

Safe Compositional Specification of Networking Systems*

A Compositional Analysis Approach

Likai Liu, Assaf J. Kfoury, Azer Bestavros, Yarom Gabay, Adam D. Bradley, and Ibrahim Matta
{liulk,kfoury,best,yarom,artdodge,matta}@cs.bu.edu
Department of Computer Science, Boston University

December 28, 2005[†]

Abstract

We present a type inference algorithm, in the style of *compositional analysis*, for the language TRAFFIC—a specification language for flow composition applications proposed in [2]—and prove that this algorithm is correct: the typings it infers are *principal typings*, and the typings agree with syntax-directed type checking on closed flow specifications. This algorithm is capable of verifying *partial* flow specifications, which is a significant improvement over syntax-directed type checking algorithm presented in [3]. We also show that this algorithm runs efficiently, i.e., in low-degree polynomial time.

1 Introduction

In our previous reports [1, 2], we established a framework in which type systems reflect simplified representations of relationships between various network flows that can be derived from complex compositional theories. We defined formal operational semantic and presented a type system for the language TRAFFIC in [3], and proved several properties of the language. Our work here cannot be fully understood without reference to our previous work.

We showed that TRAFFIC meets our flexibility expectation, that (untyped) flow composition can be carried out in any order. However, the syntax-directed type checking algorithm we presented before is only capable of checking *closed* flow specification. This means if we want to compose flows *and* check correctness of the composition, we have to submit to the ordering constraint imposed by the type checker.

A *compositional analysis* of a system with many parts means that the parts can be analyzed completely independently of each other and in any order. In this way, analyzing one part does not need to wait for any part of the analysis results of other parts. Also, when a part is updated, only the final composition steps need to be re-analyzed and unchanged parts do not need re-analysis. This is achieved by designing a type inference algorithm for a type system that has the *principal typing* property [4].

For our particular topic, the safe composition of network flows, this means the analysis of a flow is *compositional* if it can be obtained by analyzing its subflows completely independently and in any order. The approach that we take involves producing what we call *principal typings of partial flow specifications*. A typing X of a term is *principal* if every typing Y of that term is an instance obtained by applying a substitution to X . A typing will generally contain type variables, i.e., place-holders for types to be specified further, and a substitution will act on these type variables. A flow specification is *partial* if it contains free flow variables.

In this paper, we present a type inference algorithm for TRAFFIC that satisfies *principal typings* property. These technical concepts and many others will be clarified and expanded in due course in the rest of this report.

1.1 A Primer for The Language TRAFFIC

In this paper, we refer to those definitions and notations in [3], which we reproduce in Figure 1.1 and summarize below.

*This work was supported in part by NSF grants ITR ANI-0205294, ANI-0095988, ANI-9986397, and EIA-0202067.

[†]Updated on September 12, 2007.

Figure 1.1: Syntax of TRAFFIC and Its Types

$ \begin{array}{ll} x, y, z \in \text{FlowVar} & \text{flow variable} \\ A, B, C \in \text{LocalFlow} & \text{local flow} \\ \mathcal{A}, \mathcal{B}, \mathcal{C} \in \text{GlobalFlow} & ::= A \mid x \\ & \mid \mathcal{A}; \mathcal{B} \quad \text{sequential flow} \\ & \mid \mathcal{A} \parallel \mathcal{B} \quad \text{parallel flow} \\ & \mid \text{let } x = \mathcal{A} \text{ in } \mathcal{B} \quad \text{let-binding} \end{array} $	$ \begin{array}{ll} r \in \text{FwSocketType} & \\ s \in \text{BwSocketType} & \\ t \in \text{SocketType} & ::= r \mid s \\ \rho \in \text{FwType} & ::= r \mid (\rho_1 \cdot \rho_2) \\ \sigma \in \text{BwType} & ::= s \mid (\sigma_1 \cdot \sigma_2) \\ \tau \in \text{Type} & ::= \rho \mid \sigma \\ T \in \text{FlowType} & ::= \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \end{array} $
(a) Syntax of Language	(b) Syntax of Types

Syntax of the Language The language of TRAFFIC supports parallel and sequential organization of flows and the ability to abstract unknown flows to be later specified by means of a let-binding.

Syntax of Types The type of flows illustrates that flows have four connections: forward input, forward output, backward input, and backward output. Every connection is a binary-tree structure of socket types that form the basis of all types and type judgments. Fitness of socket types is based on subtyping assumptions.

Subtyping Assumptions For the purpose of type analysis, we are given Δ as a set of binary relations to characterize the properties we want to check for an application. Let

$$\Delta \subseteq (\text{FwSocketType} \times \text{FwSocketType}) \cup (\text{BwSocketType} \times \text{BwSocketType})$$

be a set of fixed, but arbitrary subtyping assumptions on socket types given by the user. We assume Δ to be a partial order relation.

2 Typing Judgment and Typing Rules

2.1 Introducing Type Variables

For the purposes of compositional analysis, we need to augment the syntax of types with type variables. Type variables are placeholders for unknown types that are initially assigned to flow variables during type inference. This allows us to defer the analysis of types until we have more information.

Specifically, we introduce an infinite supply FwTypeVar of *forward type variables*, and an infinite supply BwTypeVar of *backward type variables*, with α and β (possibly decorated) ranging over FwTypeVar and BwTypeVar respectively. The set of all type variables is $\text{TypeVar} = \text{FwTypeVar} \cup \text{BwTypeVar}$, and we let γ range over TypeVar .

To the syntax of types given in Figure 1.1, we augment the sorts with type variables as follows:

$$\begin{array}{ll}
 \alpha \in \text{FwTypeVar} & \\
 \beta \in \text{BwTypeVar} & \\
 \gamma \in \text{TypeVar} & \\
 \tilde{\rho} \in \text{FwType}^{\sim} & ::= r \mid \alpha \mid (\tilde{\rho}_1 \cdot \tilde{\rho}_2) \\
 \tilde{\sigma} \in \text{BwType}^{\sim} & ::= s \mid \beta \mid (\tilde{\sigma}_1 \cdot \tilde{\sigma}_2) \\
 \tilde{\tau} \in \text{Type}^{\sim} & ::= \tilde{\rho} \mid \tilde{\sigma} \\
 \tilde{T} \in \text{FlowType}^{\sim} & ::= \begin{bmatrix} \tilde{\rho}_1 & \tilde{\rho}_2 \\ \tilde{\sigma}_1 & \tilde{\sigma}_2 \end{bmatrix}
 \end{array}$$

We also need the notion of *subtyping constraints* (or just *constraints*) which are simply written as:

$$\tilde{\rho}_1 <: \tilde{\rho}_2 \quad \text{and} \quad \tilde{\sigma}_1 <: \tilde{\sigma}_2,$$

which may or may not be satisfied. As a special notation where we desire both $\tilde{\tau}_1 <: \tilde{\tau}_2$ and $\tilde{\tau}_2 <: \tilde{\tau}_1$, we may also write them together as $\tilde{\tau}_1 \doteq \tilde{\tau}_2$. The purpose of defining a collection of constraints as a set is to delay checking the subtyping relation of unknown entities until they are known.

Definition 2.1 (Constraints). Let $C \subseteq \text{Constraints}$ range over a set of subtyping constraints, where

$$\text{Constraints} = (\text{FwType}\tilde{} \times \text{FwType}\tilde{}) \cup (\text{BwType}\tilde{} \times \text{BwType}\tilde{})$$

The eventual objective is to check for the constraints' consistency with Δ , where the meaning of "consistent constraints" is made precise by the notion of type variable substitution.

Definition 2.2 (Type Variable Substitution). A substitution S is a total map $S : \text{TypeVar} \rightarrow \text{Type}\tilde{}$ that is sort-preserving, i.e.,

$$\{S(\alpha) \mid \alpha \in \text{FwTypeVar}\} \subseteq \text{FwType}\tilde{} \quad \text{and} \quad \{S(\beta) \mid \beta \in \text{BwTypeVar}\} \subseteq \text{BwType}\tilde{}.$$

A *closed* substitution maps type variables to closed types, i.e., $S : \text{TypeVar} \rightarrow \text{Type}$ instead of $S : \text{TypeVar} \rightarrow \text{Type}\tilde{}$. We lift a substitution S to $\text{Type}\tilde{}$, $\text{FlowType}\tilde{}$, and Constraints in the obvious way, i.e., to substitution type variable occurrences in these sorts accordingly.

Notation 2.3. Typically, a substitution S will be the identity on all but finitely many variables, say, $\{\gamma_1, \dots, \gamma_n\}$ for some $n \geq 0$, in which case we write S as a finite map.

$$S = \{\gamma_1 \mapsto \tilde{\tau}_1, \dots, \gamma_n \mapsto \tilde{\tau}_n\}$$

where $S(\gamma_i) = \tilde{\tau}_i$ for every $1 \leq i \leq n$, and $S(\tilde{\tau}) = \tilde{\tau}$ for all $\tilde{\tau} \notin \text{dom}(S)$.

Definition 2.4 (Consistent Constraints). We say that a set of constraints C is *consistent* to Δ , or simply *consistent*, exactly when there exists a closed substitution S such that $\Delta \vdash S(C)$.

2.2 Subtyping Judgments

We extend the derivation of "subtyping judgments" in [3] to allow type variables. With the addition of a constraint set C (possibly containing type variables) consistent with Δ , we write subtyping judgments as

$$\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$$

for some types $\tilde{\tau}_1$ and $\tilde{\tau}_2$. All the axioms and rules in [3] are adapted accordingly, with some new rules:

$$\begin{array}{c} \frac{\{t_1 <: t_2\} \subseteq \Delta}{\Delta, C \vdash t_1 <: t_2} \text{ (stype)} \quad \frac{t \in \text{SocketType}}{\Delta, C \vdash t <: t} \text{ (stype-refl)} \quad \frac{\Delta, C \vdash t_1 <: t_2 \quad \Delta, C \vdash t_2 <: t_3}{\Delta, C \vdash t_1 <: t_3} \text{ (stype-trans)} \\ \frac{\{\tilde{\tau}_1 <: \tilde{\tau}_2\} \subseteq C}{\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2} \text{ (cons)} \quad \frac{\gamma \in \text{TypeVar}}{\Delta, C \vdash \gamma <: \gamma} \text{ (tv-refl)} \quad \frac{\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2 \quad \Delta, C \vdash \tilde{\tau}_2 <: \tilde{\tau}_3}{\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_3} \text{ (cons-trans)} \end{array}$$

Obviously, an inconsistent C would allow us to prove false subtyping relations using the (cons) rule, so it is important that every time we construct a new set of constraints, we must algorithmically check its consistency. We leave the criteria that C is consistent for later.

