \cap \mathbf{A}_{\text{in}}) - \sum(A \cap \mathbf{A}_{\text{out}})$ is minimized/maximized w.r.t. $\mathsf{Constraints}(T)$. Consider the objective $\Theta := \theta(A) + \theta(B)$. Because $T(\mathbf{A}_{\text{in,out}}) = [0,0]$, we have $\min\Theta = 0 = \max\Theta$ where Θ is minimized/maximized w.r.t. $\mathsf{Constraints}(T)$. Think of Θ as defining a line through the origin of the $\big(\theta(A),\theta(B)\big)$-plane with slope -45^o with, say, $\theta(A)$ the horizontal coordinate and $\theta(B)$ the vertical coordinate. Hence, $\min\theta(B) = -\max\theta(A)$ and $\max\theta(B) = -\min\theta(A)$, which implies the desired conclusion. □

Definition 15 (*True Types*)**.** Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a typing over $\mathbf{A}_{\text{in,out}}$ and $\mathscr{C} = \mathsf{Constraints}(T)$ the set of linear inequalities induced by T. For an arbitrary $\varnothing \neq A \subseteq \mathbf{A}_{\text{in,out}}$, we define the *true type of A relative to T*, denoted $\mathsf{TrType}(A,T)$, as follows:

$$\begin{aligned} &\mathsf{TrType}(A,T) \;:=\; [r,s] \\ &\text{where } r \;:=\; \min\theta(A) \;\text{ w.r.t. } \mathscr{C}, \quad s \;:=\; \max\theta(A) \;\text{ w.r.t. } \mathscr{C}, \quad \text{and } \;\theta(A) \;:=\; \sum A \cap \mathbf{A}_{\text{in}} - \sum A \cap \mathbf{A}_{\text{out}}. \end{aligned}$$

By Propositions 6 and 7, the typing T is tight iff, for every $\varnothing \neq A \subseteq \mathbf{A}_{\text{in,out}}$ for which $T(A)$ is defined, we have $T(A) = \mathsf{TrType}(A,T)$. In words, a tight typing T only assigns true types, although some of these types may be unnecessary, because they can be omitted without affecting $\mathsf{Poly}(T)$.

We also pose $\mathsf{TrType}(\varnothing,T) := [0,0]$, so that $\mathsf{TrType}(\_\_,T)$ is a total function on $\mathscr{P}(\mathbf{A}_{\text{in,out}})$. □

All typings T in this report will be such that $\mathsf{TrType}(\mathbf{A}_{\text{in,out}},T) = [0,0]$, but note that this does not necessarily mean that $T(\mathbf{A}_{\text{in,out}})$ is defined or, in case T is not tight, that $T(\mathbf{A}_{\text{in,out}}) = [0,0]$.

Definition 16 (*Components of Typings*)**.** Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a typing over the arcs/coordinates $\mathbf{A}_{\text{in,out}}$ such that $\mathsf{TrType}(\mathbf{A}_{\text{in,out}},T) = [0,0]$. Let $\mathbf{A}^{(1)}_{\text{in,out}} \uplus \cdots \uplus \mathbf{A}^{(n)}_{\text{in,out}} = \mathbf{A}_{\text{in,out}}$ be the finest partition of $\mathbf{A}_{\text{in,out}}$, for some $n \geqslant 1$, satisfying the condition:

$$\mathsf{TrType}(\mathbf{A}^{(i)}_{\text{in,out}},T) = [0,0] \quad \text{for every } 1 \leqslant i \leqslant n.$$

We call the restrictions of T to $\mathbf{A}^{(1)}_{\text{in,out}},\ldots,\mathbf{A}^{(n)}_{\text{in,out}}$ the *components of T*. Specifically, the typing T_i defined by:

$$T_i \;:\; \mathscr{P}(\mathbf{A}^{(i)}_{\text{in,out}}) \;\to\; \mathcal{I}(\mathbb{R}), \quad \text{where } T_i := [T]_{\mathscr{P}(\mathbf{A}^{(i)}_{\text{in,out}})},$$

is a *component of T*, for every $1 \leqslant i \leqslant n$. We also call the set $\mathbf{A}^{(i)}_{\text{in,out}}$ of arcs/coordinates a *component of* $\mathbf{A}_{\text{in,out}}$ *relative to T*, for every $1 \leqslant i \leqslant n$. □

Proposition 17. *Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a typing as in Definition 16, and let $\mathbf{A}^{(1)}_{\text{in,out}}, \dots, \mathbf{A}^{(n)}_{\text{in,out}}$ be the n components of $\mathbf{A}_{\text{in,out}}$ relative to T.*

***Conclusion**: For every component $\mathbf{A}^{(i)}_{\text{in,out}}$, with $1 \leqslant i \leqslant n$, and every two-part partition $A \uplus B = \mathbf{A}^{(i)}_{\text{in,out}}$, we have* $\mathsf{TrType}(A, T) = -\mathsf{TrType}(B, T)$.

Proof. Straightforward consequence of Propositions 13 and 14. All details omitted. □

For later reference, we call a pair of non-empty subsets $A, B \in \mathscr{P}(\mathbf{A}^{(i)}_{\text{in,out}})$ such that $A \uplus B = \mathbf{A}^{(i)}_{\text{in,out}}$, as in the conclusion of Proposition 17, T*-companions*. The components $\mathbf{A}^{(1)}_{\text{in,out}}, \dots, \mathbf{A}^{(n)}_{\text{in,out}}$ relative to T, as well as $\varnothing$, do not have T-companions.

Later in this report, it will typically be the case that the components of $\mathbf{A}_{\text{in,out}}$ relative to a typing T contain each at least two arcs/coordinates. In such a case, every $\varnothing \neq A \subsetneq \mathbf{A}^{(i)}_{\text{in,out}}$, where $1 \leqslant i \leqslant n$, has a uniquely defined T-companion B.

5 A 'Whole-Network' Algorithm for Computing the Principal Typing

Let $\mathcal{N} = (\mathbf{N}, \mathbf{A})$ be a network. We follow the notation and conventions of Section 3 throughout. Let $\mathscr{E}$ be the collection of all equations enforcing flow conservation, and $\mathscr{C}$ the collection of all inequalities enforcing capacity constraints, in $\mathcal{N}$. Algorithm 1 gives the pseudocode of a procedure $\mathsf{WholePT}$ which computes a tight, total, and principal typing for $\mathcal{N}$, when $\mathcal{N}$ can be given at once in its entirety as a whole network. Theorem 18 asserts the correctness of $\mathsf{WholePT}$.

Algorithm 1 Calculate Tight, Total, and Principal Typing for Network $\mathcal{N}$ in Whole-Network Mode

```
    algorithm name: WholePT
    input: flow network $\mathcal{N} = (\mathbf{N}, \mathbf{A})$
    output: tight, total, and principal typing $T$ for $\mathcal{N}$

 1: $\mathscr{E}$ := {flow-conservation equations for $\mathcal{N}$}
 2: $\mathscr{C}$ := {capacity-constraint inequalities for $\mathcal{N}$}
 3: $T(\varnothing)$ := $\{0\}$
 4: for every $\varnothing \neq A \subseteq \mathbf{A}_{\text{in,out}}$ do
 5:     $\theta(A)$ := $\sum(A \cap \mathbf{A}_{\text{in}}) - \sum(A \cap \mathbf{A}_{\text{out}})$
 6:     $r_1$ := minimum of objective $\theta(A)$ relative to $\mathscr{E} \cup \mathscr{C}$
 7:     $r_2$ := maximum of objective $\theta(A)$ relative to $\mathscr{E} \cup \mathscr{C}$
 8:     $T(A)$ := $[r_1, r_2]$
 9: end for
10: return $T$
```

We write $\mathsf{WholePT}(\mathcal{N})$ for the result of applying $\mathsf{WholePT}$ to the network $\mathcal{N}$. Let $T = \mathsf{WholePT}(\mathcal{N})$. Let $f : \mathbf{A} \to \mathbb{R}_+$ be a feasible flow in network $\mathcal{N}$. It follows that f satisfies every equation and every inequality in $\mathscr{E} \cup \mathscr{C}$. Hence, for every $A \subseteq \mathbf{A}_{\text{in,out}}$, the value of $\sum f(A \cap \mathbf{A}_{\text{in}}) - \sum f(A \cap \mathbf{A}_{\text{out}})$ is in the interval $T(A)$, because this value must occur between the minimum and the maximum of the objective $\theta(A)$ relative to $\mathscr{E} \cup \mathscr{C}$.

It follows that every feasible IO function is a point in the polytope $\mathsf{Poly}(T)$. This proves one half of the "principality" of T in Theorem 18 below. The other half of the "principality" of T, namely, every point in $\mathsf{Poly}(T)$ is a feasible IO function, is a little more involved; its proof is in a companion report [19].

Theorem 18 (Inferring Tight, Total, and Principal Typings)**.** *The typing $T = \mathsf{WholePT}(\mathcal{N})$ is tight, total, and principal, for network $\mathcal{N}$.*

Complexity of $\mathsf{WholePT}$. The run-time complexity of $\mathsf{WholePT}$ depends on the linear-programming algorithm used to minimize and maximize the objective $\theta(A)$ in lines 6 and 7 in Algorithm 1, which can in principle be obtained in low-degree polynomial times as functions of $|\mathbf{A}_{\text{in,out}}|$. But the main cost is the result of assigning a type to every subset $A \subseteq \mathbf{A}_{\text{in,out}}$, for a total of $2^{|\mathbf{A}_{\text{in,out}}|}$ types. We do not analyze the complexity of $\mathsf{WholePT}$ further, because our later algorithms compute tight principal typings far more efficiently.

Example 19. We use procedure $\mathsf{WholePT}$ to infer a tight, total, and principal typing T for the network $\mathcal{N}$ shown in Figure 1. There are 8 equations in $\mathscr{E}$ enforcing flow conservation, one for each node in $\mathcal{N}$, and $2 \cdot 16 = 32$ inequalities in $\mathscr{C}$ enforcing lower-bound and upper-bound constraints, two for each arc in $\mathcal{N}$. We omit the straightforward $\mathscr{E}$ and $\mathscr{C}$.

By inspection, a minimum flow in $\mathcal{N}$ pushes 0 units through, and a maximum flow in $\mathcal{N}$ pushes 30 units. The value of all feasible flows in $\mathcal{N}$ is therefore in the interval $[0, 30]$.

