

Safe Compositional Specification of Network Systems* With Polymorphic, Constrained Types

LIKAI LIU

liulk@cs.bu.edu

ASSAF KFOURY

kfoury@cs.bu.edu

Computer Science Department
Boston University

25th October 2006

Technical report: BUCS-TR-2006-029

Abstract

In the framework of iBench research project, our previous work created a domain specific language TRAFFIC [6] that facilitates specification, programming, and maintenance of distributed applications over a network. It allows safety property to be formalized in terms of types and subtyping relations. Extending upon our previous work, we add Hindley-Milner style polymorphism [8] with constraints [9] to the type system of TRAFFIC. This allows a programmer to use for-all quantifier to describe types of network components, escalating power and expressiveness of types to a new level that was not possible before with propositional subtyping relations. Furthermore, we design our type system with a pluggable constraint system, so it can adapt to different application needs while maintaining soundness.

In this paper, we show the soundness of the type system, which is not syntax-directed but is easier to do typing derivation. We show that there is an equivalent syntax-directed type system, which is what a type checker program would implement to verify the safety of a network flow. This is followed by discussion on several constraint systems: polymorphism with subtyping constraints, Linear Programming, and Constraint Handling Rules (CHR) [3]. Finally, we provide some examples to illustrate workings of these constraint systems.

1 Introduction

Programming on a computer has enjoyed benefits of static program verification by means of type systems. A compiler that implements a type system can mechanically detect certain programming mistakes. Mature type theory and type checking technologies have been developed to become more expressive at stating a program's correctness properties. This has the advantage of both accurate and thorough coverage, which cuts debugging time and cost considerably. Sound type systems sustain Robin Milner's slogan that "well-typed programs do not go wrong" [8].

With the advent of networked computers and mobile devices, those in the position of programming the network do not yet have the benefits of a type system. In our earlier report [6], we introduced a specification language for network flow composition, which we call TRAFFIC (Typed Representation and Analysis of network Flows For Interoperability Checks). This formalizes (1) a domain specific language that describes interconnection of network components, and (2) a type system to certify the correctness of configurations.

A type system consists of a set of typing rules for every syntax cases of a language, and it is through these rules a program may be accepted or rejected for correctness. Since programs are expressions built from smaller subexpressions, type system rules work by certifying correctness of subexpressions first, then check if correctness still holds for the whole expression. In our work, we treat a flow specification as a program, and we design type system for TRAFFIC to verify correctness of a *flow*.

*This work is partially supported by NSF Award No. CCR-0205294

x, y, z	\in	FlowVar		flow variable
A, B	\in	LocalFlow		local flow
\mathcal{A}, \mathcal{B}	\in	GlobalFlow	$::=$	
			$x \mid A$	
			$\mathcal{A}; \mathcal{B}$	sequential flow
			$\mathcal{A} \parallel \mathcal{B}$	parallel flow
			let $x = \mathcal{A}$ in \mathcal{B}	let-binding

Figure 1.1: Syntax of TRAFFIC specifications.

In this framework, a *flow* is a box with four connectors: inputs and outputs of forward and backward directions. We assign a *type* to each connector. Whenever we make a connection from an output to an input, we mandate that the type representing the output has to be a *subtype* of the type representing the input. Flows can be composed in parallel, in which case we couple the connectors but no connections are made; or flows can be composed sequentially, in which case we connect the outputs to the inputs of the flows side by side.

By programming the subtyping relation, one can effectively customize the type system for various applications, such as network calculus, scheduling, queuing theory, and control theory [1]. In practice, we find it easy to program the flows, i.e., specify how flows are connected, but it is much harder to program subtyping relations. This is due to (1) difficulty in determining subtyping properties that are appropriate for a particular application; and (2) limited expressiveness of built-in subtyping relation primitives in the implementation.

In this paper, we address the latter problem by allowing programmers define arbitrary type constructors, constraint predicates, and constraint handling rules that describe how custom constraint predicates are satisfied. Subtyping and equality relation can be seen as special constraint predicates with built-in rules for satisfiability.