Types with type variables are lifted in the obvious way:

$$\frac{\Delta, C \vdash \tilde{\rho}_1 <: \tilde{\rho}'_1 \quad \Delta, C \vdash \tilde{\rho}_2 <: \tilde{\rho}'_2}{\Delta, C \vdash (\tilde{\rho}_1 \cdot \tilde{\rho}_2) <: (\tilde{\rho}'_1 \cdot \tilde{\rho}'_2)} \text{ (fwtype}\tilde{}\text{-lift)} \quad \frac{\Delta, C \vdash \tilde{\sigma}_1 <: \tilde{\sigma}'_1 \quad \Delta, C \vdash \tilde{\sigma}_2 <: \tilde{\sigma}'_2}{\Delta, C \vdash (\tilde{\sigma}_1 \cdot \tilde{\sigma}_2) <: (\tilde{\sigma}'_1 \cdot \tilde{\sigma}'_2)} \text{ (bwtype}\tilde{}\text{-lift)}$$

However, it is no longer the case that $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}'_1$ and $\Delta, C \vdash \tilde{\tau}_2 <: \tilde{\tau}'_2$ are the only way to derive the judgment $\Delta, C \vdash (\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)$. For example, let $C_1 = \{(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)\}$ and $C_2 = \{\tilde{\tau}_1 <: \tilde{\tau}'_1, \tilde{\tau}_2 <: \tilde{\tau}'_2\}$. Consider the following derivations:

$$\frac{\{(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)\} \subseteq C_1}{\Delta, C_1 \vdash (\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)} \quad \frac{\{\tilde{\tau}_1 <: \tilde{\tau}'_1\} \subseteq C_2 \quad \{\tilde{\tau}_2 <: \tilde{\tau}'_2\} \subseteq C_2}{\Delta, C_2 \vdash \tilde{\tau}_1 <: \tilde{\tau}'_1 \quad \Delta, C_2 \vdash \tilde{\tau}_2 <: \tilde{\tau}'_2} \quad \frac{\Delta, C_2 \vdash \tilde{\tau}_1 <: \tilde{\tau}'_1 \quad \Delta, C_2 \vdash \tilde{\tau}_2 <: \tilde{\tau}'_2}{\Delta, C_2 \vdash (\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)}$$

Figure 2.1: Typing Rules

$$\begin{array}{c}
\frac{\tilde{\Gamma}(x) = \tilde{T}}{\tilde{\Gamma}, \Delta, C \vdash x : \tilde{T}} \text{ (var}\tilde{\text{)}} \quad \frac{\text{type}(A) = \tilde{T}}{\tilde{\Gamma}, \Delta, C \vdash A : \tilde{T}} \text{ (local}\tilde{\text{)}} \\
\frac{\tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_1 : \begin{bmatrix} \tilde{\rho}_1 & \tilde{\rho}_2 \\ \tilde{\sigma}_1 & \tilde{\sigma}_2 \end{bmatrix} \quad \tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_2 : \begin{bmatrix} \tilde{\rho}_3 & \tilde{\rho}_4 \\ \tilde{\sigma}_3 & \tilde{\sigma}_4 \end{bmatrix} \quad \Delta, C \vdash \tilde{\rho}_2 <: \tilde{\rho}_3 \quad \Delta, C \vdash \tilde{\sigma}_3 <: \tilde{\sigma}_2}{\tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_1; \mathcal{A}_2 : \begin{bmatrix} \tilde{\rho}_1 & \tilde{\rho}_4 \\ \tilde{\sigma}_1 & \tilde{\sigma}_4 \end{bmatrix}} \text{ (seq}\tilde{\text{)}} \\
\frac{\tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_1 : \tilde{T}_1 \quad \tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_2 : \tilde{T}_2}{\tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_1 \parallel \mathcal{A}_2 : \tilde{T}_1 \bullet \tilde{T}_2} \text{ (par}\tilde{\text{)}} \quad \frac{\tilde{\Gamma}, \Delta, C \vdash \mathcal{B}_1 : \tilde{T}_1 \quad \tilde{\Gamma} \cup \{x : \tilde{T}_1\}, \Delta, C \vdash \mathcal{B}_2 : \tilde{T}}{\tilde{\Gamma}, \Delta, C \vdash \text{let } x = \mathcal{B}_1 \text{ in } \mathcal{B}_2 : \tilde{T}} \text{ (let}\tilde{\text{)}}
\end{array}$$

In some cases, we want C_1 to justify for the constraints $\tilde{\tau}_1 <: \tilde{\tau}'_1$ and $\tilde{\tau}_2 <: \tilde{\tau}'_2$ as well. This gives rise to the following projection rules:

$$\begin{array}{c}
\frac{\Delta, C \vdash (\tilde{\rho}_1 \cdot \tilde{\rho}_2) <: (\tilde{\rho}'_1 \cdot \tilde{\rho}'_2)}{\Delta, C \vdash \tilde{\rho}_1 <: \tilde{\rho}'_1} \text{ (fwtyp}\tilde{\text{e}}\text{-proj}_1) \quad \frac{\Delta, C \vdash (\tilde{\rho}_1 \cdot \tilde{\rho}_2) <: (\tilde{\rho}'_1 \cdot \tilde{\rho}'_2)}{\Delta, C \vdash \tilde{\rho}_2 <: \tilde{\rho}'_2} \text{ (fwtyp}\tilde{\text{e}}\text{-proj}_2) \\
\frac{\Delta, C \vdash (\tilde{\sigma}_1 \cdot \tilde{\sigma}_2) <: (\tilde{\sigma}'_1 \cdot \tilde{\sigma}'_2)}{\Delta, C \vdash \tilde{\sigma}_1 <: \tilde{\sigma}'_1} \text{ (bwtyp}\tilde{\text{e}}\text{-proj}_1) \quad \frac{\Delta, C \vdash (\tilde{\sigma}_1 \cdot \tilde{\sigma}_2) <: (\tilde{\sigma}'_1 \cdot \tilde{\sigma}'_2)}{\Delta, C \vdash \tilde{\sigma}_2 <: \tilde{\sigma}'_2} \text{ (bwtyp}\tilde{\text{e}}\text{-proj}_2)
\end{array}$$

2.3 Typing Judgments

We also extend the notion of derivation of “typing judgments” to the case when types contain type variables. With the addition of a constraint set C that is consistent with Δ , we write typing judgments as

$$\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$$

We write $\tilde{\Gamma}$ instead of Γ , and \tilde{T} instead of T , to indicate that type variables may be present. Substitution on $\tilde{\Gamma}$ is lifted in the usual way. Typing rules in [3] are adapted accordingly, as shown in Figure 2.1.

2.4 Properties

Subtyping Judgments There are some interesting properties we can show with regard to substitution, consistency, and derivability of subtyping judgments.

Lemma 2.5 (Subsets of a Consistent Constraint Set are Consistent). *Given sets of constraints C and C' such that $C \subseteq C'$. If C' is consistent, then C is consistent.*

Proof. Suppose C' is consistent, so there exists a closed substitution S such that $\Delta \vdash S(C')$. Since $C \subseteq C'$, we have $S(C) \subseteq S(C')$, and it is also the case that $\Delta \vdash S(C)$, therefore C is consistent. \square

A set of constraints is inconsistent when it contains one or more constraints that contradict the assumptions in Δ . However, if a set of constraints are already consistent, then taking constraints away from it does not cause contradictions, so its subsets are indeed consistent.

Lemma 2.6 (Weakening of Constraints in Subtyping Judgment). *Given sets of constraints C and C' such that $C \subseteq C'$. If $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ is derivable, then $\Delta, C' \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ is also derivable.*

Proof. This is shown by induction on the possible ways to derive $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$.

- case of (cons), we have $\{\tilde{\tau}_1 <: \tilde{\tau}_2\} \subseteq C$ as the premise. Since $C \subseteq C'$, it is also the case that $\{\tilde{\tau}_1 <: \tilde{\tau}_2\} \subseteq C'$, so $\Delta, C' \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ is derivable.
- case of (tv-refl), where $\tilde{\tau}_1 = \tilde{\tau}_2 = \gamma$ for some $\gamma \in \text{TypeVar}$. Clearly $\Delta, C' \vdash \gamma <: \gamma$ is immediately derivable.

- all other cases can be shown by straightforward application of induction hypothesis on the premises. □

If a subtyping judgment can be derived using some minimum constraint set, then additional constraints are simply dummies. They play no role in deriving the subtyping judgment, but their presence does not prevent derivability. Note that derivability on subtyping judgment does not depend on consistency of the constraint set.

Lemma 2.7 (Equivalence of Even Constraints). $\Delta, C \vdash (\tilde{\tau}_1 \cdot \tilde{\tau}'_1) <: (\tilde{\tau}_2 \cdot \tilde{\tau}'_2)$ is derivable if and only if $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ and $\Delta, C \vdash \tilde{\tau}'_1 <: \tilde{\tau}'_2$ are both derivable.

Proof. Apply projection rules for (\Rightarrow), and lifting for (\Leftarrow). □

Lemma 2.7 is used in Section 3.1 for the algorithm that normalizes constraints. It is used chiefly to show Lemma 3.5 that breaking down constraints, as described in Definition 3.1, produces equivalent constraints.

Lemma 2.8 (Substituted Subtyping Judgment is Derivable). For every given type variable substitution S , if a judgment $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ is derivable, then the judgment $\Delta, S(C) \vdash S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)$ is also derivable.