The typing T returned by $\mathsf{WholePT}(\mathcal{N})$ in this example is as follows. In addition to $T(\varnothing) = [0, 0]$, it makes the following assignments:

$a_1 : [0, 15]$	$a_2 : [0, 25]$	$-a_3 : [-15, 0]$	$-a_4 : [-25, 0]$
$a_1 + a_2 : [0, 30]$	$a_1 - a_3 : [-10, 12]$	$a_1 - a_4 : [-23, 15]$	
$a_2 - a_3 : [-15, 23]$	$a_2 - a_4 : [-12, 10]$	$-a_3 - a_4 : [-30, 0]$	
$a_1 + a_2 - a_3 : [0, 25]$	$a_1 + a_2 - a_4 : [0, 15]$	$a_1 - a_3 - a_4 : [-25, 0]$	$a_2 - a_3 - a_4 : [-15, 0]$
$a_1 + a_2 - a_3 - a_4 : [0, 0]$			

If we are only interested in computing the value of a maximum feasible flow in $\mathcal{N}$, it suffices to compute the type $[0, 30]$ assigned to $\{a_1, a_2\}$ or, equivalently, the type $[-30, 0]$ assigned to $\{a_3, a_4\}$. Algorithm $\mathsf{WholePT}$ can be adjusted so that it returns only one of these two types, but then it does not provide enough information (in the form of a principal typing) if we want to "safely" include $\mathcal{N}$ in a larger assembly of networks. □

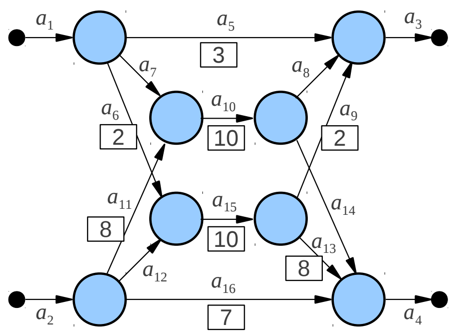

Figure 1: Network $\mathcal{N}$ in Example 19.
Omitted lower-bound capacities are 0, omitted upper-bound capacities are K (a "very large number").

6 Assembling Network Components

There are two basic ways in which we can assemble and connect networks together:

1. Let $\mathcal{M}$ and $\mathcal{N}$ be two networks, with arcs $\mathbf{A} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}} \uplus \mathbf{A}_{\#}$ and $\mathbf{B} = \mathbf{B}_{\text{in}} \uplus \mathbf{B}_{\text{out}} \uplus \mathbf{B}_{\#}$, respectively. The *parallel addition* of $\mathcal{M}$ and $\mathcal{N}$, denoted $(\mathcal{M} \| \mathcal{N})$, simply places $\mathcal{M}$ and $\mathcal{N}$ next to each other without connecting any of their outer arcs. The input and output arcs of $(\mathcal{M} \| \mathcal{N})$ are $\mathbf{A}_{\text{in}} \uplus \mathbf{B}_{\text{in}}$ and $\mathbf{A}_{\text{out}} \uplus \mathbf{B}_{\text{out}}$, respectively, and its internal arcs are $\mathbf{A}_{\#} \uplus \mathbf{B}_{\#}$.
2. Let $\mathcal{N}$ be a network with arcs $\mathbf{A} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}} \uplus \mathbf{A}_{\#}$, and let $a \in \mathbf{A}_{\text{in}}$ and $b \in \mathbf{A}_{\text{out}}$. The *binding of output arc b to input arc a in $\mathcal{N}$*, denoted $\mathsf{Bind}(\{a, b\}, \mathcal{N})$, means to connect $\textit{head}(b)$ to $\textit{tail}(a)$ and thus set $\textit{tail}(a) := \textit{head}(b)$. The input and output arcs of $\mathsf{Bind}(\{a, b\}, \mathcal{N})$ are $\mathbf{A}'_{\text{in}} = \mathbf{A}_{\text{in}} - \{a\}$ and $\mathbf{A}'_{\text{out}} = \mathbf{A}_{\text{out}} - \{b\}$, respectively, and its internal arcs are $\mathbf{A}'_{\#} = \mathbf{A}_{\#} \cup \{a\}$, thus keeping a as an internal arc and eliminating b altogether.

In the parallel-addition operation, there is no change in the capacity functions $\underline{\boldsymbol{c}}$ and $\overline{\boldsymbol{c}}$. In the binding operation also, there is no change in these functions, except at arc a:

$$\underline{\boldsymbol{c}}(a) := \max\{\underline{\boldsymbol{c}}(a), \underline{\boldsymbol{c}}(b)\} \quad \text{and} \quad \overline{\boldsymbol{c}}(a) := \min\{\overline{\boldsymbol{c}}(a), \overline{\boldsymbol{c}}(b)\}.$$

Repeatedly using *parallel addition* and *input-output pair binding*, we can assemble any network $\mathcal{N}$ with $n \geqslant 1$ nodes and simultaneously compute a tight principal typing for it: We start by breaking $\mathcal{N}$ into n separate one-node networks and then re-assemble them to recover the original topology of $\mathcal{N}$. Assembling the one-node components together, and simultaneoulsy computing a tight principal typing for $\mathcal{N}$ in bottom-up fashion, is carried out according to the algorithms in Sections 9 and 10.

Algorithm 2 Calculate Tight, Total, and Principal Typing for One-Node Network $\mathcal{N}$

```
algorithm name: OneNodePT
input: one-node network $\mathcal{N}$ with outer arcs $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$
       and lower-bound and upper-bound capacities $\underline{\boldsymbol{c}}, \overline{\boldsymbol{c}} : \mathbf{A}_{\text{in,out}} \to \mathbb{R}_+$
output: tight, total, and principal typing $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ for $\mathcal{N}$

1:  $T(\varnothing) := [0, 0]$
2:  $T(\mathbf{A}_{\text{in,out}}) := [0, 0]$
3:  for every two-part partition $A \uplus B = \mathbf{A}_{\text{in,out}}$ with $A \neq \varnothing \neq B$ do
4:      $A_{\text{in}} := A \cap \mathbf{A}_{\text{in}}$; $A_{\text{out}} := A \cap \mathbf{A}_{\text{out}}$
5:      $B_{\text{in}} := B \cap \mathbf{A}_{\text{in}}$; $B_{\text{out}} := B \cap \mathbf{A}_{\text{out}}$
6:      $r_1 := -\min\{\sum \overline{\boldsymbol{c}}(B_{\text{in}}) - \sum \underline{\boldsymbol{c}}(B_{\text{out}}),\ \sum \overline{\boldsymbol{c}}(A_{\text{out}}) - \sum \underline{\boldsymbol{c}}(A_{\text{in}})\}$
7:      $r_2 := +\min\{\sum \overline{\boldsymbol{c}}(A_{\text{in}}) - \sum \underline{\boldsymbol{c}}(A_{\text{out}}),\ \sum \overline{\boldsymbol{c}}(B_{\text{out}}) - \sum \underline{\boldsymbol{c}}(B_{\text{in}})\}$
8:      $T(A) := [r_1, r_2]$
9:  end for
10: return $T$
```

Proposition 20 (Tight, Total, and Principal Typings for One-Node Networks). *Let $\mathcal{N}$ be a network with one node, input arcs $\mathbf{A}_{\text{in}}$, output arcs $\mathbf{A}_{\text{out}}$, and lower-bound and upper-bound capacities $\underline{\boldsymbol{c}}, \overline{\boldsymbol{c}} : \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}} \to \mathbb{R}_+$.*
Conclusion: OneNodePT$(\mathcal{N})$ *is a tight, total, and principal typing for $\mathcal{N}$.*

Proof. Let $A \uplus B = \mathbf{A}_{\text{in,out}}$ be an arbitrary two-part partition of $\mathbf{A}_{\text{in,out}}$, with $A \neq \varnothing \neq B$. Let $A_{\text{in}} := A \cap \mathbf{A}_{\text{in}}$,

$A_{\text{out}} := A \cap \mathbf{A}_{\text{out}}$, $B_{\text{in}} := B \cap \mathbf{A}_{\text{in}}$, and $B_{\text{out}} := B \cap \mathbf{A}_{\text{out}}$. Define the non-negative numbers:

$$\begin{array}{llll} s_{\text{in}} & := \sum \underline{c}(A_{\text{in}}) & s'_{\text{in}} & := \sum \overline{c}(A_{\text{in}}) \\ s_{\text{out}} & := \sum \underline{c}(A_{\text{out}}) & s'_{\text{out}} & := \sum \overline{c}(A_{\text{out}}) \\ t_{\text{in}} & := \sum \underline{c}(B_{\text{in}}) & t'_{\text{in}} & := \sum \overline{c}(B_{\text{in}}) \\ t_{\text{out}} & := \sum \underline{c}(B_{\text{out}}) & t'_{\text{out}} & := \sum \overline{c}(B_{\text{out}}) \end{array}$$

Although tedious and long, one approach to complete the proof is to exhaustively consider all possible orderings of the 8 values just defined, using the standard ordering on real numbers. Cases that do not allow any feasible flow can be eliminated from consideration; for feasible flows to be possible, we can assume that:

$$s_{\text{in}} \leqslant s'_{\text{in}}, \quad s_{\text{out}} \leqslant s'_{\text{out}}, \quad t_{\text{in}} \leqslant t'_{\text{in}}, \quad t_{\text{out}} \leqslant t'_{\text{out}},$$

and also assume that:

$$s_{\text{in}} + t_{\text{in}} \leqslant s'_{\text{out}} + t'_{\text{out}}, \quad s_{\text{out}} + t_{\text{out}} \leqslant s'_{\text{in}} + t'_{\text{in}},$$

thus reducing the total number of cases to consider. We consider the intervals $[s_{\text{in}}, s'_{\text{in}}]$, $[s_{\text{out}}, s'_{\text{out}}]$, $[t_{\text{in}}, t'_{\text{in}}]$, and $[t_{\text{out}}, t'_{\text{out}}]$, and their relative positions, under the preceding assumptions. Define the objective $\theta(A)$:

$$\theta(A) := \sum A_{\text{in}} - \sum A_{\text{out}}.$$

By the material in Sections 3 and 5, if T is a tight principal typing for $\mathcal{N}$ with $T(A) = [r_1, r_2]$, then r_1 is the minimum possible feasible value of $\theta(A)$ and r_2 is the maximum possible feasible value of $\theta(A)$ relative to $\mathsf{Constraints}(T)$.

We omit the details of the just-outlined exhaustive proof by cases. Instead, we argue for the correctness of OneNodePT more informally. It is helpful to consider the particular case when all lower-bound capacities are zero, *i.e.*, the case when $s_{\text{in}} = s_{\text{out}} = t_{\text{in}} = t_{\text{out}} = 0$. In this case, it is easy to see that:

$$\begin{array}{lll} r_1 & = -\min\{\sum \overline{c}(B_{\text{in}}), \sum \overline{c}(A_{\text{out}})\} & \text{maximum amount entering at } B_{\text{in}} \text{ and exiting at } A_{\text{out}}, \\ & & \text{while minimizing amount entering at } A_{\text{in}}, \\ r_2 & = +\min\{\sum \overline{c}(A_{\text{in}}), \sum \overline{c}(B_{\text{out}})\} & \text{maximum amount entering at } A_{\text{in}} \text{ and exiting at } B_{\text{out}}, \\ & & \text{while minimizing amount exiting at } A_{\text{out}}, \end{array}$$

which are exactly the endpoints of the type $T(A)$ returned by OneNodePT$(\mathcal{N})$ in the particular case when all lower-bounds are zero.