1.1 Syntax of TRAFFIC

Specifications written in TRAFFIC consist of *flows*. Our concept of a flow can be visualized as a box with some input and output connection points, and the idea is to model end-to-end delivery of a signal through this box. Flows can be combined by connecting the output of one flow to the input of another, forming a larger conceptual box as the result. In this manner, large, composite flows are built up from small, atomic flows.

The smallest units of a flow are *local flows* and *flow variables*. A *local flow* can be seen an off-the-shelf component with known interfacing properties at its connectors. Interfacing properties, or *flow types*, of local flows are predefined by a vendor and must be provided to the type system beforehand. A *flow variable* is a placeholder for unknown flows to be specified in a *let-binding*.

Composite flows, also called *global flows*, are created by arranging two flows in *sequence* or in *parallel* (see Figure ?). A third form of composite flow is the *let-binding*, which simply says that suppose a flow \mathcal{A} and a variable x are given, then \mathcal{A} is substituted for each occurrences of x in some let-body flow \mathcal{B} by duplicating the specification of \mathcal{A} . This provides a syntactic sugar for abstracting common flows, thus allowing modular design of flows.

A formal definition of global flow syntax in BNF notation is found in Figure 1.1. Our notational convention is to use x, y, z to range over flow variables, A, B to range over local flows, and calligraphic \mathcal{A}, \mathcal{B} to range over global flows.

Example 1.1 (Valid Flows). The following lists some examples of valid flows (according to syntax).

- $A \parallel (x; y)$ is a valid flow. Note that both “ \parallel ” and “ $;$ ” are binary operators, and one should parenthesize where ambiguity may arise.
- **let** $x = (\text{let } y = A; B \text{ in } y \parallel y)$ **in** $x; x$ is a valid flow. The parentheses that surround the inner **let** are optional because **let** ... **in** effectively disambiguates the scope.

We proceed to define formally the semantics of let-binding based on the notion of flow substitution: given two flows \mathcal{A} and \mathcal{B} , suppose we were to substitute \mathcal{A} for every occurrences of variable x in \mathcal{B} , we write $[x := \mathcal{A}] \mathcal{B}$, which is defined below.

Definition 1.2 (Substitution of Flows). A flow substitution $[x := \mathcal{A}]\mathcal{B}$ is carried out recursively by cases of \mathcal{B} as follows.

$$\begin{aligned}
[x := \mathcal{A}]y &= \begin{cases} \mathcal{A} & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \\
[x := \mathcal{A}]A &= A \\
[x := \mathcal{A}](\mathcal{B}_1 ; \mathcal{B}_2) &= ([x := \mathcal{A}]\mathcal{B}_1) ; ([x := \mathcal{A}]\mathcal{B}_2) \\
[x := \mathcal{A}](\mathcal{B}_1 \parallel \mathcal{B}_2) &= ([x := \mathcal{A}]\mathcal{B}_1) \parallel ([x := \mathcal{A}]\mathcal{B}_2) \\
[x := \mathcal{A}](\text{let } y = \mathcal{B}_1 \text{ in } \mathcal{B}_2) &= \text{let } y' = [x := \mathcal{A}]\mathcal{B}_1 \text{ in } [x := \mathcal{A}][y := y']\mathcal{B}_2 \quad y' \text{ is fresh.}
\end{aligned}$$

Notice the special treatment of let-binding, which is written in order to avoid unintended variable name capturing when variable y appears in \mathcal{A} .

For the purpose of proving subject reduction (an essential theorem about the type system) in a later section, we supply the operational semantics of flow reduction below. A flow is reduced by eliminating syntactic sugar let-bindings, carrying out the desired flow substitution recursively. A flow is said to be in *normal form* if it cannot be further reduced, hence a normal flow has no let-bindings.