Proof. We proceed to prove this by examine the possible ways to derive the judgment $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$:

- case of (cons). Suppose $\{\tilde{\tau}_1 <: \tilde{\tau}_2\} \subseteq C$. We have $\{S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)\} \subseteq S(C)$, so we can derive $\Delta, S(C) \vdash S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)$.
- case of (tv-refl), where $\tilde{\tau}_1 = \tilde{\tau}_2 = \gamma$ for some $\gamma \in \text{TypeVar}$. Clearly, $S(\gamma) = S(\gamma)$, so $\Delta, S(C) \vdash S(\gamma) <: S(\gamma)$ is derivable.
- all other cases can be shown by induction. □

Lemma 2.8 is used to show the General Solution property (Lemma 2.12) below.

Lemma 2.9 (Substitution Preserves Consistency). Given any set of constraints C and a substitution S , if $S(C)$ is consistent, then C is consistent.

Proof. Suppose $S(C)$ is consistent, then by Definition 2.4, there exists a closed substitution S' such that $\Delta \vdash S'(S(C))$. Let $S'' = S' \circ S$, then S'' is a closed substitution such that $\Delta \vdash S''(C)$. By the same definition, C is consistent. □

Typing Judgments We also present properties pertaining to typing judgments, namely, weakening of environment and weakening of constraints, which allow us to add dummy bindings to the environment and dummy constraints.

Lemma 2.10 (Weakening of Environment). Given two environments $\tilde{\Gamma}$ and $\tilde{\Gamma}'$ such that $\tilde{\Gamma} \subseteq \tilde{\Gamma}'$ (i.e., for all $x \in \text{dom}(\tilde{\Gamma})$, it is the case that $\tilde{\Gamma}(x) = \tilde{\Gamma}'(x)$ holds). If $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$ is derivable, then $\tilde{\Gamma}', \Delta, C \vdash \mathcal{A} : \tilde{T}$ is derivable.

Proof. Induction by cases. Suppose $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$ is derivable.

- case \mathcal{A} is x , we have $\tilde{\Gamma}(x) = \tilde{T}$ as the premise. Since $x \in \text{dom}(\tilde{\Gamma})$, we have $\tilde{\Gamma}'(x) = \tilde{\Gamma}(x) = \tilde{T}$, so $\tilde{\Gamma}', \Delta, C \vdash x : \tilde{T}$ is derivable.
- case \mathcal{A} is A , then $\tilde{\Gamma}', \Delta, C \vdash A : \tilde{T}$ is immediately derivable.
- case \mathcal{A} is $\mathcal{A}_1; \mathcal{A}_2$ or $\mathcal{A}_1 \parallel \mathcal{A}_2$, we have $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_1 : \tilde{T}_1$ and $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_2 : \tilde{T}_2$ as the premises. By induction hypothesis, both $\tilde{\Gamma}', \Delta, C \vdash \mathcal{A}_1 : \tilde{T}_1$ and $\tilde{\Gamma}', \Delta, C \vdash \mathcal{A}_2 : \tilde{T}_2$ are derivable, so $\tilde{\Gamma}', \Delta, C \vdash \mathcal{A} : \tilde{T}$ is derivable.
- case \mathcal{A} is **let** $x = \mathcal{B}_0$ **in** \mathcal{B} , we have (1) $\tilde{\Gamma}, \Delta, C \vdash \mathcal{B}_0 : \tilde{T}_0$ and (2) $\tilde{\Gamma} \cup \{x : \tilde{T}_0\}, \Delta, C \vdash \mathcal{B} : \tilde{T}$ as the premises. By induction hypothesis on (1), $\tilde{\Gamma}', \Delta, C \vdash \mathcal{B}_0 : \tilde{T}_0$ is derivable. For (2), note that $x \notin \text{dom}(\tilde{\Gamma})$. If $x \in \text{dom}(\tilde{\Gamma}')$, then $\tilde{\Gamma} \cup \{x : \tilde{T}_0\} = \tilde{\Gamma}' \cup \{x : \tilde{T}_0\}$ (since new binding overrides the old one), so $\tilde{\Gamma}' \cup \{x : \tilde{T}_0\}, \Delta, C \vdash \mathcal{B} : \tilde{T}$ is immediately derivable. Otherwise, $\tilde{\Gamma} \cup \{x : \tilde{T}_0\} \subseteq \tilde{\Gamma}' \cup \{x : \tilde{T}_0\}$, so using induction hypothesis on (2), $\tilde{\Gamma}' \cup \{x : \tilde{T}_0\} \vdash \mathcal{B} : \tilde{T}$ is derivable.

□

Note that for all $y \in \text{dom}(\tilde{\Gamma}') - \text{dom}(\tilde{\Gamma})$, y does not occur free in \mathcal{A} .

Lemma 2.11 (Weaking of Constraints in Typing Judgment). *Given two sets of constraints C and C' such that $C \subseteq C'$. If $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$ is derivable, then $\tilde{\Gamma}, \Delta, C' \vdash \mathcal{A} : \tilde{T}$ is also derivable.*

Proof. This is a result that can be shown by induction on the structure of typing rules, using Lemma 2.6 when a premise is a subtyping judgment. □

Typing as a General Solution The following property is part of the Principal Typing theorem, which says that a typing returned by the compositional analysis algorithm is a general solution in the sense that all other typings are an instance of it, subject to substitution.

Lemma 2.12 (General Solution for Type Judgment). *Given $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$, then for every substitution S —not necessarily closed—if $S(C)$ is consistent, then there is a typing derivation where the final judgment is $S(\tilde{\Gamma}), \Delta, S(C) \vdash \mathcal{A} : S(\tilde{T})$.*

Proof. Suppose $S(C)$ is consistent, then C is consistent by Lemma 2.9. For every typing derivation rule concluding $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$, we analyze its premises by possible forms:

- Premise is a judgment of the form $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$, then by Lemma 2.8, $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ implies $\Delta, S(C) \vdash S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)$ is derivable.
- Premise is of the form $\tilde{\Gamma}(x) = \tilde{T}$, then we have $(S(\tilde{\Gamma}))(x) = S(\tilde{\Gamma}(x))$ by definition of substitution, and $S(\tilde{\Gamma}(x)) = S(\tilde{T})$ by definition of the environment, so we have $(S(\tilde{\Gamma}))(x) = S(\tilde{T})$.
- Premise is of the form $\text{type}(A) = \tilde{T}$. We have $S(\tilde{T}) = \tilde{T}$, since \tilde{T} does not contain type variables, so $\text{type}(A) = S(\tilde{T})$.
- Premise is a typing judgment of the form $\tilde{\Gamma}', \Delta, C' \vdash \mathcal{A}' : \tilde{T}'$. By induction hypothesis, the final judgment $S(\tilde{\Gamma}'), \Delta, S(C') \vdash \mathcal{A}' : S(\tilde{T}')$ is derivable.

□

These properties presented in the section are used to prove the correctness of the compositional analysis algorithm in Section 3.3.

3 Algorithm

For the syntax-directed type checking in [3], a flow has a typing if it is determined to be compatible using the typing rules, but the entire flow must be known in advance; in compositional analysis, a flow may have unspecified holes, and it has a typing if no incompatibility is found among the specified parts. The difference is subtle. It is important to realize that when two typings are composed per flow composition, there may not be a valid resulting typing. The set of constraints allows us to defer the checking of subtyping relations until we have more information.

Deciding a solution for a set of constraints is known as *unification*. For our type inference algorithm, unification is done in two steps: normalizing a set of constraints to a manageable form and deciding consistency on the normalized constraints. Consistent constraints necessarily conform to subtyping assumptions in the sense that, there must exist a way to substitute all type variables in the set of constraints to concrete types, and the resulting subtyping relations are verifiable by the subtyping assumptions in Δ .

3.1 Normalization of Constraints

The need to decide the consistency of a constraint set, which could contain constraints that are complicated to solve, gives rise to the normalization algorithm below. The objective of the algorithm is to produce an equivalent constraint set in the normalized form, i.e., as a subset of NormConstraints, for which we can decide consistency by a straightforward reduction to solving a graph problem, which is both efficient and works around the problem that subtyping rules are not syntax directed.

Definition 3.1 (Normalized Constraints). A set of normalized constraints is for subtyping relations that contain no binary structured types on either the left or right side. It is defined as follows:

$$\begin{aligned} \text{NormConstraints} = & (\text{FwTypeVar} \times \text{FwSocketType}) \cup (\text{FwSocketType} \times \text{FwTypeVar}) \cup \\ & (\text{BwTypeVar} \times \text{BwSocketType}) \cup (\text{BwSocketType} \times \text{BwTypeVar}) \cup \\ & (\text{FwTypeVar} \times \text{FwTypeVar}) \cup (\text{FwSocketType} \times \text{FwSocketType}) \cup \\ & (\text{BwTypeVar} \times \text{BwTypeVar}) \cup (\text{BwSocketType} \times \text{BwSocketType}) \end{aligned}$$

Given $C \subseteq \text{Constraints}$, we will rewrite C to obtain an *equivalent* constraint set $C' \subseteq \text{NormConstraints}$ by the means of substitution and simplification. Normalized constraints are used to decide consistency in lieu of the given constraint set. The details of constraint set normalization follows.

Definition 3.2 (Simplification of Constraints). We define $\text{simplify}(C)$ to be a function that replaces all constraints in C of the form $(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)$ with two constraints, $\tilde{\tau}_1 <: \tilde{\tau}'_1$ and $\tilde{\tau}_2 <: \tilde{\tau}'_2$, leaving other constraints as is.

Algorithm 3.3 (Normalization of Constraints). We describe a normalization algorithm that normalizes a constraint set $C \subseteq \text{Constraints}$ to an equivalent constraint set $C' \subseteq \text{NormConstraints}$, based on a rewrite sequence of the form:

$$C_0 \xrightarrow{S_0} C_1 \xrightarrow{S_1} C_2 \xrightarrow{S_2} \dots$$

where we construct a substitution S_i and a new constraint set C_{i+1} from C_i , for $i \geq 0$, in the following manner: choose a constraint $\{\tilde{\tau} <: \tilde{\tau}'\} \subseteq C_i$ to be rewritten with S_i according to one of the rewrite rules below, and let $C_{i+1} = \text{simplify}(S_i(C_i))$. A rewrite step rewrites one constraint at a time, leaving other constraints intact. However, a rewrite step may produce at most two constraints due to simplification. Rewriting terminates when C_i contains no constraints for which the rewrite rules apply.