Consider now the case when some of the lower-bounds are not zero. To determine the maximum throughput r_2 using the arcs of A, we consider two quantities:

$$r'_2 := \sum \overline{c}(A_{\text{in}}) - \sum \underline{c}(A_{\text{out}}) \quad \text{and} \quad r''_2 := \sum \overline{c}(B_{\text{out}}) - \sum \underline{c}(B_{\text{in}}).$$

It is easy to see that r'_2 is the flow that is simultaneously maximized at A_{in} and minimized at A_{out}, provided that $r'_2 \leqslant r''_2$, *i.e.*, the whole amount r'_2 can be made to enter at A_{in} and to exit at B_{out}. However, if $r'_2 > r''_2$, then only the amount r''_2 can be made to enter at A_{in} and to exit at B_{out}. Hence, the desired value of r_2 is $\min\{r'_2, r''_2\}$, which is exactly the higher endpoint of the type $T(A)$ returned by OneNodePT$(\mathcal{N})$. A similar argument, here omitted, is used again to determine the minimum throughput r_1 using the arcs of A. □

Complexity of OneNodePT. We estimate the run-time of OneNodePT as a function of: $d = |\mathbf{A}_{\text{in,out}}| \geqslant 2$, the number of outer arcs, also assuming that there is at least one input arc and one output arc in $\mathcal{N}$. OneNodePT assigns the type/interval $[0,0]$ to $\varnothing$ and $\mathbf{A}_{\text{in,out}}$. For every $\varnothing \neq A \subsetneq \mathbf{A}_{\text{in,out}}$, it then computes a type $[r_1, r_2]$, simultaneously for A and its complement $B = \mathbf{A}_{\text{in,out}} - A$. (That B is assigned $[-r_2, -r_1]$ is not explicitly shown in Algorithm 2.) Hence, OneNodePT computes $(2^d - 2)/2 = (2^{d-1} - 1)$ such types/intervals, each involving 8 summations and 4 subtractions, in lines 6 and 7, on the lower-bound capacities (d of them) and upper-bound capacities (d of them) of the outer arcs.

Remark 21. A different version of Algorithm 2 uses linear programming to compute the typing of a one-node network, but this is an unnecessary overkill. The resulting run-time complexity is also worse than that of our version here. The linear-programming version works as follows. Let $\mathscr{E}$ be the set of flow-conservation equations and $\mathscr{C}$ the set of capacity-constraint inequalities of the one-node network. For every $A \in \mathscr{P}(\mathbf{A}_{\text{in,out}})$, we define the objective $\theta(A) := \sum(A \cap \mathbf{A}_{\text{in}}) - \sum(A \cap \mathbf{A}_{\text{out}})$. The desired type $T(A) = [r_1, r_2]$ is obtained by setting: $r_1 := \min \theta(A)$ and $r_2 := \max \theta(A)$, *i.e.*, the objective $\theta(A)$ is minimized/maximized relative to $\mathscr{E} \cup \mathscr{C}$ using linear programming. □

7 Parallel Addition

Given tight principal typings T and U for networks $\mathcal{M}$ and $\mathcal{N}$, respectively, we need to compute a tight principal typing for the parallel addition $(\mathcal{M} \| \mathcal{N})$.

Lemma 22. *Let T and U be tight typings over disjoint sets, $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$ and $\mathbf{B}_{\text{in,out}} = \mathbf{B}_{\text{in}} \uplus \mathbf{B}_{\text{out}}$, respectively. The* partial addition $(T \oplus_{\text{p}} U)$ *of T and U is defined as follows. For every $A \subseteq \mathbf{A}_{\text{in,out}} \cup \mathbf{B}_{\text{in,out}}$:*

$$(T \oplus_{\text{p}} U)(A) := \begin{cases} [0,0] & \textit{if } A = \varnothing \textit{ or } A = \mathbf{A}_{\text{in,out}} \cup \mathbf{B}_{\text{in,out}}, \\ T(A) & \textit{if } A \subseteq \mathbf{A}_{\text{in,out}} \textit{ and } T(A) \textit{ is defined}, \\ U(A) & \textit{if } A \subseteq \mathbf{B}_{\text{in,out}} \textit{ and } U(A) \textit{ is defined}, \\ \textit{undefined} & \textit{otherwise}. \end{cases}$$

***Conclusion**: $(T \oplus_{\text{p}} U)$ is a tight typing over $\mathbf{A}_{\text{in,out}} \cup \mathbf{B}_{\text{in,out}}$.*

Proof. Straightforward from the definitions. All details omitted. □

Proposition 23 (Typing for Parallel Addition)**.** *Let $\mathcal{M}$ and $\mathcal{N}$ be networks with outer arcs $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$ and $\mathbf{B}_{\text{in,out}} = \mathbf{B}_{\text{in}} \uplus \mathbf{B}_{\text{out}}$, respectively. Let T and U be tight principal typings for $\mathcal{M}$ and $\mathcal{N}$, respectively.*
***Conclusion**: $(T \oplus_{\text{p}} U)$ is a tight principal typing for the network $(\mathcal{M} \| \mathcal{N})$.*

Proof. By Lemma 22, $(T \oplus_{\text{p}} U)$ is a tight typing. That $(T \oplus_{\text{p}} U)$ is also principal for $(\mathcal{M} \| \mathcal{N})$ is a straightforward consequence of the definitions. All details omitted. □

The typing $(T \oplus_{\text{p}} U)$ is partial even when T and U are total typings. In this report, when assembling intermediate networks together, we restrict attention to their total typings. In the next lemma, we define the *total addition* of total typings which is another total typing. If $[r_1, s_1]$ and $[r_2, s_1]$ are intervals of real numbers for some $r_1 \leqslant s_1$ and $r_2 \leqslant s_2$, we write $[r_1, s_1] + [r_2, s_2]$ to denote the interval $[r_1 + r_2, s_1 + s_2]$:

$$[r_1, s_1] + [r_2, s_2] := \{\, t \in \mathbb{R} \mid r_1 + r_2 \leqslant t \leqslant s_1 + s_2 \,\}$$

Lemma 24. *Let T and U be tight and total typings over disjoint sets of outer arcs,* $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$ *and* $\mathbf{B}_{\text{in,out}} = \mathbf{B}_{\text{in}} \uplus \mathbf{B}_{\text{out}}$*, respectively. We define the* total addition $(T \oplus_{\text{t}} U)$ *of the typings T and U as follows. For every* $A \subseteq \mathbf{A}_{\text{in,out}} \cup \mathbf{B}_{\text{in,out}}$*:*

$$(T \oplus_{\text{t}} U)(A) := \begin{cases} [0,0] & \text{if } A = \varnothing \text{ or } A = \mathbf{A}_{\text{in,out}} \cup \mathbf{B}_{\text{in,out}}, \\ T(A') + U(A'') & \text{if } A = A' \uplus A'' \text{ with } A' = A \cap \mathbf{A}_{\text{in,out}} \text{ and } A'' = A \cap \mathbf{B}_{\text{in,out}}. \end{cases}$$

Conclusion: *$(T \oplus_{\text{t}} U)$ is a tight and total typing over* $\mathbf{A}_{\text{in,out}} \cup \mathbf{B}_{\text{in,out}}$.

Proof. Straightforward from the definitions, similar to the proof of Lemma 22. All details omitted. □

Proposition 25 (Total Typing for Parallel Addition)**.** *Let $\mathcal{M}$ and $\mathcal{N}$ be networks with outer arcs* $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$ *and* $\mathbf{B}_{\text{in,out}} = \mathbf{B}_{\text{in}} \uplus \mathbf{B}_{\text{out}}$*, respectively. Let T and U be tight, total, and principal typings for $\mathcal{M}$ and $\mathcal{N}$, respectively.*

Conclusion: *$(T \oplus_{\text{t}} U)$ is a tight, total, and principal typing for the network* $(\mathcal{M} \| \mathcal{N})$.

Proof. Similar to the proof of Proposition 23. By Lemma 24, $(T \oplus_{\text{t}} U)$ is a tight and total typing. That $(T \oplus_{\text{t}} U)$ is also principal for $(\mathcal{M} \| \mathcal{N})$ is a straightforward consequence of the definitions. All details omitted. □

8 Binding Input-Output Pairs

Given a tight principal typing T for network $\mathcal{N}$ with arcs $\mathbf{A} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}} \uplus \mathbf{A}_{\#}$, together with $a \in \mathbf{A}_{\text{in}}$ and $b \in \mathbf{A}_{\text{out}}$, we need to compute a tight principal typing T' for the network $\textsf{\textbf{Bind}}(\{a,b\},\mathcal{N})$. A straightforward, but expensive, way of computing T' is Algorithm 3. We invoke this algorithm by writing $\textsf{BindOne}(\{a,b\},T)$.

Algorithm 3 Bind One Input-Output Pair

```
    algorithm name: BindOne
    input: typing $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$, not necessarily tight, $a \in \mathbf{A}_{\text{in}}$, $b \in \mathbf{A}_{\text{out}}$
    output: tight typing $T' : \mathscr{P}(\mathbf{A}'_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$
            where $\mathbf{A}'_{\text{in,out}} = \mathbf{A}_{\text{in,out}} - \{a,b\}$ and $\mathsf{Poly}(T') = [\mathsf{Poly}(\mathsf{Constraints}(T) \cup \{a=b\})]_{\mathbf{A}'_{\text{in,out}}}$

1: $T'(\varnothing) := \{0\}$
2: $T'(\mathbf{A}'_{\text{in,out}}) := \{0\}$
3: for every $\varnothing \neq A \subsetneq \mathbf{A}'_{\text{in,out}}$ do
4:     $\theta_A := \sum(A \cap (\mathbf{A}_{\text{in}} - \{a\})) - \sum(A \cap (\mathbf{A}_{\text{out}} - \{b\}))$
5:     $r_1 :=$ minimum of objective $\theta_A$ relative to $\mathsf{Constraints}(T) \cup \{a=b\}$
6:     $r_2 :=$ maximum of objective $\theta_A$ relative to $\mathsf{Constraints}(T) \cup \{a=b\}$
7:     $T'(A) := [r_1, r_2]$
8: end for
9: return $T'$
```

Proposition 26 (Typing for Binding One Input/Output Pair)**.** *Let* $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ *be a principal typing for network $\mathcal{N}$, with outer arcs* $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$*, and let $a \in \mathbf{A}_{\text{in}}$ and $b \in \mathbf{A}_{\text{out}}$.*

Conclusion: $\textsf{BindOne}(\{a,b\},T)$ *is a tight principal typing for* $\textsf{\textbf{Bind}}(\{a,b\},\mathcal{N})$.