Definition 1.3 (Reduction on Flow). Given a flow \mathcal{A} , we say \mathcal{A} reduce to some flow \mathcal{A}' , written as $\mathcal{A} \rightarrow \mathcal{A}'$, according to the axiom:

$$\text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}_2 \rightarrow [x := \mathcal{A}_1]\mathcal{A}_2$$

and the rules below:

$$\begin{array}{c}
\frac{\mathcal{A}_1 \rightarrow \mathcal{A}'_1}{\mathcal{A}_1 ; \mathcal{A}_2 \rightarrow \mathcal{A}'_1 ; \mathcal{A}_2} \quad \frac{\mathcal{A}_1 \rightarrow \mathcal{A}'_1}{\mathcal{A}_1 \parallel \mathcal{A}_2 \rightarrow \mathcal{A}'_1 \parallel \mathcal{A}_2} \quad \frac{\mathcal{A}_1 \rightarrow \mathcal{A}'_1}{(\text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}_2) \rightarrow (\text{let } x = \mathcal{A}'_1 \text{ in } \mathcal{A}_2)} \\
\frac{\mathcal{A}_2 \rightarrow \mathcal{A}'_2}{\mathcal{A}_1 ; \mathcal{A}_2 \rightarrow \mathcal{A}_1 ; \mathcal{A}'_2} \quad \frac{\mathcal{A}_2 \rightarrow \mathcal{A}'_2}{\mathcal{A}_1 \parallel \mathcal{A}_2 \rightarrow \mathcal{A}_1 \parallel \mathcal{A}'_2} \quad \frac{\mathcal{A}_2 \rightarrow \mathcal{A}'_2}{(\text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}_2) \rightarrow (\text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}'_2)}
\end{array}$$

Remark 1.4 (Confluence). Reduction can be done in any order, but we have shown previously [6] that all possible reduction sequences converge to the same normal form.

Example 1.5 (Reducing a Flow). The flow specification $\text{let } x = (\text{let } y = A ; B \text{ in } y \parallel y) \text{ in } x ; x$ is reducible, and we get $((A ; B) \parallel (A ; B)) ; ((A ; B) \parallel (A ; B))$ when we reach the normal form.

1.2 Previous TRAFFIC Type System

In this section, we introduce the type system that was previously used in [6] with a slightly different presentation. The type system in Figure 1.2 concerns typing judgment of the form $\Gamma \vdash \mathcal{A} : T$ with some environment Γ and flow type T . A type environment Γ is a partial function from FlowVar (the set of flow variables) to FlowType (the set of flow types). If Γ has a finite domain of definition, we write it as

$$\Gamma = \{x_1 : T_1, \dots, x_n : T_n\}$$

where $n \geq 0$ for some distinct flow variables x_1, \dots, x_n and some flow types T_1, \dots, T_n . A typing judgment $\Gamma \vdash \mathcal{A} : T$ is read as “under the hypotheses that x_1 has flow type T_1 and so on, the flow \mathcal{A} has the flow type T .”

A flow type (see Figure 1.3) is written as a two by two matrix $\begin{bmatrix} \tau_1 & \tau_2 \\ \tau_3 & \tau_4 \end{bmatrix}$, where τ_1 denotes the type for forward input, τ_2 for forward output, τ_3 for backward input, and τ_4 for backward output¹. A type τ can be a pair of types, written $(\tau_1 \cdot \tau_2)$, which is used to represent types of a parallel flow where two connections are bundled but not connected; or it can be a type literal t (also called “socket type” in [1]).

¹In the previous paper [6], we distinguish forward and backward types and prohibit subtyping relation to span across the two sets of types, but this distinction is removed here.