Rewrite Rules

1. If $\tilde{\tau} <: \tilde{\tau}'$ has the form $(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: \gamma'$, then let $S_i = \{\gamma' \mapsto (\gamma'_1 \cdot \gamma'_2)\}$ with γ'_1, γ'_2 fresh. Abort if γ' occurs in $\tilde{\tau}_1$ or $\tilde{\tau}_2$.
2. If $\tilde{\tau} <: \tilde{\tau}'$ has the form $\gamma <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)$, then let $S_i = \{\gamma \mapsto (\gamma_1 \cdot \gamma_2)\}$ with γ_1, γ_2 fresh. Abort if γ occurs in $\tilde{\tau}'_1$ or $\tilde{\tau}'_2$.
3. If $\tilde{\tau} <: \tilde{\tau}'$ has the form $(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)$, then S_i is the identity.

We know as a fact that rewrite sequence always either aborts or terminates. We write the sequence all the way to its termination in n steps as $C \xrightarrow[nf]{S}^* C_n$ and let $C' = C_n$, in which case C' is in the normalized form, and we say that constraint set C normalizes to C' .

The desired output of the algorithm upon termination is $\langle S, C' \rangle$, where $S = S_{n-1} \circ \dots \circ S_1 \circ S_0$ (and “ \circ ” is the usual function composition). The algorithm, when expressed as a function, is written as $\langle S, C' \rangle = \text{normalizeConstraints}(C)$.

Notation 3.4. We also write $\tilde{\tau} <: \tilde{\tau}' \Downarrow S$ to denote that, given a constraint $\tilde{\tau} <: \tilde{\tau}'$, a particular substitution S is chosen based on the rewrite rules described in Algorithm 3.3.

3.1.1 Correctness

Lemma 3.5 (Constraints Equivalence for Simplify). Let C be a set of constraints, and let $C' = \text{simplify}(C)$. Then $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ iff $\Delta, C' \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$.

Proof. We enumerate by the three basic rules to derive $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$.

- case $\{\tilde{\tau}_1 <: \tilde{\tau}_2\} \subseteq C$, $\tilde{\tau}_1 \neq \tilde{\tau}_2$. If $\tilde{\tau}_1 = (\tilde{\tau}_{1,1} \cdot \tilde{\tau}_{1,2})$ and $\tilde{\tau}_2 = (\tilde{\tau}_{2,1} \cdot \tilde{\tau}_{2,2})$, then

$$\begin{aligned} \Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2 & \text{ iff } \Delta, C \vdash \tilde{\tau}_{1,1} <: \tilde{\tau}_{2,1} \text{ and } \Delta, C \vdash \tilde{\tau}_{1,2} <: \tilde{\tau}_{2,2} \text{ (by Lemma ??)} \\ & \text{ iff } \Delta, C' \vdash \tilde{\tau}_{1,1} <: \tilde{\tau}_{2,1} \text{ and } \Delta, C' \vdash \tilde{\tau}_{1,2} <: \tilde{\tau}_{2,2} \\ & \quad \text{(since } \{\tilde{\tau}_{1,1} <: \tilde{\tau}_{2,1}, \tilde{\tau}_{1,2} <: \tilde{\tau}_{2,2}\} \subseteq C') \\ & \text{ iff } \Delta, C' \vdash \tilde{\tau}_1 <: \tilde{\tau}_2 \text{ (by Lemma ??)} \end{aligned}$$

Otherwise, $\text{simplify}(C)$ leaves $\{\tilde{\tau}_1 <: \tilde{\tau}_2\}$ intact, so the claim is immediate.

- case $\tilde{\tau}_1 = \tilde{\tau}_2$, then by reflexivity, both $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ and $\Delta, C' \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ are immediately derivable.
- case using transitivity. We proceed to show both ways:
 - (\Rightarrow) Suppose there exists some $\tilde{\tau}'$ such that $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}'$ and $\Delta, C \vdash \tilde{\tau}' <: \tilde{\tau}_2$, then we have $\Delta, C' \vdash \tilde{\tau}_1 <: \tilde{\tau}'$ and $\Delta, C' \vdash \tilde{\tau}' <: \tilde{\tau}_2$ by induction hypothesis, which implies $\Delta, C' \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ by transitivity.
 - (\Leftarrow) Suppose there exists some $\tilde{\tau}'$ such that $\Delta, C' \vdash \tilde{\tau}_1 <: \tilde{\tau}'$ and $\Delta, C' \vdash \tilde{\tau}' <: \tilde{\tau}_2$, then we have $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}'$ and $\Delta, C \vdash \tilde{\tau}' <: \tilde{\tau}_2$ by induction hypothesis, which implies $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ by transitivity.

□

Lemma 3.6 (Consistency Preservation for Simplify). *Let C be a set of constraints, then C is consistent iff $\text{simplify}(C)$ is consistent.*

Proof. When we say C is consistent, there exists a closed substitution S such that $\Delta \vdash S(C)$. In other words, for every $\{\tilde{\tau}_1 <: \tilde{\tau}_2\} \subseteq C$, we have $\Delta \vdash S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)$. It holds that $\{\tilde{\tau}_1 <: \tilde{\tau}_2\} \subseteq C$ iff either $\{\tilde{\tau}_1 <: \tilde{\tau}_2\} \subseteq \text{simplify}(C)$ or, when $\tilde{\tau}_1 = (\tilde{\tau}_{1,1} \cdot \tilde{\tau}_{1,2})$ and $\tilde{\tau}_2 = (\tilde{\tau}_{2,1} \cdot \tilde{\tau}_{2,2})$, we have $\{\tilde{\tau}_{1,1} <: \tilde{\tau}_{2,1}, \tilde{\tau}_{1,2} <: \tilde{\tau}_{2,2}\} \subseteq \text{simplify}(C)$ by definition of $\text{simplify}(C)$. The former case is trivial because $\text{simplify}(C)$ leaves $\{\tilde{\tau}_1 <: \tilde{\tau}_2\}$ intact. In the latter case, it suffices to show that $\Delta \vdash S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)$ iff $\Delta \vdash S(\tilde{\tau}_{1,1}) <: S(\tilde{\tau}_{2,1})$ and $\Delta \vdash S(\tilde{\tau}_{1,2}) <: S(\tilde{\tau}_{2,2})$, and this holds as a special case of Lemma 2.7. □

Lemma 3.7 (Constraints Equivalence for Rewriting). *For some $\{\tilde{\tau} <: \tilde{\tau}'\} \subseteq C$ and $\tilde{\tau} <: \tilde{\tau}' \Downarrow S$, then for all types $\tilde{\tau}_l$ and $\tilde{\tau}_r$, it is the case that $\Delta, C \vdash \tilde{\tau}_l <: \tilde{\tau}_r$ iff $\Delta, S(C) \vdash S(\tilde{\tau}_l) <: S(\tilde{\tau}_r)$.*

Proof. We enumerate by the three ways to derive $\Delta C \vdash \tilde{\tau}_l <: \tilde{\tau}_r$.

- case $\{\tilde{\tau}_l <: \tilde{\tau}_r\} \subseteq C$. If $\tilde{\tau}_l = \tilde{\tau}$ and $\tilde{\tau}_r = \tilde{\tau}'$, and $\tilde{\tau} <: \tilde{\tau}'$ matches one of the rewriting rules, then we examine the substitution according to the rules, as shown below.
 1. $S = \{\gamma' \mapsto (\gamma'_1 \cdot \gamma'_2)\}$. Suffices to show $\{(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\gamma'_1 \cdot \gamma'_2)\} \subseteq S(C)$ implies $\{(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: \gamma'\} \subseteq C$. This inherently holds, since it cannot be the case that $\{(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: (\gamma'_1 \cdot \gamma'_2)\} \subseteq S(C)$ but $\{(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: \gamma'\} \not\subseteq C$, for the reason that γ'_1 and γ'_2 are created fresh for the substitution (i.e., fresh variables that does not occur anywhere in C).
 2. $\gamma <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2) \Downarrow \{\gamma \mapsto (\gamma_1 \cdot \gamma_2)\}$. Similar to the argument in (1), need to show

$$\Delta, C \vdash \gamma <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2) \quad \text{iff} \quad \Delta, S(C) \vdash (\gamma_1 \cdot \gamma_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)$$

which suffices to show

$$\{\gamma <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)\} \subseteq C \quad \text{iff} \quad \{(\gamma_1 \cdot \gamma_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)\} \subseteq S(C)$$

It cannot be the case that $\{(\gamma_1 \cdot \gamma_2) <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)\} \subseteq S(C)$ but $\{\gamma <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2)\} \not\subseteq C$, since γ_1 and γ_2 are created fresh for the substitution.

Otherwise, the substitution has no effect on $\tilde{\tau}_l <: \tilde{\tau}_r$, so the equivalence is trivial.

- case $\tilde{\tau}_l = \tilde{\tau}_r$, then it is the case that $S(\tilde{\tau}_l) = S(\tilde{\tau}_r)$, so by reflexivity, both $\Delta, C \vdash \tilde{\tau}_l <: \tilde{\tau}_r$ and $\Delta, S(C) \vdash S(\tilde{\tau}_l) <: S(\tilde{\tau}_r)$ are immediately derivable.
- case using transitivity. We proceed to show both ways:
 - (\Rightarrow) Suppose there exists some $\tilde{\tau}_c$ such that $\Delta, C \vdash \tilde{\tau}_l <: \tilde{\tau}_c$ and $\Delta, C \vdash \tilde{\tau}_c <: \tilde{\tau}_r$, then we have $\Delta, S(C) \vdash S(\tilde{\tau}_l) <: S(\tilde{\tau}_c)$ and $\Delta, S(C) \vdash S(\tilde{\tau}_c) <: S(\tilde{\tau}_r)$ by induction hypothesis, which implies $\Delta, S(C) \vdash S(\tilde{\tau}_l) <: S(\tilde{\tau}_r)$ by transitivity.
 - (\Leftarrow) Suppose there exists some $\tilde{\tau}_c$ such that $\Delta, S(C) \vdash S(\tilde{\tau}_l) <: S(\tilde{\tau}_c)$ and $\Delta, S(C) \vdash S(\tilde{\tau}_c) <: S(\tilde{\tau}_r)$, then we have $\Delta, C \vdash \tilde{\tau}_l <: \tilde{\tau}_c$ and $\Delta, C \vdash \tilde{\tau}_c <: \tilde{\tau}_r$ by induction hypothesis, which implies $\Delta, C \vdash \tilde{\tau}_l <: \tilde{\tau}_r$ by transitivity.