Proof. If T is a principal typing for $\mathcal{N}$, then $\mathsf{Poly}(\mathsf{Constraints}(T))$ is the set of all feasible IO functions in $\mathcal{N}$, when every IO function is viewed as a point in the hyperspace of dimension $k + \ell$ and each dimension is identified with one of the arcs in $\mathbf{A}_{\text{in,out}}$. Hence, if $\mathscr{E}$ is the set of all flow-conservation equations in $\mathcal{N}$ and $\mathscr{C}$ the set of all capacity-constraint inequalities in $\mathcal{N}$, we must have:

$$\mathsf{Poly}(\mathsf{Constraints}(T)) \;=\; \big[\mathsf{Poly}(\mathscr{E} \cup \mathscr{C})\big]_{\mathbf{A}_{\text{in,out}}}.$$

Adding the constraint $\{a = b\}$ to both sides of the preceding equality, we obtain:

$$\mathsf{Poly}(\mathsf{Constraints}(T) \cup \{a = b\}) \;=\; \big[\mathsf{Poly}(\mathscr{E} \cup \mathscr{C} \cup \{a = b\})\big]_{\mathbf{A}_{\text{in,out}}}.$$

Hence, by the definition of the algorithm $\mathsf{BindOne}$, we also have:

$$\mathsf{Poly}(T') \;=\; \big[\mathsf{Poly}(\mathsf{Constraints}(T) \cup \{a = b\})\big]_{\mathbf{A}'_{\text{in,out}}},$$

which implies that $\mathsf{Poly}(T') = \big[\mathsf{Poly}(\mathscr{E} \cup \mathscr{C} \cup \{a = b\})\big]_{\mathbf{A}'_{\text{in,out}}}$ and that $T' = \mathsf{BindOne}(\{a, b\}, T)$ is a principal typing for the network $\mathbf{Bind}\big(\{a, b\}, \mathcal{N}\big)$. We omit the straightforward proof that T' is also tight. □

Complexity of BindOne. The run-time of $\mathsf{BindOne}$ is excessive, because it assigns a type to every subset $A \subseteq \mathbf{A}'_{\text{in,out}}$, which also makes the typing T' returned by $\mathsf{BindOne}$ a total typing, even if the initial typing T is not total. For every $A \subseteq \mathbf{A}'_{\text{in,out}}$, moreover, $\mathsf{BindOne}$ invokes a linear-programming algorithm twice, once to minimize and once to maximize the objective θ_A.

We do not analyze the complexity of $\mathsf{BindOne}$ any further, because it is an unnecessary overkill: We can compute a tight principal typing for $\mathbf{Bind}\big(\{a, b\}, \mathcal{N}\big)$ far more efficiently below.

Algorithm 4 is the efficient counterpart of the earlier Algorithm 3, which now requires restrictions on the input for correct execution that are unnecessary in the earlier.

Proposition 27 (Typing for Binding One Input/Output Pair – Efficiently)**.** *Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a tight principal typing for network $\mathcal{N}$, with outer arcs $\mathbf{A}_{\text{in,out}} = \mathbf{A}_{\text{in}} \uplus \mathbf{A}_{\text{out}}$, and let $a \in \mathbf{A}_{\text{in}}$ and $b \in \mathbf{A}_{\text{out}}$.*
Conclusion: *$\mathsf{BindOneEff}(\{a, b\}, T)$ is a tight principal typing for $\mathbf{Bind}\big(\{a, b\}, \mathcal{N}\big)$. Moreover, if T is total, then so is $\mathsf{BindOneEff}(\{a, b\}, T)$ total.*

Proof. Consider the intermediate typings T_1 and T_2 as defined in algorithm $\mathsf{BindOneEff}$:

$$T_1, T_2 : \mathscr{P}(\mathbf{A}'_{\text{in,out}}) \to \mathcal{I}(\mathbb{R}) \quad \text{where } \mathbf{A}'_{\text{in,out}} = \mathbf{A}_{\text{in,out}} - \{a, b\}.$$

The definitions of T_1 and T_2 can be repeated differently as follows. For every $A \subseteq \mathbf{A}_{\text{in,out}}$:

$$T_1(A) \;:=\; \begin{cases} T(A) & \text{if } A \subseteq \mathbf{A}'_{\text{in,out}} \text{ and } T(A) \text{ is defined,} \\ \text{undefined} & \text{if } A \nsubseteq \mathbf{A}'_{\text{in,out}} \text{ or } T(A) \text{ is undefined,} \end{cases}$$

$$T_2(A) \;:=\; \begin{cases} T_1(A) & \text{if } A \subsetneq \mathbf{A}'_{\text{in,out}} \text{ and } T_1(A) \text{ is defined,} \\ [0, 0] & \text{if } A = \mathbf{A}'_{\text{in,out}}, \\ \text{undefined} & \text{if } A \nsubseteq \mathbf{A}'_{\text{in,out}} \text{ or } T_1(A) \text{ is undefined.} \end{cases}$$

If T is tight, then so is T_1, by Proposition 8. The only difference between T_1 and T_2 is that the latter includes the new type assignment $T_2(\mathbf{A}'_{\text{in,out}}) = [0, 0]$, which is equivalent to the constraint:

$$\sum \mathbf{A}'_{\text{in}} - \sum \mathbf{A}'_{\text{out}} = 0, \qquad \text{where } \mathbf{A}'_{\text{in}} = \mathbf{A}_{\text{in}} - \{a\} \text{ and } \mathbf{A}'_{\text{out}} = \mathbf{A}_{\text{out}} - \{b\},$$

Algorithm 4 Bind One Input-Output Pair Efficiently

```
algorithm name: BindOneEff
input: tight typing $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R}),\ a \in \mathbf{A}_{\text{in}},\ b \in \mathbf{A}_{\text{out}}$
        where for every two-part partition $A \uplus B = \mathbf{A}'_{\text{in,out}} := \mathbf{A}_{\text{in,out}} - \{a, b\}$,
        either both $T(A)$ and $T(B)$ are defined or both $T(A)$ and $T(B)$ are undefined
output: tight typing $T' : \mathscr{P}(\mathbf{A}'_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$
        where $\mathsf{Poly}(T') = [\mathsf{Poly}(\mathsf{Constraints}(T) \cup \{a = b\})]_{\mathbf{A}'_{\text{in,out}}}$

Definition of intermediate typing $T_1 : \mathscr{P}(\mathbf{A}'_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$
1: $T_1 := [T]_{\mathscr{P}(\mathbf{A}'_{\text{in,out}})}$,
   i.e., every type assigned by $T$ to a set $A$ such that $A \cap \{a, b\} \neq \varnothing$ is omitted in $T_1$

Definition of intermediate typing $T_2 : \mathscr{P}(\mathbf{A}'_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$
2: $T_2 := T_1[\mathbf{A}'_{\text{in,out}} \mapsto [0, 0]]$
   i.e., the type assigned by $T_1$ to $\mathbf{A}'_{\text{in,out}}$, if any, is changed to the type $[0, 0]$ in $T_2$

Definition of final typing $T' : \mathscr{P}(\mathbf{A}'_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$
3: for every two-part partition $A \uplus B = \mathbf{A}'_{\text{in,out}}$ do
4:     if $T_2(A)$ is defined with $T_2(A) = [r_1, s_1]$ and $T_2(B)$ is defined with $-T_2(B) = [r_2, s_2]$ then
5:         $T'(A) := [\max\{r_1, r_2\}, \min\{s_1, s_2\}]$ ; $T'(B) := -T'(A)$
6:     else if both $T_2(A)$ and $T_2(B)$ are undefined then
7:         $T'(A)$ is undefined ; $T'(B)$ is undefined
8:     end if
9: end for
10: return $T'$
```

which, given Lemma 13 and the fact that $\sum \mathbf{A}_{\text{in}} - \sum \mathbf{A}_{\text{out}} = 0$, is in turn equivalent to the constraint $a = b$. This implies the following equalities:

$$\begin{aligned}\big[\mathsf{Poly}(\mathsf{Constraints}(T) \cup \{a = b\})\big]_{\mathbf{A}'_{\text{in,out}}} &= \mathsf{Poly}(\mathsf{Constraints}(T_1) \cup \{\mathbf{A}'_{\text{in}} = \mathbf{A}'_{\text{out}}\}) \\ &= \mathsf{Poly}(\mathsf{Constraints}(T_2)) \\ &= \mathsf{Poly}(T_2)\end{aligned}$$

Hence, if T is a principal typing for $\mathcal{N}$, then T_2 is a principal typing for $\textbf{Bind}(\{a, b\}, \mathcal{N})$. It remains to show that T' as defined in algorithm $\mathsf{BindOneEff}$ is the tight version of T_2.

We define an additional typing $T_3 : \mathscr{P}(\mathbf{A}'_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ as follows. For every $A \subseteq \mathbf{A}'_{\text{in,out}}$ for which $T_2(A)$ is defined, let the objective θ_A be $\sum(A \cap \mathbf{A}'_{\text{in}}) - \sum(A \cap \mathbf{A}'_{\text{out}})$ and let:

$$T_3(A) \;:=\; [r, s] \qquad \text{where } r = \min \theta_A \text{ and } s = \max \theta_A \text{ relative to } \mathsf{Constraints}(T_2).$$

T_3 is obtained from T_2 in an "expensive" process, because it uses a linear-programming algorithm to minimize/maximize the objectives θ_A. Clearly $\mathsf{Poly}(T_2) = \mathsf{Poly}(T_3)$. Moreover, T_3 is guaranteed to be tight by the definitions and results in Section 2 – we leave to the reader the straightforward details showing that T_3 is tight. In particular, for every $A \subseteq \mathbf{A}'_{\text{in,out}}$ for which $T_2(A)$ is defined, it holds that $T_3(A) \subseteq T_2(A)$. Hence, for every $A \uplus B = \mathbf{A}'_{\text{in,out}}$ for which $T_2(A)$ and $T_2(B)$ are both defined:

(7) $\quad T_3(A) \;\subseteq\; T_2(A) \cap -T_2(B),$

since $T_3(A) = -T_3(B)$ by Lemma 13. Keep in mind that:

(8) $\quad \mathsf{Poly}(T_3)$ is the largest polytope satisfying $\mathsf{Constraints}(T_2)$,

and every other polytope satisfying $\mathsf{Constraints}(T_2)$ is a subset of $\mathsf{Poly}(T_3)$. We define one more typing $T_4 : \mathscr{P}(\mathbf{A}'_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ by appropriately restricting T_2; namely, for every two-part partition $A \uplus B = \mathbf{A}'_{\text{in,out}}$:

$$T_4(A) \;:=\; \begin{cases} T_2(A) \cap -T_2(B) & \text{if both } T_2(A) \text{ and } T_2(B) \text{ are defined,} \\ \text{undefined} & \text{if both } T_2(A) \text{ and } T_2(B) \text{ are undefined.} \end{cases}$$