$$\begin{array}{l}
\text{(VAR)} \quad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(LET)} \quad \frac{\Gamma \vdash \mathcal{A} : T \quad \Gamma \cup \{x : T\} \vdash \mathcal{B} : T'}{\Gamma \vdash \text{let } x = \mathcal{A} \text{ in } \mathcal{B} : T'} \\
\text{(LOCAL)} \quad \frac{\text{type}(A) = T}{\Gamma \vdash A : T} \qquad \text{(PAR)} \quad \frac{\Gamma \vdash \mathcal{A} : [\begin{smallmatrix} \tau_1 & \tau_2 \\ \tau_3 & \tau_4 \end{smallmatrix}] \quad \Gamma \vdash \mathcal{B} : [\begin{smallmatrix} \tau_5 & \tau_6 \\ \tau_7 & \tau_8 \end{smallmatrix}]}{\Gamma \vdash \mathcal{A} \parallel \mathcal{B} : [\begin{smallmatrix} \tau_1, \tau_5 & \tau_2, \tau_6 \\ \tau_3, \tau_7 & \tau_4, \tau_8 \end{smallmatrix}]} \\
\text{(SEQ)} \quad \frac{\Gamma \vdash \mathcal{A} : [\begin{smallmatrix} \tau_1 & \tau_2 \\ \tau_3 & \tau_4 \end{smallmatrix}] \quad \Gamma \vdash \mathcal{B} : [\begin{smallmatrix} \tau_5 & \tau_6 \\ \tau_7 & \tau_8 \end{smallmatrix}] \quad \Delta \Vdash \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}}{\Gamma \vdash \mathcal{A}; \mathcal{B} : [\begin{smallmatrix} \tau_1 & \tau_6 \\ \tau_3 & \tau_8 \end{smallmatrix}]} \\
\text{(SUB)} \quad \frac{\Gamma \vdash \mathcal{A} : T \quad \Delta \Vdash T <: T'}{\Gamma \vdash \mathcal{A} : T'}
\end{array}$$

Figure 1.2: Typing rules for TRAFFIC.

$$\begin{array}{l}
t \in \text{TypeLit} \\
\tau \in \text{Type} \quad ::= \quad t \mid (\tau_1 \cdot \tau_2) \\
T \in \text{FlowType} \quad ::= \quad [\begin{smallmatrix} \tau_1 & \tau_2 \\ \tau_3 & \tau_4 \end{smallmatrix}]
\end{array}$$

Figure 1.3: Syntax of types for TRAFFIC.

Actual designation of the set of type literals depends on what kind of interfacing properties are to be checked. We abstract the notion of compatibility check by a subtyping assumption Δ , which is a partial order relation on type literals. The set of type literals and the set of subtyping relation over type literals together form the customizable aspect of the type system.

Assumption 1.6 (Subtyping Assumptions). *Let $\Delta \subseteq \text{TypeLit} \times \text{TypeLit}$ be a possibly infinite, arbitrary but fixed partial order relation for subtyping (Δ must be reflexive, transitive, and antisymmetric), each written as $t_1 <: t_2$ for some type literals t_1 and t_2 .*

Subtyping relation can be unambiguously lifted to `Type` and `FlowType`. We write $\Delta \Vdash X_1 <: X_2$ to mean that under the assumptions of Δ , it is the case that X_1 is a subtype of X_2 . Both X_1 and X_2 must be the same kind that is one of `TypeLit`, `Type`, and `FlowType`. Furthermore, $\Delta \Vdash X_1 \doteq X_2$ is a shorthand for $\Delta \Vdash \{X_1 <: X_2, X_2 <: X_1\}$ due to antisymmetry property of Δ .

Example 1.7 (Subtyping Relations). The following is a partial list of possible ways to describe interfacing property and compatibility.

Bandwidth Let the set of type literals be real numbers and subtyping relations be the binary predicate $a \leq b$ for a is less than or equal to b . The real numbers represent bandwidth in some measurement per second. An output can be connected to an input if the bandwidth produced by the output is less than what the input can handle.

Bandwidth Jitter Instead of a constant bandwidth requirement, we could instead specify both an upper bound and lower bound of bandwidth as a range where $(lo_1, hi_1) <: (lo_2, hi_2)$ iff $lo_1 \geq lo_2$ and $hi_1 \leq hi_2$.

Security Levels Let the set of type literals be \mathbb{Z}_3 and subtyping relations be the binary predicate \leq . The integers represent security clearance levels where 0 is the lowest and 3 the highest. An output may only be connected to an input with a higher or the same clearance level.