□

Lemma 3.8 (Rewrite Rules Preserve Consistency). *Let $\{\tilde{\tau} <: \tilde{\tau}'\} \subseteq C$ and $\tilde{\tau} <: \tilde{\tau}' \Downarrow S_0$, then C is consistent iff $S_0(C)$ is consistent.*

Proof. By definition on consistency, let C be consistent when every $\{\tilde{\tau}_l <: \tilde{\tau}_r\} \subseteq C$ holds for $\Delta \vdash S(\tilde{\tau}_l) <: S(\tilde{\tau}_r)$ with some closed substitution S ; and $S_0(C)$ be consistent when every $\{\tilde{\tau}_l <: \tilde{\tau}_r\} \subseteq S_0(C)$ holds for $\Delta \vdash S'(\tilde{\tau}_l) <: S'(\tilde{\tau}_r)$ with some closed substitution S' . We enumerate on the possible S_0 according to the rewrite rules, as shown below:

1. $\tilde{\tau}_l <: \tilde{\tau}_r$ matches $(\tilde{\tau}_1 \cdot \tilde{\tau}_2) <: \gamma' \Downarrow \{\gamma' \mapsto (\gamma'_1 \cdot \gamma'_2)\}$.
 (\Rightarrow) Since S is closed and $\Delta \vdash S(\tilde{\tau}_l) <: S(\tilde{\tau}_r)$ (which is to be shown by lifting), there must be a substitution in S such that $S(\gamma') = (\tau'_1 \cdot \tau'_2)$ for some τ'_1 and τ'_2 . Construct $S_1 = \{\gamma'_1 \mapsto \tau'_1, \gamma'_2 \mapsto \tau'_2\}$, and clearly, $S(S_1(S_0(C))) = S(C)$ by overriding the substitution of γ' in S . That is, we can construct $S' = S \circ S_1$ where, if S shows that C is consistent, then S' shows $S_0(C)$ to be consistent.
 (\Leftarrow) On the other hand, if we are given S' , then there must be two substitutions in S' such that $S'(\gamma'_1) = \tau'_1$ and $S'(\gamma'_2) = \tau'_2$ for some τ'_1 and τ'_2 . Clearly, $\{\gamma' \mapsto (\tau'_1 \cdot \tau'_2)\}$ is the same as $\{\gamma'_1 \mapsto \tau'_1, \gamma'_2 \mapsto \tau'_2\} \circ \{\gamma' \mapsto (\gamma'_1 \cdot \gamma'_2)\}$, so we construct $S = S' \circ S_0$ where, if S' shows that $S_0(C)$ is consistent, then S shows C to be consistent.
2. $\tilde{\tau}_l <: \tilde{\tau}_r$ matches $\gamma <: (\tilde{\tau}'_1 \cdot \tilde{\tau}'_2) \Downarrow \{\gamma \mapsto (\gamma_1 \cdot \gamma_2)\}$. Similar to the argument in (1), one can show that, when given S , we can construct $S' = S \circ S_1$ where $S_1 = \{\gamma_1 \mapsto \tau_1, \gamma_2 \mapsto \tau_2\}$ while $S(\gamma) = (\tau_1 \cdot \tau_2)$; when given S' , we construct $S = S' \circ S_0$.
3. Otherwise, if there is no match, S_0 is the identity, so $S_0(C) = C$, and the claim is trivial.

□

Lemma 3.9 (Equivalence and Preservation of Consistency for Normalization of Constraints). *Let $C \xrightarrow[nf]{S} C'$, then*

1. $\Delta, C \vdash \tilde{\tau} <: \tilde{\tau}'$ if and only if $\Delta, C' \vdash S(\tilde{\tau}) <: S(\tilde{\tau}')$.
2. C is consistent if and only if C' is consistent.

Proof. For (1), it is shown by using Equivalence for Rewrite Rule (Lemma 3.7) and that for Simplify (Lemma 3.5) on one step of the type expansion rewrite sequence, then apply the one step proof repeatedly to get the desired result. (2) is similarly shown by using Consistency for Rewrite Rule (Lemma 3.8) and that for Simplify (Lemma 3.6) instead. □

Lemma 3.10 (Normalization Preserves Typing Derivability). *For all global-flow specification \mathcal{A} , environment $\tilde{\Gamma}$, constraints set C , and a flow type \tilde{T} , and let $C \xrightarrow[nf]{S} C'$, then $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$ is derivable iff $S(\tilde{\Gamma}), \Delta, C' \vdash \mathcal{A} : S(\tilde{T})$ is derivable.*

Proof. This is a result that can be shown by induction on the structure of typing rules, using Lemma 3.9 when a premise is a subtyping judgment. □

3.2 Checking Consistency of Constraints

Recall that given a constraint set C , C is consistent exactly when there exists a closed substitution such that all subtyping relations in $S(C)$ can be checked against Δ , i.e., $\Delta \vdash S(C)$. Since Δ is assumed to be a partial ordering relation, it is necessarily the case that $S(C) \subseteq \Delta$, so $S(C)$ must not contain subtyping relations that contradict partial ordering assumption of Δ .

One criterion for C to be consistent is that types are reflexive, but this is trivially the case given our subtyping derivation rules.

Moreover, we say that C is *closed under* Δ exactly when $\Delta, C \vdash t <: t'$ implies $\{t <: t'\} \subseteq \Delta$. If C is consistent, it must be the case that C is closed under Δ for the obvious reason. If C is not closed under Δ , then C cannot be consistent.

A substitution is said to be *induced* from a set of constraints C if the substitution forces the constraints in C to conform to antisymmetry of subtyping. In many cases, such substitution may be the identity map. However, if C

contains constraints that are contra-variant to each other, e.g., $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ and $\Delta, C \vdash \tilde{\tau}_2 <: \tilde{\tau}_1$ for some $\tilde{\tau}_1$ and $\tilde{\tau}_2$, then it must be the case that $\tilde{\tau}_1$ and $\tilde{\tau}_2$ are equal. The substitution induced by C only exists if at least one of $\tilde{\tau}_1$ or $\tilde{\tau}_2$ is a type variable, in which case we can have $S(\tilde{\tau}_1) = S(\tilde{\tau}_2)$. More generally, it is the case that if C is consistent, then the induced substitution exists. On the other hand, if an induced substitution does not exist, then C is not consistent.

Here we describe our desired definition for inducing substitution on C .

Definition 3.11 (Substitution Induced By Constraints). Let C be a set of normalized constraints. We define the function $\text{inducedBy}(C)$ as follows. If C is not consistent, then $\text{inducedBy}(C)$ aborts with “error”. If C is consistent, then $\text{inducedBy}(C)$ returns a substitution, namely:

$$\begin{aligned} \text{inducedBy}(C) = & \{ \gamma \mapsto \tau \mid \Delta, C \vdash \gamma <: \tau \text{ and } \Delta, C \vdash \tau <: \gamma \} \cup \\ & \{ \gamma' \mapsto \gamma \mid \Delta, C \vdash \gamma <: \gamma' \text{ and } \Delta, C \vdash \gamma' <: \gamma \text{ and } \gamma \prec \gamma' \} \end{aligned}$$

here “ \prec ” is a fixed, but otherwise arbitrary, total ordering on the sets of type variables, FwTypeVar and BwTypeVar . Note that the substitution returned by $\text{inducedBy}(C)$ is not necessarily closed.

However, derivation of the judgment $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ is not syntax directed, and we need a deterministic approach to solving consistency of constraints. We propose an algorithm to decide whether C is consistent and to produce an induced substitution, based on a directed graph representation G of subtyping relations. The notion of $\Delta, C \vdash \tilde{\tau} <: \tilde{\tau}'$ can be queried in the graph by asking if $v_{\tilde{\tau}} \rightsquigarrow v_{\tilde{\tau}'}$ (there is a path from $v_{\tilde{\tau}}$ to $v_{\tilde{\tau}'}$).

The first requirement for C to be consistent is that C is *closed under* Δ . In other words, if $v_t \rightsquigarrow v_{t'}$, then we require that $\{t <: t'\} \subseteq \Delta$.

The second requirement is that “ Δ, C ” (or simply “ $\Delta \cup C$ ”) is a partial order. In particular, “ Δ, C ” must be antisymmetric. This implies that if $v_{\tilde{\tau}} \rightsquigarrow v_{\tilde{\tau}'}$ and $v_{\tilde{\tau}'} \rightsquigarrow v_{\tilde{\tau}}$, then $\tilde{\tau} = \tilde{\tau}'$. We simply compute the strongly connected components SCC_1, \dots, SCC_n of G and see if all vertices of every SCC_i can be coalesced, according to the following rules:

- v_γ and $v_{\gamma'}$ can be coalesced, resulting in a substitution $\{\gamma' \mapsto \gamma\}$, and
- v_t and v_γ can be coalesced, resulting in a substitution $\{\gamma \mapsto t\}$.
- No other cases can be coalesced.

We present our consistency checking algorithm that satisfies the two requirements of consistency.

Algorithm 3.12 (Checking Consistency of Constraints). Given a set of normalized constraints C , the algorithm outputs a set of substitutions if C is consistent, and outputs “error” otherwise.