Hence, $\mathsf{Poly}(T_4)$ satisfies $\mathsf{Constraints}(T_2)$, so that also for every $A \subseteq \mathbf{A}'_{\text{in,out}}$ for which $T_4(A)$ is defined, we have $T_3(A) \supseteq T_4(A)$, by (8) above. Hence, for every $A \uplus B = \mathbf{A}'_{\text{in,out}}$ for which $T_4(A)$ and $T_4(B)$ are both defined, we have:

(9) $\quad T_3(A) \;\supseteq\; T_4(A) \;=\; -T_4(B) \;=\; T_2(A) \cap -T_2(B).$

Putting (7) and (9) together:

$$T_2(A) \cap -T_2(B) \;\subseteq\; T_3(A) \;\subseteq\; T_2(A) \cap -T_2(B),$$

which implies $T_3(A) = T_2(A) \cap -T_2(B) = T_4(A)$. Hence, also, for every $A \subseteq \mathbf{A}'_{\text{in,out}}$ for which $T_3(A)$ is defined, $T_3(A) = T_4(A)$. This implies $\mathsf{Poly}(T_3) = \mathsf{Poly}(T_4)$ and that T_4 is tight. T_4 is none other than T' in algorithm $\mathsf{BindOneEff}$, thus concluding the proof of the first part in the conclusion of the proposition. For the second part, it is readily checked that if T is a total typing, then so is T' (details omitted). □

Complexity of BindOneEff. We estimate the run-time of BindOneEff as a function of:

- $|\mathbf{A}_{\text{in,out}}|$, the number of outer arcs,
- $|T|$, the number of assigned types in the initial typing T.

We consider each of the three parts separately:

1. The first part, line 1, runs in $\mathcal{O}(|\mathbf{A}_{\text{in,out}}|\cdot|T|)$ time, according to the following reasoning. Suppose the types of T are organized as a list with $|T|$ entries, which we can scan from left to right. The algorithm removes every type assigned to a subset $A \subseteq \mathbf{A}_{\text{in,out}}$ intersecting $\{a,b\}$. There are $|T|$ types to be inspected, and the subset A to which $T(A)$ is assigned has to be checked that it does not contain a or b. The resulting intermediate typing T_1 is such that $|T_1| \leqslant |T|$.
2. The second part of BindOneEff, line 2, runs in $\mathcal{O}(|\mathbf{A}'_{\text{in,out}}| \cdot |T_1|)$ time. It inspects each of the $|T_1|$ types, looking for one assigned to $\mathbf{A}'_{\text{in,out}}$, each such inspection requiring $|\mathbf{A}'_{\text{in,out}}|$ comparison steps. If it finds such a type, it replaces it by $[0,0]$. If it does not find such a type, it adds the type assignment $\{\mathbf{A}'_{\text{in,out}} \mapsto [0,0]\}$. The resulting intermediate typing T_2 is such that $|T_2| = |T_1|$ or $|T_2| = 1 + |T_1|$.
3. The third part, from line 3 to line 9, runs in $\mathcal{O}(|\mathbf{A}'_{\text{in,out}}| \cdot |T_2|^2)$ time. For every type $T_2(A)$, it looks for a type $T_2(B)$ in at most $|T_2|$ scanning steps, such that $A \uplus B = \mathbf{A}'_{\text{in,out}}$ in at most $|\mathbf{A}'_{\text{in,out}}|$ comparison steps; if and when it finds a type $T_2(B)$, which is guaranteed to be defined, it carries out the operation in line 4.

Adding the estimated run-times in the three parts, the overall run-time of BindOneEff is $\mathcal{O}(|\mathbf{A}_{\text{in,out}}| \cdot |T|^2)$. Let $\delta = |\mathbf{A}_{\text{in,out}}| \geqslant 2$. In the particular case when T is a total typing which therefore assigns a type to each of the 2^δ subsets of $\mathbf{A}_{\text{in,out}}$, the overall run-time of BindOneEff is $\mathcal{O}(\delta \cdot 2^{2\delta})$.

Note there are no arithmetical steps (addition, multiplication, etc.) in the execution of BindOneEff; besides the bookkeeping involved in partitioning $\mathbf{A}'_{\text{in,out}}$ in two disjoint parts, BindOneEff uses only comparison of numbers in line 4.

9 A 'Compositional' Algorithm for Computing the Principal Typing

We call a non-empty finite set of networks a *network collection* and use the mnemonic $\mathbf{NC}$, possibly decorated, to denote it. Let $\mathbf{NC}$ be a network collection. We obtain another collection $\mathbf{NC}'$ from $\mathbf{NC}$ in one of two cases:

1. There is a member $\mathcal{M} \in \mathbf{NC}$, whose set of outer arcs is $\mathbf{A}_{\text{in,out}}$, and there is a pair of input/output arcs $\{a,b\} \subseteq \mathbf{A}_{\text{in,out}}$ such that:

   $$\mathbf{NC}' := (\mathbf{NC} - \{\mathcal{M}\}) \cup \{\textsf{Bind}(\{a,b\},\mathcal{M})\}.$$

   In words, $\mathbf{NC}'$ is obtained from $\mathbf{NC}$ by binding an input/output pair in the same member $\mathcal{M}$ of $\mathbf{NC}$. More succintly, we write for this first case:

   $$\mathbf{NC}' := \textsf{Bind}(\{a,b\},\mathcal{M},\mathbf{NC}).$$

2. There are two members $\mathcal{M},\mathcal{N} \in \mathbf{NC}$, whose sets of outer arcs are $\mathbf{A}_{\text{in,out}}$ and $\mathbf{B}_{\text{in,out}}$, respectively, and there is a pair of input/output arcs $\{a,b\}$ with $a \in \mathbf{A}_{\text{in,out}}$ and $b \in \mathbf{B}_{\text{in,out}}$ such that:

   $$\mathbf{NC}' := (\mathbf{NC} - \{\mathcal{M},\mathcal{N}\}) \cup \{\textsf{Bind}(\{a,b\},(\mathcal{M}\,\|\,\mathcal{N}))\}.$$

   In words, $\mathbf{NC}'$ is obtained from $\mathbf{NC}$ by binding an input/output pair between two different members $\mathcal{M}$ and $\mathcal{N}$ of $\mathbf{NC}$. More succintly, we write for this second case:

   $$\mathbf{NC}' := \textsf{Bind}(\{a,b\},\mathcal{M},\mathcal{N},\mathbf{NC}).$$

For brevity, we may also write $\mathbf{NC}' := \textsf{Bind}\big(\{a,b\},\mathbf{NC}\big)$, meaning that $\mathbf{NC}'$ is obtained according to one of the two cases above.

Let $\mathcal{N} = (\mathbf{N},\mathbf{A})$ a flow network with $n \geqslant 1$ nodes, $\mathbf{N} = \{\nu_1,\dots,\nu_n\}$. We "decompose" $\mathcal{N}$ into a collection of one-node networks, denoted $\textsf{BreakUp}(\mathcal{N})$, as follows:

$$\textsf{BreakUp}(\mathcal{N}) \;:=\; \{\mathcal{N}_1,\dots,\mathcal{N}_n\}$$

where, for every $1 \leqslant i \leqslant n$, network $\mathcal{N}_i$ is defined by:

$$\mathcal{N}_i \;:=\; (\{\nu_i\},\mathbf{A}_i) \quad \text{where } \mathbf{A}_i = \{\, a \in \mathbf{A} \mid \textit{head}(a) = \nu_i \text{ or } \textit{tail}(a) = \nu_i \,\}.$$

Every input arc $a \in \mathbf{A}_{\text{in}}$ in the original $\mathcal{N}$ remains an input arc in one of the one-node networks in $\textsf{BreakUp}(\mathcal{N})$; specifically, if $\textit{head}(a) = \nu_i$, for some $1 \leqslant i \leqslant n$, then a is an input arc of $\mathcal{N}_i$. Similarly every output arc $b \in \mathbf{A}_{\text{out}}$ in the original $\mathcal{N}$ remains an output arc in one of the one-node networks in $\textsf{BreakUp}(\mathcal{N})$.

Let $a \in \mathbf{A}_{\#}$ be an internal arc in the original $\mathcal{N}$, with $\textit{tail}(a) = \nu_i$ and $\textit{head}(a) = \nu_j$ where $i \neq j$. We distinguish between a as an output arc of $\mathcal{N}_i$, and a as an input arc of $\mathcal{N}_j$, by writing a^- for the former ("a^- is an output arc of $\mathcal{N}_i$") and a^+ for the latter ("a^+ is an input arc of $\mathcal{N}_j$").

Lemma 28 (Rebuilding a Network from its One-Node Components)**.** *Let $\mathcal{N} = (\mathbf{N},\mathbf{A})$ be a flow network, with $|\mathbf{N}| = n$ and $|\mathbf{A}_{\#}| = m$. Let $\mathcal{N}$ be connected, i.e., for every two distinct nodes $\nu,\nu' \in \mathbf{N}$, there is an undirected path between ν and ν'. If $\sigma = b_1 b_2 \cdots b_m$ is an ordering of all the internal arcs in $\mathbf{A}_{\#}$, then*

$$\{\mathcal{N}\} \;=\; \textsf{Bind}\Big(\{b_1^+,b_1^-\},\Big(\textsf{Bind}\Big(\{b_2^+,b_2^-\},\;\cdots\;\Big(\textsf{Bind}\Big(\{b_m^+,b_m^-\},\textsf{BreakUp}(\mathcal{N})\Big)\Big)\Big)\Big)\Big)$$

i.e., we can reconstruct $\mathcal{N}$ from its one-node components with m binding operations.

Proof. Straightforward from the definitions preceding the proposition. □

Algorithm 5 is invoked by its name $\textsf{CompPT}$ and uses: algorithm $\textsf{OneNodePT}$ in Section 6, the operation $\oplus_{\text{t}}$ in Section 7, and algorithm $\textsf{BindOneEff}$ in Section 8. $\textsf{CompPT}$ takes two arguments: a flow network $\mathcal{N}$ which is assumed to be connected, and an ordering $\sigma = b_1 b_2 \cdots b_m$ of $\mathcal{N}$'s internal arcs which we call a *binding schedule*.