A more complicated security model can be achieved by letting type literals be some symbols representing security entities where the subtyping relations are manually specified. For example, suppose entity a is cleared by both b and c , both entities b and c are cleared by entity d , but neither b nor c clear each other. We can write Δ as $\{a <: b, a <: c, b <: d, c <: d\}$.

Capabilities Suppose we have a list of symbols that denote qualitative capabilities. Let each type literal be a set that may contain any of these symbols and subtyping relations be the binary predicate $A \subseteq B$ for A is a subset of B or is equal to B . Capabilities for an instant messaging application may include `text-chat`, `audio-chat`, `video-chat`, `file-transfer`, and `encryption`.

r, s, t	\in	TypeLit	
α, β	\in	TypeVar	
F	\in	TypeCons	
ρ, σ, τ	\in	Type	$::= t \mid \alpha \mid \tau_1 \cdot \tau_2 \mid F \tau_1 \dots \tau_n$
P	\in	Predicate	
c, d	\in	Constraint	$::= \tau_1 < \tau_2 \mid \tau_1 \doteq \tau_2 \mid P \tau_1 \dots \tau_n \mid true \mid false$ $\mid T_1 < T_2 \mid T_1 \doteq T_2 \mid P T_1 \dots T_n$
C, D, E	\in	ConstraintSet	$::= \{c_1, \dots, c_n\}$
T	\in	FlowType	$::= \begin{bmatrix} \tau_1 & \tau_2 \\ \tau_3 & \tau_4 \end{bmatrix}$
S	\in	Scheme	$::= T \mid \forall \bar{\alpha}(C).T$

Figure 2.1: Syntax of Types

Our implementation for TRAFFIC provides built-in subtyping relation on numerics, strings, and sets, which may be used to build tuples whose subtyping relation is lifted to the conjunction of subtyping relations of tuple elements.

We also provide a special operator to reverse a subtyping relation. For example, subtyping relation of a numeric range is expressed as $(!Numeric, Numeric)$, read as “a pair of numerics whose first element is checked against reversed subtyping relation (\geq) and second element against the usual subtyping relation (\leq).” Therefore, the range $(1.2, 5.4)$ is a subtype of $(0.9, 5.6)$ because $1.2 \geq 0.9$ and $5.4 \leq 5.6$.

2 The Type System TRAFFIC(\mathbf{X})

Oftentimes it is desired that input and output types of some flow be related in terms of a parameter. An example is a content delivery network where the same bandwidth passes through from the input to the output. Furthermore, we may want to impose a bandwidth cap. Our previous system lacks the ability to describe such scenario. This is a consequence of Theorem 3.13.

Having a type system that is more descriptive and flexible is the incentive of augmenting the type system with polymorphic and constrained flow types. The resulting type system is called TRAFFIC(\mathbf{X}), where \mathbf{X} denotes an arbitrary choice of constraint system to be used. Some examples can be found in Section 4.

2.1 Syntax of Types

We extend the syntax of types in Figure 1.3 to include type variables. In Type, in addition to the type constructor \cdot that existed before, we allow customized type constructors. We use F to range over type constructors of any arity. Similarly, in addition to the subtyping constraint $<$: and equality constraint \doteq , we allow customized constraint predicates. We use P to range over constraint predicates of any arity. Furthermore, we have the nullary predicates *true* and *false*, where *true* may be neglected (a constraint set that contains only *true* is equivalent to the empty constraint set), and *false* signals that a constraint set is not solvable.

The environment Γ is extended to be

$$\Gamma = \{x_1 : S_1, \dots, x_n : S_n\}$$

We proceed to define common functions and operators on types.

Substitution on Types Given a type τ' , substitution replaces free occurrences of type variables in τ' for some types. Suppose we are given a vector of type variables $\bar{\alpha} = (\alpha_1, \dots, \alpha_n)$ and a vector of types $\bar{\tau} = (\tau_1, \dots, \tau_n)$, the notion that τ_i is substituted for occurrences of α_i in τ' for $1 \leq i \leq n$ is written as $[\bar{\alpha} := \bar{\tau}]\tau'$. This operation can also be applied to constraints, flow types, flow type schemes, and environments. We use the letter ψ to denote substitutions when we don't care about the specifics of substitution.