1. We construct a graph $G = (V, E)$ to represent C as the following:

$$\begin{aligned} V &= \{v_{\tilde{\tau}} \mid \tilde{\tau} \in \text{SocketType} \cup \text{TypeVar} \text{ occurs in } C.\} \\ E &= \{(v_{\tilde{\tau}}, v_{\tilde{\tau}'}) \mid \{\tilde{\tau} <: \tilde{\tau}'\} \subseteq C \text{ and } \tilde{\tau} \neq \tilde{\tau}'\} \cup \\ &\quad \{(v_t, v_{t'}) \mid v_t, v_{t'} \in V \text{ and } \{t <: t'\} \subseteq \Delta\} \end{aligned}$$

2. For every $v_t \in V$ such that $t \in \text{SocketType}$, we perform either a breadth first search or depth first search on G from v_t . For every $v_{t'}$ we encounter during the search such that $t' \in \text{SocketType}$, return “error” if $\{t <: t'\} \not\subseteq \Delta$.
3. Compute SCC_1, \dots, SCC_n , the strongly connected components of G .
4. For all $i \in \{1 \dots n\}$ where $SCC_i = (V_i, E_i)$, check that V_i has at most one vertex that represents a socket type. If not, return “error”. Otherwise we emit substitution for SCC_i according to the following two cases:

- (a) If V_i has exactly one v_t for some $t \in \text{SocketType}$, construct a substitution

$$S_i = \{ \gamma \mapsto t \mid v_\gamma \in V_i - \{v_t\} \}.$$

(b) Otherwise, choose a “least” type variable γ_0 within V_i by lexical-graphical order \prec , and construct a substitution

$$S_i = \{\gamma \mapsto \gamma_0 \mid v_\gamma \in V_i - \{v_{\gamma_0}\}\}.$$

Finally, return the substitution $S_n \circ \dots \circ S_1$.

We write $S = \text{inducedBy}(C)$ to mean letting the input of Algorithm 3.12 be C , and the output of the algorithm be S .

Lemma 3.13 (Deciding Consistency Is Efficient). *If we can decide in constant time whether an arbitrary subtyping assumption $t_1 <: t_2$ holds, i.e., whether $\{t_1 <: t_2\} \subseteq \Delta$, then we can effectively compute $\text{inducedBy}(C)$, where C is a set of normalized constraints, in low-degree polynomial time in the size of C .*

Proof. For n as the size of C , step (1) of Algorithm 3.12 takes $O(n)$ to construct V and $O(n + n^2)$ to construct E . For step (2), the number of vertices that represent socket types is bounded by n , and each iterations of searches are also bounded by n , so step (2) runs in $O(n^2)$ time. For step (3), we determine strongly connected components SCC_1, \dots, SCC_n in time linear to the size of E using depth first search twice, and the size of E is in the same order as size of C , so this step runs in $O(n)$ time. Step (4) takes a total of $O(n)$, since $|V| = \sum_i^n |V_i|$ where $SCC_i = (V_i, E_i)$. Overall, the algorithm runs in $O(n^2)$ time due to step (1) and step (2). \square

3.2.1 Correctness

Lemma 3.14 (Relating Consistency to Strongly Connected Components). *Let G be a graph representation of Δ and C be a set of normalized constraints closed under Δ . All strongly connected components in G contain at most one socket type vertex (other vertices of the component are type variables) iff C is consistent.*

Proof. We proceed to prove the claim in two directions:

(\Rightarrow) For every $\tilde{\tau}_1$ and $\tilde{\tau}_2$ in a strongly connected component, we have $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ and $\Delta, C \vdash \tilde{\tau}_2 <: \tilde{\tau}_1$, so it must be $\tilde{\tau}_1 = \tilde{\tau}_2$ for Δ, C to be consistent. If $\tilde{\tau}_1$ and $\tilde{\tau}_2$ are both `SocketType`, they cannot be equal (otherwise they would have been the same vertex). If one of $\tilde{\tau}_1$ or $\tilde{\tau}_2$ is a `TypeVar`, then there exists a substitution S by substituting $\tilde{\tau}_1$ for $\tilde{\tau}_2$ (or vice versa) and keep $S(C)$ consistent with Δ . Hence C is consistent.

(\Leftarrow) If C is consistent, then there is no $\tilde{\tau}_1$ and $\tilde{\tau}_2$, $\tilde{\tau}_1 \neq \tilde{\tau}_2$ such that $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ and $\Delta, C \vdash \tilde{\tau}_2 <: \tilde{\tau}_1$. Furthermore, there exists a closed substitution S such that all type variables in C are substituted to a closed socket type. Let $G = (V, E)$ represent $\Delta, S(C)$. It follows that G is acyclic, and all strongly connected components in G are simply a vertex by itself. \square

Lemma 3.15 (Equivalence of Constraint Sets With Induced Substitution). *Let C be a finite constraint set. If C is consistent and $S = \text{inducedBy}(C)$, then C and $S(C)$ are equivalent, i.e.,*

$$\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2 \quad \text{iff} \quad \Delta, S(C) \vdash S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)$$

for all socket types $\tilde{\tau}_1$ and $\tilde{\tau}_2$.

Proof. By the definition of $\text{inducedBy}(C)$, it is the case that $\Delta, C \vdash S(\tilde{\tau}_1) <: \tilde{\tau}_1$ and $\Delta, C \vdash \tilde{\tau}_2 <: S(\tilde{\tau}_2)$, so we have $\Delta, C \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ iff $\Delta, C \vdash S(\tilde{\tau}_1) <: \tilde{\tau}_1 <: \tilde{\tau}_2 <: S(\tilde{\tau}_2)$ iff $\Delta, S(C) \vdash S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)$. \square

Lemma 3.16. Induced Substitution Preserves Typing Derivability] *For all global-flow specification \mathcal{A} , environment $\tilde{\Gamma}$, normalized constraints set C , and a flow type \tilde{T} , and let $S = \text{inducedBy}(C)$, then $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$ is derivable iff $S(\tilde{\Gamma}), \Delta, S(C) \vdash \mathcal{A} : S(\tilde{T})$ is derivable.*

Proof. This is a result that can be shown by induction on the structure of typing rules, using Lemma 3.15 when a premise is a subtyping judgment. \square

3.3 Compositional Analysis

We define a procedure \mathcal{P}_C —our desired *type analysis* as shown in in Figure 3.1—which, given an arbitrary global flow \mathcal{A} as input, always terminates and returns an output, denoted $\mathcal{P}_C(\mathcal{A})$, in one of two cases:

1. $\mathcal{P}_C(\mathcal{A}) = \text{'no-solution'}$, meaning that \mathcal{A} is not typable.
2. $\mathcal{P}_C(\mathcal{A}) = \langle \tilde{\Gamma}, C, \tilde{T} \rangle$, where $\tilde{\Gamma}$ is a type environment, C is a consistent set (possibly empty) of constraints, and \tilde{T} is a type.

\mathcal{P}_C produces a principal typing that consists of an environment, a set of consistent constraints, and a flow type that may contain type variables. Given a global-flow on input, it breaks down the composition of flow, computes principal typing of the pieces, and compose the typings. It adds more constraints to the typing for those subtyping relations that need to hold for the flow composition it just broke down. It checks that the resulting constraints are still consistent and returns the typing if it is the case.

The result 'no-solution' indicates that composition within \mathcal{A} was inferred to have resulted in an internal incompatibility. On the other hand, if \mathcal{P}_C returns $\langle \tilde{\Gamma}, C, \tilde{T} \rangle$, that means there is no known compatibility issues when the blanks in \mathcal{A} are left open, but there may still be a problem when one or more of the blanks are filled in with incompatible controllers.

\mathcal{P}_C makes use of an additional function that merges two environments from the two typings to be composed, defined below.

Definition 3.17 (Merger of Environments). Given $\tilde{\Gamma}_1$ and $\tilde{\Gamma}_2$, we define a function $\text{mergeEnv}(\tilde{\Gamma}_1, \tilde{\Gamma}_2)$ that returns a pair $\langle \tilde{\Gamma}, C \rangle$ of some environment $\tilde{\Gamma}$ and a constraint set C such that

$$\begin{aligned} \tilde{\Gamma} &= \tilde{\Gamma}_1 \cup \left\{ x : \tilde{\Gamma}_2(x) \mid x \in \text{dom}(\tilde{\Gamma}_2) \text{ and } x \notin \text{dom}(\tilde{\Gamma}_1) \right\} \\ C &= \left\{ \tilde{\Gamma}_1(x) \doteq \tilde{\Gamma}_2(x) \mid x \in \text{dom}(\tilde{\Gamma}_1) \cap \text{dom}(\tilde{\Gamma}_2) \right\} \end{aligned}$$

Furthermore, we say that $\tilde{\Gamma}_1 \cup \tilde{\Gamma}_2$ yields a *merger constraint* C for the pair $\langle \tilde{\Gamma}, C \rangle$ returned by $\text{mergeEnv}(\tilde{\Gamma}_1, \tilde{\Gamma}_2)$.

We will proceed to prove the following result.

Theorem 3.18 (Solvability and Principality). *Let \mathcal{A} be a global-flow specification. Suppose \mathcal{P}_C terminates and returns the triple $\mathcal{P}_C(\mathcal{A}) = \langle \tilde{\Gamma}, C, \tilde{T} \rangle$. It then holds that:*

1. *There is a typing derivation with final judgment*

$$\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$$

i.e., there is a typing for \mathcal{A} that assigns it the type \tilde{T} .

2. *For every substitution S , not necessarily closed, if $S(C)$ is consistent, then there is a typing derivation where the final judgment is*

$$S(\tilde{\Gamma}), \Delta, S(C) \vdash \mathcal{A} : S(\tilde{T})$$

3. *For every typing derivation with final judgment*

$$\Gamma, \Delta \vdash \mathcal{A} : T$$

there is a closed substitution S such that

- (a) $\Gamma = S(\tilde{\Gamma})$,
- (b) $\Delta \vdash S(C)$,
- (c) $T = S(\tilde{T})$

Properties 1, 2, and 3, together, mean $\langle \tilde{\Gamma}, C, \tilde{T} \rangle$ defines a “principal typing” for \mathcal{A} .