Theorem 29 (Inferring Tight, Total, and Principal Typings)**.** *Let $\mathcal{N}$ be a flow network and $\sigma = b_1 b_2 \cdots b_m$ be an ordering of all the internal arcs of $\mathcal{N}$. Then the typing $T = \textsf{CompPT}(\mathcal{N},\sigma)$ is tight, total, and principal, for network $\mathcal{N}$.*

Proof. Consider the intermediate collection $\mathbf{NC}_k$ of networks, and the corresponding collection $\mathscr{T}_k$ of typings, in algorithm $\textsf{CompPT}$. It suffices to show that, for every $k = 0,1,2,\dots,m$, the set $\mathscr{T}_k$ contains only tight and total typings, which are moreover each principal for the corresponding network in the collection $\mathbf{NC}_k$. This is true for $k = 0$ by Proposition 20, and is true again for each $k \geqslant 1$ by Proposition 25 and Proposition 27. □

Let $\mathcal{N}$ be a network and $\mathbf{A}_{\text{in,out}}$ its set of outer arcs. We define the measure $\textsf{degree}(\mathcal{N}) := |\mathbf{A}_{\text{in,out}}|$, which is the number of outer arcs of $\mathcal{N}$. Let $\mathbf{NC}$ be a network collection. We define:

$$\textsf{maxDegree}(\mathbf{NC}) \;:=\; \max\{\,\textsf{degree}(\mathcal{N}) \mid \mathcal{N} \in \mathbf{NC}\,\}.$$

The sequence of network collections $\langle \mathbf{NC}_i \,|\, 0 \leqslant i \leqslant m\rangle := \langle \mathbf{NC}_0,\mathbf{NC}_1,\dots,\mathbf{NC}_m\rangle$ thus depends on the binding sequence σ. We indicate this fact by saying that $\langle \mathbf{NC}_i \,|\, 0 \leqslant i \leqslant m\rangle$ is *induced* by σ. We write $\langle \mathbf{NC}_i\rangle$ instead of $\langle \mathbf{NC}_i \,|\, 0 \leqslant i \leqslant m\rangle$ for brevity. If σ' is another binding sequence, it induces another sequence $\langle \mathbf{NC}'_i\rangle$ of network collections. Clearly, $\mathbf{NC}_0 = \mathbf{NC}'_0$ and $\mathbf{NC}_m = \mathbf{NC}'_m = \{\mathcal{N}\}$, while the intermediate network collections are generally different in the two sequences. We define

$$\textsf{maxDegree}(\langle \mathbf{NC}_i\rangle) := \max\big\{\,\textsf{maxDegree}(\mathbf{NC}) \,|\, \mathbf{NC} \in \langle \mathbf{NC}_i\rangle\,\big\}$$

Algorithm 5 Calculate Tight, Total, and Principal Typing for Network $\mathcal{N}$ in Compositional Mode

algorithm name: CompPT

input: $\mathcal{N} = (\mathbf{N}, \mathbf{A})$ and an ordering $\sigma = b_1 b_2 \cdots b_m$ of the internal arcs of $\mathcal{N}$,
where $\mathbf{N} = \{\nu_1, \nu_2, \ldots, \nu_n\}$, $\mathbf{A} = \mathbf{A}_{\text{in,out}} \uplus \mathbf{A}_{\#}$, and $\mathbf{A}_{\#} = \{b_1, b_2, \ldots, b_m\}$

output: tight, total, and principal typing T for $\mathcal{N}$

```
1:  $\mathbf{NC}_0 := \{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_n\}$
        where $\{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_n\} = \mathsf{BreakUp}(\mathcal{N})$
2:  $\mathscr{T}_0 := \{T_1, T_2, \ldots, T_n\}$
        where $\{T_1, T_2, \ldots, T_n\} = \{\mathsf{OneNodePT}(\mathcal{N}_1), \mathsf{OneNodePT}(\mathcal{N}_2), \ldots, \mathsf{OneNodePT}(\mathcal{N}_n)\}$
3:  for every $k = 1, 2, \ldots, m$ do
4:      if $b_k^+$ and $b_k^-$ are in the same component $\mathcal{M}_i \in \mathbf{NC}_{k-1}$ then
5:          $\mathbf{NC}_k := \mathsf{Bind}(\{b_k^+, b_k^-\}, \mathcal{M}_i, \mathbf{NC}_{k-1})$
6:          $\mathscr{T}_k := (\mathscr{T}_{k-1} - \{T_i\}) \cup \{\mathsf{BindOneEff}(\{b_k^+, b_k^-\}, T_i)\}$
7:      else if $b_k^+$ and $b_k^-$ are in distinct components $\mathcal{M}_i, \mathcal{M}_j \in \mathbf{NC}_{k-1}$ then
8:          $\mathbf{NC}_k := \mathsf{Bind}(\{b_k^+, b_k^-\}, \mathcal{M}_i, \mathcal{M}_j, \mathbf{NC}_{k-1})$
9:          $\mathscr{T}_k := (\mathscr{T}_{k-1} - \{T_i, T_j\}) \cup \{\mathsf{BindOneEff}(\{b_k^+, b_k^-\}, (T_i \oplus_{\mathrm{t}} T_j))\}$
10:     end if
11: end for
12: $T := T'$ where $\{T'\} = \mathscr{T}_m$
13: return $T$
```

and similarly for $\mathsf{maxDegree}(\langle \mathbf{NC}'_i \rangle)$. We say the binding sequence σ is *optimal* if for every other binding sequence σ' :

$$\mathsf{maxDegree}(\langle \mathbf{NC}_i \rangle) \leqslant \mathsf{maxDegree}(\langle \mathbf{NC}'_i \rangle)$$

where $\langle \mathbf{NC}_i \rangle$ and $\langle \mathbf{NC}'_i \rangle$ are induced by σ and σ', respectively. Let $\delta = \mathsf{maxDegree}(\langle \mathbf{NC}_i \rangle)$. We call δ the *index* of the binding schedule σ.

Complexity of CompPT. We estimate the run-time complexity of CompPT by the number of types it has to compute, *i.e.*, by the total number of assignments made by the typings in $\mathscr{T}_0, \mathscr{T}_1, \mathscr{T}_2, \ldots, \mathscr{T}_m$.

We ignore the effort to define the initial collection $\mathbf{NC}_0$ and then to update it to $\mathbf{NC}_1, \mathbf{NC}_2, \ldots, \mathbf{NC}_m$. In fact, beyond the initial $\mathbf{NC}_0$, which we use to define the initial set of typings $\mathscr{T}_0$, the remaining collections $\mathbf{NC}_1, \mathbf{NC}_2, \ldots, \mathbf{NC}_m$ play no role in the computation of the final typing T returned by CompPT; we included them in the algorithm for clarity and to make explicit the correspondence between $\mathscr{T}_k$ and $\mathbf{NC}_k$ (which is used in the proof of Theorem 29).

The run-time complexity of CompPT depends on $\mathcal{N}$ as well as on the *binding schedule*, *i.e.*, the ordering $\sigma = b_1 b_2 \cdots b_m$ of all the internal arcs of $\mathcal{N}$ which specifies the *order* in which CompPT must re-connect each of the m arcs that are initially disconnected. Let δ be the index of σ.

There are at most $n \cdot 2^{\delta}$ type assignments in $\mathscr{T}_0$ in line 2. In the **for**-loop from line 3 to line 11, CompPT calls BindOneEff once in each of m iterations, for a total of m calls. Each such call to BindOneEff runs in time $\mathcal{O}\big(\delta \cdot 2^{2\delta}\big)$ according to the analysis in Section 8. Hence, the run-time complexity of CompPT is $\mathcal{O}\big(n \cdot 2^{\delta} + m \cdot \delta \cdot 2^{2\delta}\big)$ or also $\mathcal{O}\big((m+n) \cdot \delta \cdot 2^{2\delta}\big)$.

Moreover, if the binding sequence σ is optimal, this upper-bound is generally the best for CompPT asymptotically. Of course, this upper-bound is dominated by the exponential $2^{2\delta}$. For certain graph topologies, the parameter δ can be kept "small", *i.e.*, much smaller than $(m+n)$, as we mention in Section 11. And if in addition $m = \mathcal{O}(n)$, then CompPT runs in time linear in n.

10 A 'Compositional' Algorithm for Computing the Maximum-Flow Value

Algorithm CompMaxFlow below calls CompPT of the preceding section as a subroutine. As a consequence, given a network $\mathcal{N}$ as input argument, CompMaxFlow runs *optimally* if the call $\mathsf{CompPT}(\mathcal{N}, \sigma)$ uses an *optimal* binding schedule σ, as defined at the end of Section 9.

We delay to a later report our examination of the possible alternatives for computing an optimal, or near-optimal, binding schedule. For our purposes here, we assume the existence of an algorithm BindingSch which, given a network $\mathcal{N}$ as input argument, returns an optimal binding schedule σ for $\mathcal{N}$.

Algorithm 6 Calculate Maximum-Flow Value for Network $\mathcal{N}$ in Compositional Mode

```
    algorithm name: CompMaxFlow
    input: $\mathcal{N} = (\mathbf{N}, \mathbf{A})$, with a single input arc $a_{\text{in}}$ and a single output arc $a_{\text{out}}$
    output: maximum-flow value for $\mathcal{N}$

1: $\sigma$ := BindingSch($\mathcal{N}$)
2: $T$ := CompPT($\mathcal{N}, \sigma$)
3: $[r, r']$ := $T(\{a_{\text{in}}\})$
4: return $r'$
```

Under the assumption that the network $\mathcal{N}$, given as input argument to CompMaxFlow, has a single input arc a_{in} and a single output arc a_{out}, the types assigned by $T = \text{CompMaxFlow}(\mathcal{N})$ are of the form:

$$a_{\text{in}} : [r, r'] \qquad - a_{\text{out}} : [-r', -r] \qquad a_{\text{in}} - a_{\text{out}} : [0, 0]$$

where r and r' are the values of a minimum feasible flow and a maximum feasible flow in $\mathcal{N}$, respectively. Together with the definition of tight principal typings in Sections 3 and 4, this immediately implies the correctness of algorithm CompMaxFlow.

Complexity of CompMaxFlow. The run-time complexity of CompMaxFlow is that of BindingSch($\mathcal{N}$) plus that of CompPT($\mathcal{N}$). In a follow-up report, we intend to examine different ways of efficiently implementing algorithm BindingSch that returns an optimal, or near-optimal, binding schedule σ.

11 Special Cases

We briefly discuss a graph topology for which we can choose a binding schedule σ that makes both CompPT in Section 9 and CompMaxFlow in Section 10 run efficiently. This is work in progress and only partially presented here. There are other such graph topologies, whose examination we intend to take up in a follow-up report. We simplify our discussion below by assuming that, in all graph topologies under consideration:

- there is no node of degree $\leqslant 2$,
- there is no node with only input arcs,
- there is no node with only output arcs.

Thus, the degree of every node is $\geqslant 3$ with its incident arcs including both inputs and outputs. We also assume:

- the graph is bi-connected, with exactly one input arc (or source) and exactly one output arc (or sink).

As far as the maximum-flow problem is concerned, there is no loss of generality from these various assumptions. To simplify our discussion even further, we make one further assumption, though not essential:

- all lower-bound capacities on all arcs are zero.

This last assumption is common in other studies on maximum flow and makes it easier to compare our approach with other approaches.