Free Type Variables We write $\text{ftv}(X_1, \dots, X_n)$ to extract free type variables in objects X_1 through X_n , and these objects can be a mix of types, constraints, flow types, flow type schemes and environments, which are described as follows. Given a type τ , its free type variable occurrences are all the type variables that occur in τ . The same applies to constraints and flow types. For a flow type scheme $\forall \bar{\alpha}(D).T$, its free type variables are $\text{ftv}(D, T) - \bar{\alpha}$, since the variables quantified by $\bar{\alpha}$ are no longer free. Free type variables in an environment Γ is a collection of all free type variables in its bindings.

2.2 Constraints

We introduce the notion of *constraint entailment*, which is a binary relation, denoted \Vdash , on constraint sets. The typing system in Figure 2.2 is parameterized with respect to \Vdash , i.e., there is such a typing system for each instance of the relation \Vdash .

If we write $C \Vdash D$ for some constraint sets C and D , then we say “ C entails D .” For brevity, we write $C \Vdash d$ (constraint set C entails a single constraint d) as a shorthand for $C \Vdash \{d\}$. There are different ways to define \Vdash , some of which are considered in Section 4. However, regardless of its definition, we require that \Vdash satisfies certain properties.

Constraint entailment has some properties similar to sequent: (i) below corresponds to weakening, (iii) corresponds to cut, (iv) corresponds to invocation of hypotheses, (v) corresponds to conjunction, and (vi) corresponds to projection.

Assumption 2.1 (Properties of Constraint Entailment). *Given constraint sets C and D , any constraint system that models our notion of constraint entailment $C \Vdash D$ must satisfy these properties:*

$$\begin{array}{lll} \text{(i)} \frac{C \Vdash D}{C \cup C' \Vdash D} & \text{(ii)} \frac{C \Vdash D}{\psi C \Vdash \psi D} & \text{(iii)} \frac{C \Vdash D \quad C \cup D \Vdash E}{C \Vdash E} \\ \text{(iv)} \frac{D \subseteq C}{C \Vdash D} & \text{(v)} \frac{C \Vdash D \quad C \Vdash E}{C \Vdash D \cup E} & \text{(vi)} \frac{C \Vdash D_1 \cup D_2}{C \Vdash D_i} \quad i = 1, 2 \end{array}$$

Definition 2.2 (Constraint Consistency). We define a unary relation Φ on a set of constraint C , written $\Phi[C]$, as a shorthand for $C \not\Vdash \text{false}$.

Remark 2.3 (Properties of Consistent Constraints). According to the constraint entailment rules above, we have the following properties for Φ . If it holds that $\Phi[C]$, then:

1. For every $C' \subseteq C$, we have $\Phi[C']$. *Proof.* Suppose $C' \Vdash \text{false}$, then by (i), $C \Vdash \text{false}$, contradiction.
2. If $C = \psi D$ for some ψ , then $\Phi[D]$. *Proof.* Suppose $D \Vdash \text{false}$, then by (ii), $\psi D \Vdash \text{false}$, contradiction.
3. If $C \Vdash D$, then $\Phi[C \cup D]$. *Proof.* Suppose $C \cup D \Vdash \text{false}$ and $C \Vdash D$, then by (iii), $C \Vdash \text{false}$, contradiction.

Furthermore, $\Phi[\emptyset]$ holds, since \emptyset is a subset of any consistent constraint set.

2.3 Typing Rules

The type system TRAFFIC(\mathbf{X}) consists of rules in Figure 2.2.

Notation 2.4 (Disjoint Sets). We write $X \# Y$ to denote that sets X and Y are disjoint, i.e. $X \cap Y = \emptyset$.