Proof. We break the properties down to Lemma 2.12 above, and Lemma 3.20 and 3.21 below. □

Figure 3.1: Compositional Analysis Algorithm \mathcal{P}_C

1. **IF** $\mathcal{A} = A$ **THEN RETURN** $\langle \emptyset, \emptyset, \text{type}(\mathcal{A}) \rangle$.
2. **IF** $\mathcal{A} = x$ **THEN**
 - LET** $\tilde{T} = \begin{bmatrix} \alpha_{x,1} & \alpha_{x,2} \\ \beta_{x,1} & \beta_{x,2} \end{bmatrix}$ **WHERE** $\alpha_{x,1}, \alpha_{x,2}, \beta_{x,1}, \beta_{x,2}$ **FRESH**
 - IN** $\langle \{x : \tilde{T}\}, \emptyset, \tilde{T} \rangle$.
3. **IF** $\mathcal{A} = (\mathcal{A}_1; \mathcal{A}_2)$ **THEN**
 - LET** $\langle \tilde{\Gamma}_1, C_1, \tilde{T}_1 \rangle = \mathcal{P}_C(\mathcal{A}_1)$,
 - $\langle \tilde{\Gamma}_2, C_2, \tilde{T}_2 \rangle = \mathcal{P}_C(\mathcal{A}_2)$
 - IN** **LET** $\langle \tilde{\Gamma}, C_0 \rangle = \text{mergeEnv}(\tilde{\Gamma}_1, \tilde{\Gamma}_2)$,
 - $C = C_0 \cup C_1 \cup C_2 \cup \{ \text{f-out}(\tilde{T}_1) <: \text{f-in}(\tilde{T}_2), \text{b-out}(\tilde{T}_2) <: \text{b-in}(\tilde{T}_1) \}$,
 - $\tilde{T} = \begin{bmatrix} \text{f-in}(\tilde{T}_1) & \text{f-out}(\tilde{T}_2) \\ \text{b-out}(\tilde{T}_1) & \text{b-in}(\tilde{T}_2) \end{bmatrix}$,
 - $\langle S_1, C' \rangle = \text{normalizeConstraints}(C)$,
 - $S_2 = \text{inducedBy}(C')$
 - IN** **IF** $S_2 = \text{error}$ **THEN FAILWITH** 'no-solution' **ELSE**
 - LET** $S = S_2 \circ S_1$ **IN RETURN** $\langle S(\tilde{\Gamma}), S_2(C'), S(\tilde{T}) \rangle$.
4. **IF** $\mathcal{A} = (\mathcal{A}_1 \parallel \mathcal{A}_2)$ **THEN**
 - LET** $\langle \tilde{\Gamma}_1, C_1, \tilde{T}_1 \rangle = \mathcal{P}_C(\mathcal{A}_1)$,
 - $\langle \tilde{\Gamma}_2, C_2, \tilde{T}_2 \rangle = \mathcal{P}_C(\mathcal{A}_2)$
 - IN** **LET** $\langle \tilde{\Gamma}, C_0 \rangle = \text{mergeEnv}(\tilde{\Gamma}_1, \tilde{\Gamma}_2)$,
 - $C = C_0 \cup C_1 \cup C_2$,
 - $\tilde{T} = \tilde{T}_1 \bullet \tilde{T}_2$,
 - $\langle S_1, C' \rangle = \text{normalizeConstraints}(C)$,
 - $S_2 = \text{inducedBy}(C')$
 - IN** **IF** $S_2 = \text{error}$ **THEN FAILWITH** 'no-solution' **ELSE**
 - LET** $S = S_2 \circ S_1$ **IN RETURN** $\langle S(\tilde{\Gamma}), S_2(C'), S(\tilde{T}) \rangle$.
5. **IF** $\mathcal{A} = (\text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}_2)$ **THEN**
 - LET** $\langle \tilde{\Gamma}_1, C_1, \tilde{T}_1 \rangle = \mathcal{P}_C(\mathcal{A}_1)$,
 - $\langle \tilde{\Gamma}_2, C_2, \tilde{T}_2 \rangle = \mathcal{P}_C(\mathcal{A}_2)$
 - IN** **LET** $\langle \tilde{\Gamma}, C_0 \rangle = \text{mergeEnv}(\tilde{\Gamma}_1, \tilde{\Gamma}_2 - \{x : \tilde{\Gamma}_2(x)\})$,
 - $C = C_0 \cup C_1 \cup C_2 \cup$
 - $\{ \text{f-in}(\tilde{\Gamma}_2(x)) <: \text{f-in}(\tilde{T}_1), \text{f-out}(\tilde{T}_1) <: \text{f-out}(\tilde{\Gamma}_2(x)),$
 - $\text{b-in}(\tilde{\Gamma}_2(x)) <: \text{b-in}(\tilde{T}_1), \text{b-out}(\tilde{T}_1) <: \text{b-out}(\tilde{\Gamma}_2(x)) \}$,
 - $\tilde{T} = \tilde{T}_2$,
 - $\langle S_1, C' \rangle = \text{normalizeConstraints}(C)$,
 - $S_2 = \text{inducedBy}(C)$
 - IN** **IF** $S_2 = \text{error}$ **THEN FAILWITH** 'no-solution' **ELSE**
 - LET** $S = S_2 \circ S_1$ **IN RETURN** $\langle S(\tilde{\Gamma}), S_2(C'), S(\tilde{T}) \rangle$.

Lemma 3.19 (Correctness of Environment Merging). *If $\tilde{\Gamma}_1, \Delta, C_1 \vdash \mathcal{A}_1 : \tilde{T}_1$ and $\tilde{\Gamma}_2, \Delta, C_2 \vdash \mathcal{A}_2 : \tilde{T}_2$ are derivable, and $\langle \tilde{\Gamma}, C_0 \rangle = \text{mergeEnv}(\tilde{\Gamma}_1, \tilde{\Gamma}_2)$, and $C_1 \cup C_2 \cup C_0$ is consistent, then*

1. $\tilde{\Gamma}, \Delta, C_1 \cup C_0 \vdash \mathcal{A}_1 : \tilde{T}_1$ is derivable.
2. $\tilde{\Gamma}, \Delta, C_2 \cup C_0 \vdash \mathcal{A}_2 : \tilde{T}_2$ is derivable.

Proof. □

1. By Definition 3.17, $\tilde{\Gamma}_1 \subseteq \tilde{\Gamma}$, so by Lemma 2.10, $\tilde{\Gamma}, \Delta, C_1 \vdash \mathcal{A}_1 : \tilde{T}_1$ is derivable. Finally, by Lemma 2.11, $\tilde{\Gamma}, \Delta, C_1 \cup C_0 \vdash \mathcal{A}_1 : \tilde{T}_1$ is derivable.
2. Suppose $\tilde{\Gamma}_2, \Delta, C_2 \vdash \mathcal{A}_2 : \tilde{T}_2$ is derivable. By Lemma 2.11, $\tilde{\Gamma}_2, \Delta, C_2 \cup C_0 \vdash \mathcal{A}_2 : \tilde{T}_2$ is derivable. Let $S = \text{inducedBy}(C_0)$. Note that $S \neq \text{'error'}$ since, by Lemma 2.5, $C_1 \cup C_2 \cup C_0$ being consistent implies that C_0 is consistent. By Lemma 2.12, $S(\tilde{\Gamma}_2), \Delta, S(C_2 \cup C_0) \vdash \mathcal{A}_2 : S(\tilde{T}_2)$ is derivable. Note that $S(\tilde{\Gamma}_2) \subseteq \tilde{\Gamma}$, so by Lemma 2.10, $\tilde{\Gamma}, \Delta, S(C_2 \cup C_0) \vdash \mathcal{A}_2 : S(\tilde{T}_2)$ is derivable. By Lemma 3.15, we have $\Delta, C_2 \cup C_0 \vdash \tilde{\tau}_1 <: \tilde{\tau}_2$ iff $\Delta, S(C_2 \cup C_0) \vdash S(\tilde{\tau}_1) <: S(\tilde{\tau}_2)$, so $\tilde{\Gamma}, \Delta, C_2 \cup C_0 \vdash \mathcal{A}_2 : S(\tilde{T}_2)$ is derivable, and that $\Delta, C_2 \cup C_0 \vdash S(\tilde{T}_2) <: \tilde{T}_2$; finally, $\tilde{\Gamma}, \Delta, C_2 \cup C_0 \vdash \mathcal{A}_2 : \tilde{T}_2$ is derivable.

Lemma 3.20 (Type Derivation for Compositional Analysis Typing). *If $\mathcal{P}_C(\mathcal{A}) = \langle \tilde{\Gamma}^*, C^*, \tilde{T}^* \rangle$, then there is a typing derivation with final judgment $\tilde{\Gamma}^*, \Delta, C^* \vdash \mathcal{A} : \tilde{T}^*$.*

Proof. Induction by cases in the compositional analysis algorithm in Figure 3.1. In the proof, we refer to pseudo-code variables in the algorithm for the corresponding case.

1. If $\mathcal{A} = A$, then

$$\frac{\text{type}(A) = \tilde{T}^*}{\emptyset, \Delta, \emptyset \vdash A : \tilde{T}^*}$$

is immediately satisfied.

2. If $\mathcal{A} = x$, then

$$\frac{\tilde{\Gamma}^*(x) = \tilde{T}^*}{\tilde{\Gamma}^*, \Delta, \emptyset \vdash x : \tilde{T}^*}$$

is satisfied because $\tilde{\Gamma}^* = \{x : \tilde{T}^*\}$ for some $\tilde{T}^* = \begin{bmatrix} \alpha_1 & \alpha_2 \\ \beta_1 & \beta_2 \end{bmatrix}$ where the type variables are fresh.

3. If $\mathcal{A} = (\mathcal{A}_1; \mathcal{A}_2)$, then we show the final judgment is derivable by showing that the premises of the (seq $\tilde{}$) rule are all derivable. By induction hypothesis on results of $\mathcal{P}_C(\mathcal{A}_1)$ and $\mathcal{P}_C(\mathcal{A}_2)$, $\tilde{\Gamma}_1, \Delta, C_1 \vdash \mathcal{A}_1 : \tilde{T}_1$ and $\tilde{\Gamma}_2, \Delta, C_2 \vdash \mathcal{A}_2 : \tilde{T}_2$ are both derivable, from which both $\tilde{\Gamma}, \Delta, C_1 \cup C_0 \vdash \mathcal{A}_1 : \tilde{T}_1$ and $\tilde{\Gamma}, \Delta, C_2 \cup C_0 \vdash \mathcal{A}_2 : \tilde{T}_2$ are derivable by Lemma 3.19. Since $C_1 \cup C_0 \subseteq C$ and $C_2 \cup C_0 \subseteq C$, by Lemma 2.11, both $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_1 : \tilde{T}_1$ and $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A}_2 : \tilde{T}_2$ are derivable. Finally, $\tilde{\Gamma}^*, \Delta, C^* \vdash \mathcal{A}_1 : \tilde{T}_1^*$ and $\tilde{\Gamma}^*, \Delta, C^* \vdash \mathcal{A}_2 : \tilde{T}_2^*$ are derivable due to Lemma 3.10 and Lemma 3.16.