Outerplanar Networks The qualifier *outerplanar* is usually applied to undirected graphs. We adapt it to a network $\mathcal{N} = (\mathbf{N}, \mathbf{A})$ as follows. Let G be the underlying graph of $\mathcal{N}$ where all arcs are inserted without their directions; in particular, if both $\langle \nu, \nu' \rangle$ and $\langle \nu', \nu \rangle$ are directed arcs in $\mathcal{N}$, then G merges the two into the single undirected arc $\{\nu, \nu'\}$. We say the network $\mathcal{N}$ is outerplanar if its undirected underlying graph G is outerplanar.

The (undirected) graph G is outerplanar if it can be embedded in the plane in such a way that all the nodes are on the outermost face, *i.e.*, the outermost boundary of the graph. Put differently, the graph is fully dismantled by removing all the nodes (and arcs incident to them) that are on the outermost face. In an outerplanar graph, an arc other than an input/output arc is of three kinds:

(1) a *bridge*, if its deletion disconnects the graph,

(2) a *peripheral* arc, if it is not a bridge and it occurs on the outermost boundary,

(3) a *chord*, if it is not a bridge and it is not peripheral.

Under our assumption that the graph is bi-connected, there are no bridges and there are only peripheral arcs and chordal arcs. Although for a planar graph in general there may exist several non-isomorphic embeddings in the plane, for an outerplanar graph the embedding is unique [20], which in turn implies that the classification into *peripheral* and *chordal* arcs is uniquely determined.[7]

With the assumptions that every node in G has degree $\geqslant 3$, that G is bi-connected, and that the outermost boundary of G is the (unique) Hamiltonian cycle [20], every face of the outerplanar graph G is either *triangular* or *quadrilateral*. We can therefore view an outerplanar graph as a finite sequence of adjacent triangular faces and quadrilateral faces. There is a triangular face at each end of the sequence, with the undirected input arc (resp. output arc) incident to the apex of the triangular face on the left (resp. on the right).

We transform G into another undirected graph G' where every node is of degree $= 3$. Specifically, if node ν is incident to arcs $\{\nu,\nu_1\},\{\nu,\nu_2\},\ldots,\{\nu,\nu_d\}$ where $d \geqslant 4$, then we perform the following steps:

1. Split ν into $(d-2)$ nodes, denoted $\nu^{(1)},\nu^{(2)},\ldots,\nu^{(d-2)}$. If we identify $\nu^{(1)}$ with ν, this means that we add $(d-3)$ new nodes.
2. Insert $(d-3)$ new arcs $\{\nu^{(1)},\nu^{(2)}\},\{\nu^{(2)},\nu^{(3)}\},\ldots,\{\nu^{(d-3)},\nu^{(d-2)}\}$.
   We call these the *auxiliary arcs* of G', to distinguish them from the *original arcs* in G.
3. Rename the original arc $\{\nu,\nu_1\}$ as $\{\nu^{(1)},\nu_1\}$, rename the original arc $\{\nu,\nu_d\}$ as $\{\nu^{(d-2)},\nu_d\}$, and, for every $2 \leqslant i \leqslant d-1$, rename the original arc $\{\nu,\nu_i\}$ as $\{\nu^{(i-1)},\nu_i\}$.

Clearly, G is obtained from G' by deleting all auxiliary arcs and merging $\{\nu^{(1)},\nu^{(2)},\ldots,\nu^{(d-2)}\}$ into the single node ν. From the construction, G' is an outerplanar graph in the form of a sequence of adjacent quadrilateral faces, plus one triangular face at the left end and one triangular face at the right end. Put differently, G' is a grid of size $2 \times ((n'/2)-1)$ where n' is the number of nodes in G', *i.e.*, G' consists of two rows each with $((n'/2)-1)$ nodes with a triangular face at each end of the grid.

Lemma 30. *Let m be the total number of peripheral and chordal arcs in G and n the total number of nodes in G. Let m' be the total number of peripheral and chordal arcs in G' and n' the total number of nodes in G'.* ***Conclusion:*** *$m' \leqslant 3\cdot m + 2$ and $n' = (2/3)\cdot(m'+1)$.*

Proof. Let m_c be the number of chords in G and m'_c the number of chords in G'. By construction, $m_c = m'_c$. Let m_p be the number of peripheral arcs in G and m'_p the number of peripheral arcs in G'. In general, m_p is smaller than m'_p. But the following relationship is easily checked:

$$m'_p \;=\; 2\cdot m'_c + 2 \;=\; 2\cdot m_c + 2,$$

where the "2" added on the right of the equality accounts for the apex nodes of the two triangular faces at both end of G'. It follows that we have the following sequence of three equalities and one inequality:

$$m' \;=\; m'_c + m'_p \;=\; m_c + 2\cdot m_c + 2 \;=\; 3\cdot m_c + 2 \;\leqslant\; 3\cdot m + 2,$$

which proves the first part of the conclusion. The second part is calculated from the fact that, excluding the triangular faces at both ends of G', there is a total of $2\cdot(m'-2)/3$ nodes. Adding the two apex nodes at both ends, we get:

$$2 + 2\cdot(m'-2)/3 \;=\; \frac{6 + 2\cdot m' - 4}{3} \;=\; (2/3)\cdot(m'+1),$$

which is the desired conclusion. □

[7] We use the expressions "peripheral arc" and "chordal arc" here not to confuse them with the expressions "internal arc" and "outer arc" in earlier sections. The set of all peripheral arcs and all chordal arcs here is exactly the set of *internal arcs*.

From the undirected outerplanar graph G', we define a new network $\mathcal{N}' = (\mathbf{N}', \mathbf{A}')$, with flow capacities $\underline{\boldsymbol{c}}', \overline{\boldsymbol{c}}' : \mathbf{A}' \to \mathbb{R}_+$, as follows:

1. The set $\mathbf{N}'$ of nodes in $\mathcal{N}'$ is identical to the set of nodes in G'.
2. Every undirected *original* arc in G', and therefore in G, is given the direction it had in $\mathcal{N}$ and then included in $\mathbf{A}'$. In particular, if $\{\nu, \nu'\}$ in G and G' is the result of merging $\langle \nu, \nu' \rangle$ and $\langle \nu', \nu \rangle$ in $\mathcal{N}$, then both $\langle \nu, \nu' \rangle$ and $\langle \nu', \nu \rangle$ are included in $\mathbf{A}'$. Hence, $\mathbf{A} \subseteq \mathbf{A}'$.
3. Every undirected *auxiliary* arc of the form $\{\nu^{(i)}, \nu^{(i+1)}\}$ in G' is split into two directed auxiliary arcs in $\mathcal{N}'$, namely, $\langle \nu^{(i)}, \nu^{(i+1)} \rangle$ and $\langle \nu^{(i+1)}, \nu^{(i)} \rangle$.
4. The flow capacities of every original arc $a \in \mathbf{A}$ in $\mathcal{N}$ are preserved in $\mathcal{N}'$, *i.e.*, $\underline{\boldsymbol{c}}'(a) = \underline{\boldsymbol{c}}(a) = 0$ and $\overline{\boldsymbol{c}}'(a) = \overline{\boldsymbol{c}}(a)$.
5. The flow capacities of every auxiliary arc $a \in \mathbf{A}' - \mathbf{A}$ in $\mathcal{N}'$ are defined by setting $\underline{\boldsymbol{c}}'(a) = 0$ and $\overline{\boldsymbol{c}}'(a) = K$, where K is a "very large number".

Just as G is obtained from G' by contracting all undirected auxiliary arcs, so is $\mathcal{N}$ obtained from $\mathcal{N}'$ by contracting all directed auxiliary arcs.

Lemma 31. *Let $\mathcal{N}' = (\mathbf{N}', \mathbf{A}')$ be the network just defined from the outerplanar $\mathcal{N}$.*

1. *Every maximum flow $f : \mathbf{A} \to \mathbb{R}_+$ in $\mathcal{N}$ can be extended to a maximum flow $f' : \mathbf{A}' \to \mathbb{R}_+$ in $\mathcal{N}'$.*
2. *Every maximum flow $f' : \mathbf{A}' \to \mathbb{R}_+$ in $\mathcal{N}'$ restricted to $\mathbf{A}$ is a maximum flow $[f']_{\mathbf{A}} : \mathbf{A} \to \mathbb{R}_+$ in $\mathcal{N}$.*

Hence, to determine a maximum flow in $\mathcal{N}$ it suffices to determine a maximum flow in $\mathcal{N}'$.

Proof. This is a straightforward consequence of the construction of $\mathcal{N}'$ preceding the lemma, because all auxiliary arcs in $\mathcal{N}'$ put no restriction on the flow. We omit all the details. □

Lemma 32. *Let $\mathcal{N}'$ be the outerplanar network defined by the construction preceding Lemma 31. There is a binding schedule σ for $\mathcal{N}'$ with index $\delta = 11$.*

Proof. We view G' as a sequence of adjacent quadrilateral faces, from left to right, with a single triangular face at each end. Excluding the apex node ν_1 of the triangular face on the left, and the apex node $\nu_{n'}$ of the triangular face on the right, we list the vertical chords from left to right as:

$$\{\nu_2, \nu_3\},\ \{\nu_4, \nu_5\},\ \ldots\ ,\ \{\nu_{n'-2}, \nu_{n'-1}\}.$$

An undirected arc in G' corresponds to one or two directed arcs in $\mathcal{N}'$. Hence, because every node in G' has degree = 3, the degree of every node in $\mathcal{N}'$ is $\leqslant 6$. We define the desired binding schedule in such a way that $\mathcal{N}'$ is re-assembled in stages, from its one-node components, from left to right. Each stage comprises between 3 and 6 bindings.

More specifically, at the end of stage $k \geqslant 1$, we have re-bound all the arcs to the left of chord $\{\nu_k, \nu_{k+1}\}$ but none to the right of $\{\nu_k, \nu_{k+1}\}$ yet. Also, at the end of stage $k \geqslant 1$, we have re-bound $\langle \nu_k, \nu_{k+1} \rangle$ or $\langle \nu_{k+1}, \nu_k \rangle$ or both, according to which of these three cases occur in the original network $\mathcal{N}$. At stage $k+1$, we re-bind the one or two directed arcs corresponding to each of the undirected:

(1) peripheral arc $\{\nu_k, \nu_{k+2}\}$,

(2) peripheral arc $\{\nu_{k+1}, \nu_{k+3}\}$, and

(3) chordal arc $\{\nu_{k+2}, \nu_{k+3}\}$,

in this order. It is now straightforward to check that the number of input/output arcs in every of the intermediate networks, in the course of re-assembling $\mathcal{N}'$ from its one-node components, does not exceed 11. $\square$

Theorem 33. *Let $\mathcal{N} = (\mathbf{N}, \mathbf{A})$ be an outerplanar network with a single input arc and a single output arc. Let m be the total number of internal arcs in $\mathcal{N}$.* ***Conclusion:*** *We can compute the value of a maximum flow in $\mathcal{N}$ in time $\mathcal{O}(m)$.*

Proof. By Lemma 31, we can deal with $\mathcal{N}'$ instead of $\mathcal{N}$ in order to find a maximum-flow value in $\mathcal{N}$. By Lemma 30, the total number m' of internal arcs and the total number n' of nodes in $\mathcal{N}'$ are both $\mathcal{O}(m)$. By the complexity analysis at the end of Section 9, together with Lemma 32, we can find a maximum-flow value in $\mathcal{N}'$ in time $\mathcal{O}\big((m' + n') \cdot \delta \cdot 2^{2\delta}\big)$ and therefore in time $\mathcal{O}(m)$. $\square$

k-Outerplanar Networks A graph is *k-outerplanar* for some $k \geqslant 1$ if it can be embedded in the plane in such a way that it can be fully dismantled by k repetitions of the process of removing all the nodes (and arcs incident to them) that are on the outermost face. Put differently, if we remove all the nodes on the outermost face (and arcs incident to them), we obtain a $(k-1)$-outerplanar graph. An outerplanar graph is 1-outerplanar.