Assumption 2.5 (Local Flows have Closed Type). *For every local flow $A \in \text{dom}(\text{type})$, we have $\text{ftv}(\text{type}(A)) = \emptyset$, i.e., $\text{type}(A)$ is closed.*

Lemma 2.6 (Consistency Condition in Typing). *Given a derivation $C, \Gamma \vdash A : S$,*

- *If $S = T$, then $\Phi[C]$.*
- *If $S = \forall \bar{\alpha}(D).T$, then $\Phi[C]$ and $C \supseteq D$.*

Proof. This holds for cases (VAR-ADD) and (LOCAL-ADD) because $\Phi[C]$ is given as a premise and $C \supseteq D$ is inherent from the resulting judgment. For the case (\forall -INTRO), use induction hypothesis to show $\Phi[C]$, then $C \supseteq D$ is inherent from the resulting judgment. All other cases are simply shown by induction hypothesis. \square

$$\begin{array}{c}
\text{(VAR-ADD)} \quad \frac{\Gamma(x) = \forall \bar{\alpha}(D).T \quad \Phi[C \cup D]}{C \cup D, \Gamma \vdash x : \forall \bar{\alpha}(D).T} \quad \text{(LET)} \quad \frac{C, \Gamma \vdash \mathcal{A} : S \quad C, \Gamma \cup \{x : S\} \vdash \mathcal{B} : T'}{C, \Gamma \vdash \text{let } x = \mathcal{A} \text{ in } \mathcal{B} : T'} \\
\text{(LOCAL-ADD)} \quad \frac{\text{type}(A) = \forall \bar{\alpha}(D).T \quad \Phi[C \cup D]}{C \cup D, \Gamma \vdash A : \forall \bar{\alpha}(D).T} \quad \text{(PAR)} \quad \frac{C, \Gamma \vdash \mathcal{A} : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}] \quad C, \Gamma \vdash \mathcal{B} : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}]}{C, \Gamma \vdash \mathcal{A} \parallel \mathcal{B} : [\frac{\tau_1 \cdot \tau_5 \ \tau_2 \cdot \tau_6}{\tau_3 \cdot \tau_7 \ \tau_4 \cdot \tau_8}]} \\
\text{(SEQ)} \quad \frac{C, \Gamma \vdash \mathcal{A} : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}] \quad C, \Gamma \vdash \mathcal{B} : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}] \quad C \Vdash \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}}{C, \Gamma \vdash \mathcal{A}; \mathcal{B} : [\frac{\tau_1 \ \tau_6}{\tau_3 \ \tau_8}]} \\
\text{(\forall-INTRO)} \quad \frac{C \cup D, \Gamma \vdash \mathcal{A} : T \quad \bar{\alpha} \# \text{ftv}(C, \Gamma)}{C \cup D, \Gamma \vdash \mathcal{A} : \forall \bar{\alpha}(D).T} \quad \text{(SUB)} \quad \frac{C, \Gamma \vdash \mathcal{A} : T \quad C \Vdash T <: T'}{C, \Gamma \vdash \mathcal{A} : T'} \\
\text{(\forall-ELIM)} \quad \frac{C, \Gamma \vdash \mathcal{A} : \forall \bar{\alpha}(D).T \quad C \Vdash \psi D \quad \text{where } \text{dom}(\psi) = \bar{\alpha}.}{C, \Gamma \vdash \mathcal{A} : \psi T}
\end{array}$$

Figure 2.2: Typing rules of TRAFFIC(X).

Our type system is almost an exact replica of HM(X) described in [9] with some exceptions. The rule (SEQ) can be seen as a special case of (APP), for being analogous to composing two functions. Besides rules (SEQ) and (PAR) that are specific for flow specifications and the absence of (ABS) and (APP), we modified the system to ensure that if a typing judgment is derivable, then the constraints in the typing are consistent.

Four possible designs of the (\forall -INTRO) rule are discussed in [9], and we chose the fourth kind, which is the best alternative in the absence of existential constraints. In our system, all constraints with free type variables are implicitly existential. We also modified the rules (VAR) and (LOCAL) to mirror this design choice by making an unquantified copy of constraints in a flow type scheme in the left hand side of a typing. This allows us to preserve the fact that all constraints in a typing are consistent when we transform the typing derivation between TRAFFIC(X) and its syntax-directed system.