$\Delta, C^* \vdash \text{f-out}(\tilde{T}_1) <: \text{f-in}(\tilde{T}_2)$ and $\Delta, C^* \vdash \text{b-out}(\tilde{T}_2) <: \text{b-in}(\tilde{T}_1)$ are both derivable because $\{\text{f-out}(\tilde{T}_1) <: \text{f-in}(\tilde{T}_2), \text{b-out}(\tilde{T}_2) <: \text{b-in}(\tilde{T}_1)\} \subseteq C$, and we have $\Delta, C \vdash \text{f-out}(\tilde{T}_1) <: \text{f-in}(\tilde{T}_2)$ implies $\Delta, C^* \vdash \text{f-out}(\tilde{T}_1) <: \text{f-in}(\tilde{T}_2)$ and $\Delta, C \vdash \text{b-out}(\tilde{T}_2) <: \text{b-in}(\tilde{T}_1)$ implies $\Delta, C^* \vdash \text{b-out}(\tilde{T}_2) <: \text{b-in}(\tilde{T}_1)$ by Lemma 3.9, Lemma 3.15, and Lemma 2.8.

4. If $\mathcal{A} = (\mathcal{A}_1 \parallel \mathcal{A}_2)$, then we may show the final judgment is derivable by showing, in a way similar to (3) above, that the premises of the (par $\tilde{}$) rule, $\tilde{\Gamma}^*, \Delta, C^* \vdash \mathcal{A}_1 : \tilde{T}_1^*$ and $\tilde{\Gamma}^*, \Delta, C^* \vdash \mathcal{A}_2 : \tilde{T}_2^*$, are both derivable.
5. If $\mathcal{A} = \text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}_2$, then we may show the final judgment is derivable similar to the way it is done in (3) above, noting that $(\tilde{\Gamma}_2 - \{x : \tilde{\Gamma}_2(x)\}) \cup \{x : \tilde{T}_1\} \subseteq \tilde{\Gamma} \cup \{x : \tilde{T}_1\}$, and that $\Delta, C^* \vdash \tilde{T}_1^* \doteq \tilde{\Gamma}_2(x)$ is derivable.

□

Lemma 3.21 (Correctness of Compositional Analysis). *If $\mathcal{P}_C(\mathcal{A}) = \langle \tilde{\Gamma}, C, \tilde{T} \rangle$ and $\Gamma, \Delta \vdash \mathcal{A} : T$, then there exists a closed substitution S such that $\Gamma = S(\tilde{\Gamma})$, $T = S(\tilde{T})$, and $\Delta \vdash S(C)$.*

Proof. We proceed to prove this by enumerating the possible ways to derive the judgment $\Gamma, \Delta \vdash \mathcal{A} : T$.

- case \mathcal{A} is A , then S is the identity since $\tilde{T} = T$ is a closed type.
- case \mathcal{A} is x , $\tilde{T} = \begin{bmatrix} \alpha_1 & \alpha_2 \\ \beta_1 & \beta_2 \end{bmatrix}$, and $T = \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$, then $S = \{\alpha_1 \mapsto \rho_1, \alpha_2 \mapsto \rho_2, \beta_1 \mapsto \sigma_1, \beta_2 \mapsto \sigma_2\}$.
- case \mathcal{A} is $\mathcal{A}_1 \parallel \mathcal{A}_2$, then we proceed directly by induction hypothesis on $\mathcal{A}_1, \mathcal{A}_2$. $S = S_1 \circ S_2$, where S_1 is such closed substitution for \mathcal{A}_1 , and S_2 for \mathcal{A}_2 .
- case \mathcal{A} is $\mathcal{A}_1; \mathcal{A}_2$, we show that subtyping judgment introduced in this derivation will satisfy $\Delta \vdash S(C)$, then proceed by induction hypothesis on $\mathcal{A}_1, \mathcal{A}_2$.

Suppose $\mathcal{P}_C(\mathcal{A}_1) = \langle \tilde{\Gamma}_1, C_1, \begin{bmatrix} \tilde{\rho}_1 & \tilde{\rho}_2 \\ \tilde{\sigma}_1 & \tilde{\sigma}_2 \end{bmatrix} \rangle$ and $\mathcal{P}_C(\mathcal{A}_2) = \langle \tilde{\Gamma}_2, C_2, \begin{bmatrix} \tilde{\rho}_3 & \tilde{\rho}_4 \\ \tilde{\sigma}_3 & \tilde{\sigma}_4 \end{bmatrix} \rangle$. By Lemma 3.20, we have subtyping judgment of the form $\Delta, C \vdash \tilde{\rho}_2 <: \tilde{\rho}_3$ and $\Delta, C \vdash \tilde{\sigma}_3 <: \tilde{\sigma}_2$ as the premise for deriving $\tilde{\Gamma}, \Delta, C \vdash \mathcal{A} : \tilde{T}$. We can construct a substitution to produce $\Delta \vdash \rho_2 <: \rho_3$ and $\Delta \vdash \sigma_3 <: \sigma_2$ respectively (for $\Gamma, \Delta \vdash \mathcal{A}_1 : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$ and $\Gamma, \Delta \vdash \mathcal{A}_2 : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}$) in the following way:

$$\begin{aligned}
S_0(\gamma) &= \begin{cases} \rho_2 & \text{if } \tilde{\rho}_2 \in \text{TypeVar and } \gamma = \tilde{\rho}_2 \\ \rho_3 & \text{if } \tilde{\rho}_3 \in \text{TypeVar and } \gamma = \tilde{\rho}_3 \\ \sigma_3 & \text{if } \tilde{\sigma}_3 \in \text{TypeVar and } \gamma = \tilde{\sigma}_3 \\ \sigma_2 & \text{if } \tilde{\sigma}_2 \in \text{TypeVar and } \gamma = \tilde{\sigma}_2 \end{cases} \\
S_1 &= \text{closed substitution from analyzing } \mathcal{A}_1 \\
S_2 &= \text{closed substitution from analyzing } \mathcal{A}_2 \\
S &= (S_1 \circ S_2) \circ S_0
\end{aligned}$$

Using Preservation of Consistency (clause (2) of Lemma 3.9) and Equivalence on Induced Substitution (Lemma 3.15), it is clear that $\Delta \vdash S(C)$, since C is consistent. Furthermore, S is closed by induction hypothesis on \mathcal{A}_1 and \mathcal{A}_2 .

- case \mathcal{A} is **let** $x = \mathcal{A}_1$ **in** \mathcal{A}_2 , suppose $\Gamma, \Delta \vdash \mathcal{A}_1 : T_1$ and $\mathcal{P}_C(\mathcal{A}_2) = \langle \tilde{\Gamma}_2, C_2, \tilde{T}_2 \rangle$, and that $\tilde{\Gamma}_2(x) = \begin{bmatrix} \tilde{\rho}_1 & \tilde{\rho}_2 \\ \tilde{\sigma}_1 & \tilde{\sigma}_2 \end{bmatrix}$, $T_1 = \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$. We simply construct a substitution in the following way:

$$\begin{aligned}
S_0(\gamma) &= \begin{cases} \rho_1 & \text{if } \tilde{\rho}_1 \in \text{TypeVar and } \gamma = \tilde{\rho}_1 \\ \rho_2 & \text{if } \tilde{\rho}_2 \in \text{TypeVar and } \gamma = \tilde{\rho}_2 \\ \sigma_1 & \text{if } \tilde{\sigma}_1 \in \text{TypeVar and } \gamma = \tilde{\sigma}_1 \\ \sigma_2 & \text{if } \tilde{\sigma}_2 \in \text{TypeVar and } \gamma = \tilde{\sigma}_2 \end{cases} \\
S_1 &= \text{closed substitution from analyzing } \mathcal{A}_1 \\
S_2 &= \text{closed substitution from analyzing } \mathcal{A}_2 \\
S &= (S_1 \circ S_2) \circ S_0
\end{aligned}$$

□

This concludes the correctness of our compositional analysis algorithm \mathcal{P}_C .

4 Conclusion

We presented a compositional analysis algorithm for the language TRAFFIC that satisfies principal typing property. In order to achieve this, we introduced type variables to denote unknown types, and extended subtyping relation to check against constraints, which allows us to defer the actual verification against subtyping assumptions in Δ until we have more information. Inconsistent constraints mean a violation of Δ , in which case the compositional analysis

aborts with error. We also presented an algorithm to decide whether a set of constraints is consistent and showed that it is correct.

Our approach to decide for consistency of a set of constraints involves normalizing the constraints and run a graph analysis on the normalized constraints. We have shown that graph analysis runs in $O(n^2)$ time, but normalization can run in exponential time using naive data representation of constraints. It is known that if the constraints are expressed as directed acyclic graphs, then normalization can be run in low-degree polynomial time as well.

Assuming that Algorithms 3.3 and 3.12 run in low-degree polynomial time, then compositional analysis also runs in low-degree polynomial time to the size of the input. Therefore, compositional analysis can do at least as well as syntax-directed type checking. Furthermore, compositional analysis has the advantage that it can deal with holes in a flow specification, and that principal typing of a flow specification needs only be computed once—one can reuse typing of the same flow specification to see, for example, if the hole can be fitted with several candidates. Therefore, we consider compositional analysis an improved type analysis technique over syntax-directed analysis.

References

- [1] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Ibrahim Matta. Safe Compositional Specification of Networking Systems. Technical Report BUCS-TR-2004-021, CS Department, Boston University, May 14 2004.
- [2] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Ibrahim Matta. Typed Abstraction of Complex Network Compositions. Technical Report BUCS-TR-2005-014, CS Department, Boston University, May 1 2005.
- [3] Likai Liu, Assaf Kfoury, Azer Bestavros, Adam Bradley, Yarom Gabay, and Ibrahim Matta. Safe Compositional Specification of Networking Systems: TRAFFIC The Language and Its Type Checking. Technical Report BUCS-TR-2005-015, CS Department, Boston University, May 12 2005.
- [4] Joe B. Wells. The essence of principal typings. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer, 2002.