Every planar graph is k-outerplanar for some $k \geqslant 1$. We delay to a follow-up report an extension of the preceding analysis for outerplanar networks to k-outerplanar networks in general, also using recent efficient algorithms to determine the smallest outerplanarity index k [8, 16].

12 Future Work

In both Sections 10 and 11, we mentioned several issues in our approach that are under current examination. All of these are related – and limited in one way or another – to the use of *total* typings: Algorithm CompPT returns a *total* tight and principal typing, whereas tight and principal typings in general do not have to be total. The same limitation applies to algorithm CompMaxFlow, which calls CompPT.

Being forced to work with total typings, the run-time complexity of these algorithms is excessive, unless it is possible to couple their execution with binding schedules that have small indeces, as discussed in Sections 9, 10, and 11. It is very likely that, if these algorithms could be adapted to work with *partial* typings, then their run-time complexity would be reduced to feasible (or low-degree polynomial) bounds and their execution less dependent on the existence and calculation of small-index binding schedules.

Hence, beyond the issues mentioned in Sections 10 and 11 for further examination, and more fundamentally, we need to tackle the problem of how to calculate *partial* tight and principal typings. In the definition and proposition below, we use the notation and conventions of Sections 3 and 4.

Definition 34 (*Redundant and Non-Redundant Typings*)**.** Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a tight typing satisfying two conditions:[8]

1. For every $a \in \mathbf{A}_{\text{in,out}}$, the type assignment $T(\{a\})$ is defined.
2. $T(\mathbf{A}_{\text{in,out}}) = [0, 0]$.

For $A \subseteq \mathbf{A}_{\text{in,out}}$ with $2 \leqslant |A| < |\mathbf{A}_{\text{in,out}}|$, define the typing $U : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ from T by omitting the type which T assigns to A, *i.e.*, if $T(A) = [r, s]$, then:

$$U := T - \{A : [r, s]\}.$$

We say the type assignment $T(A)$ is *redundant* if $\mathsf{Poly}(U) = \mathsf{Poly}(T)$. Informally, omitting $T(A)$ from T does not affect $\mathsf{Poly}(T)$.

[8]These conditions are not essential. They are included to match the same conditions elsewhere in this report, and to simplify a little the statement of Proposition 35. Note that we restrict the definition of "redundant" and "non-redudndant" to *tight* types and typings.

We say the typing T is *redundant* if it makes one or more redundant type assignments. Keep in mind that we limit the definition to type assignments $T(A)$ where $\{a\} \subsetneq A \subsetneq \mathbf{A}_{\text{in,out}}$ for every $a \in \mathbf{A}_{\text{in,out}}$.

We say the typing T is *non-redundant* if T makes no redundant type assignments. Informally, T is *non-redundant* if no type assignment $T(A)$ with $2 \leqslant |A| < |\mathbf{A}_{\text{in,out}}|$ can be omitted from the definition of T, without changing $\mathsf{Poly}(T)$ to another polytope properly containing $\mathsf{Poly}(T)$. □

Proposition 35. *Let $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$ be a tight and non-redundant typing, as in Definition 34. Let $a \in \mathbf{A}_{\text{in}}$ and $b \in \mathbf{A}_{\text{out}}$. Let U be the tight and non-redundant typing equivalent to $T' = \mathsf{BindOne}(\{a,b\},T)$ where* BindOne *is defined in Section 8.* ***Conclusion****: The number of types assigned by U is not greater than the number of types assigned by T.*

Proof. Let $\mathscr{C} = \mathsf{Constraints}(T)$ and $\mathscr{C}' = \mathsf{Constraints}(T')$. Every type $T(A)$ contributes two constraints, $T^{\min}_{\geqslant}(A)$ and $T^{\max}_{\leqslant}(A)$, both defined in (4) in Section 3. Let $T(\{a\}) = [r_1, r_2]$ and $-T(\{b\}) = [s_1, s_2]$, which correspond to four constraints:

$$T^{\min}_{\geqslant}(\{a\}) = \{a \geqslant r_1\}, \quad T^{\max}_{\leqslant}(\{a\}) = \{a \leqslant r_2\}, \quad T^{\min}_{\geqslant}(\{b\}) = \{-b \geqslant s_1\}, \quad T^{\max}_{\leqslant}(\{b\}) = \{-b \leqslant s_2\}.$$

Let $t_1 = \max\{r_1, -s_2\}$ and $t_2 = \min\{r_2, -s_1\}$. Reviewing the definition of BindOne, the typing T' is tight and total, such that $\mathsf{Poly}(\mathscr{C}') = \mathsf{Poly}(T')$ is also defined by the set of constraints:

$$\mathscr{D} := \Big(\mathscr{C} - \{a \geqslant r_1,\ a \leqslant r_2,\ -b \geqslant s_1,\ -b \leqslant s_2\}\Big) \cup \{a = b\} \cup \{a \geqslant t_1,\ a \leqslant t_2\},$$

i.e., $\mathscr{D}$ is obtained from $\mathscr{C}$ by omitting 4 constraints and adding 4 constraints. We thus have $\mathsf{Poly}(\mathscr{D}) = \mathsf{Poly}(\mathscr{C}') = \mathsf{Poly}(T')$ and the number of constraints in $\mathscr{D}$ is the same as the number of constraints in $\mathscr{C}$. This implies that the tight and non-redundant typing U, which is equivalent to T', makes at most as many type assignments as T. □

The tight and non-redundant typing U in the statement of Proposition 35 can be obtained from the typing $T' = \mathsf{BindOne}(\{a,b\},T)$ by standard techniques of linear programming, but this is precisely what we want to avoid because of its prohibitive cost. Hence, left to future work is to discover an efficient approach for finding the new tight and non-redundant typing U after each binding step. Moreover, given a tight and non-redundant typing $T : \mathscr{P}(\mathbf{A}_{\text{in,out}}) \to \mathcal{I}(\mathbb{R})$, we need an efficient way to compute $\mathsf{TrType}(A,T)$ for any $A \subseteq \mathbf{A}_{\text{in,out}}$ in case $T(A)$ is not defined.

References

[1] R.K. Ahuja, T. L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, N.J., 1993.

[2] Takao Asano and Yasuhito Asano. Recent Developments in Maximum Flow Algorithms. *Journal of the Operations Research Society of Japan*, 43(1), March 2000.

[3] A. Bestavros and A. Kfoury. A Domain-Specific Language for Incremental and Modular Design of Large-Scale Verifiably-Safe Flow Networks. In *Proc. of IFIP Working Conference on Domain-Specific Languages (DSL 2011), EPTCS Volume 66*, pages 24–47, Sept 2011.

[4] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: Tool and Use Cases. In *IEEE Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Wash D.C., December 2009.

[5] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: The Formal Framework. In *13th ACM HSCC*, Stockholm, April 2010.

[6] Bala G. Chandran and Dorit S. Hochbaum. A Computational Study of the Pseudoflow and Push-Relabel Algorithms for the Maximum Flow Problem. *Oper. Res.*, 57(2):358–376, March/April 2009.

[7] E.A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[8] Yuval Emek. k-Outerplanar Graphs, Planar Duality, and Low Stretch Spanning Trees. In Amos Fiat and Peter Sanders, editors, *Proceedings of 17th Annual European Symposium on Algorithms, ESA 2009*, pages 203–214. LNCS 5757, Springer Verlag, September 2009.

[9] L.R. Ford and D.R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[10] Andrew V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT, 1985.

[11] Andrew V. Goldberg. Recent Developments in Maximum Flow Algorithms (Invited Lecture). In *SWAT '98: Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, pages 1–10. Springer Verlag, 1998.

[12] Andrew V. Goldberg. Two-Level Push-Relabel Algorithm for the Maximum Flow Problem. In *Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management*, AAIM '09, pages 212–225, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Andrew V. Goldberg and Satish Rao. Beyond the Flow Decomposition Barrier. *J. ACM*, 45(5):783–797, September 1998.

[14] Andrew V. Goldberg and Robert E. Tarjan. A New Approach to the Maximum Flow Problem. In *Proceedings of the 18th Annual ACM Symposium on the Theory of Computing*, pages 136–146, 1986.

[15] Dorit S. Hochbaum. The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem. *Oper. Res.*, 56(4):992–1009, July 2008.

[16] Frank Kammer. Determining the Smallest k Such That G Is k-Outerplanar. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Proceedings of 15th Annual European Symposium on Algorithms, ESA 2007*, pages 359–370. LNCS 4698, Springer Verlag, September 2007.

[17] A. Kfoury. A Domain-Specific Language for Incremental and Modular Design of Large-Scale Verifiably-Safe Flow Networks (Part 1). Technical Report BUCS-TR-2011-011, CS Dept, Boston Univ, May 2011.

[18] A. Kfoury. The Denotational, Operational, and Static Semantics of a Domain-Specific Language for the Design of Flow Networks. In *Proc. of SBLP 2011: Brazilian Symposium on Programming Languages*, Sept 2011.

[19] A. Kfoury. A Typing Theory for Flow Networks (Part I). Technical Report BUCS-TR-2012-018, CS Dept, Boston Univ, December 2012.

[20] Josef Leydold and Peter Stadler. Minimal Cycle Bases of Outerplanar Graphs. *Electronic Journal of Combinatorics*, 5(R16), 1998.

[21] G. Mazzoni, S. Pallottino, and M.G. Scutella. The Maximum Flow Problem: A Max-Preflow Approach. *European Journal of Operational Research*, 53:257–278, 1991.

[22] James B. Orlin. Max Flows in $\mathcal{O}(mn)$ Time, or Better. Pre-publication draft, October 2012.

[23] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization, New York, USA, 1986.

[24] N. Soule, A. Bestavros, A. Kfoury, and A. Lapets. Safe Compositional Equation-based Modeling of Constrained Flow Networks. In *Proc. of 4th Int'l Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, Zürich, September 2011.