3 Correctness Results

For the purpose of constructing a type checking routine and for designing a type inference algorithm, we introduce an alternate, syntax-directed set of typing rules for TRAFFIC(X) as shown in Figure 3.1. We will show that these rules are equivalent to the earlier, non-syntax directed rules.

Syntax-directedness is achieved by eliminating (\forall -INTRO), (\forall -ELIM) and (SUB) rules. The idea is that (\forall -ELIM) and (SUB) can be lifted to the leaves of a derivation tree, i.e., to (VAR-ADD) and (LOCAL-ADD), and that (\forall -INTRO) can be dropped to be bottom of a derivation but above (LET). Therefore, (VAR-ADD) and (LOCAL-ADD) are modified to instantiate type variables and perform subtyping of flow types, and (LET) is modified to generalize type variables.

We use the following function to deterministically compute the most general type that can be obtained for (LET).

Definition 3.1 (Deterministic Generalization). A deterministic way to accomplish (\forall -INTRO) rule can be expressed in terms of a function $\text{gen}(C, \Gamma, T)$, defined as follows:

$$\begin{aligned}
\text{gen}(C, \Gamma, T) &= \forall \bar{\alpha}(D).T \\
\text{where } \bar{\alpha} &= \text{ftv}(C, T) - \text{ftv}(\Gamma) \\
\text{and } D &= \{c \in C \mid \bar{\alpha} \cap \text{ftv}(c) \neq \emptyset\}
\end{aligned}$$

Lemma 3.2. For every C, Γ and T , if $\text{gen}(C, \Gamma, T) = \forall \bar{\alpha}(D).T$, then $\bar{\alpha} \# \text{ftv}(C - D, \Gamma)$.

Proof. Immediately follows from definition of $\text{gen}(C, \Gamma, T)$. □

Notation 3.3. Let \mathcal{D} and \mathcal{E} range over typing derivations.

$$\begin{array}{l}
\text{(VAR-ADD-INST-SUB)} \quad \frac{\Gamma(x) = \forall \bar{\alpha}(D).T \quad \Phi[C \cup D \cup \psi D \cup \{\psi T <: T'\}]}{C \cup D \cup \psi D \cup \{\psi T <: T'\}, \Gamma \vdash x : T'} \\
\text{(LOCAL-ADD-INST-SUB)} \quad \frac{\text{type}(A) = \forall \bar{\alpha}(D).T \quad \Phi[C \cup D \cup \psi D \cup \{\psi T <: T'\}]}{C \cup D \cup \psi D \cup \{\psi T <: T'\}, \Gamma \vdash A : T'} \\
\text{(SEQ-ADD)} \quad \frac{C_1, \Gamma \vdash \mathcal{A}_1 : [\frac{\tau_1}{\tau_3} \frac{\tau_2}{\tau_4}] \quad C_2, \Gamma \vdash \mathcal{A}_2 : [\frac{\tau_5}{\tau_7} \frac{\tau_6}{\tau_8}] \quad \Phi[C_1 \cup C_2 \cup \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}]}{C_1 \cup C_2 \cup \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}, \Gamma \vdash \mathcal{A}_1 ; \mathcal{A}_2 : [\frac{\tau_1}{\tau_3} \frac{\tau_6}{\tau_8}]} \\
\text{(PAR-ADD)} \quad \frac{C_1, \Gamma \vdash \mathcal{A}_1 : [\frac{\tau_1}{\tau_3} \frac{\tau_2}{\tau_4}] \quad C_2, \Gamma \vdash \mathcal{A}_2 : [\frac{\tau_5}{\tau_7} \frac{\tau_6}{\tau_8}] \quad \Phi[C_1 \cup C_2]}{C_1 \cup C_2, \Gamma \vdash \mathcal{A}_1 \parallel \mathcal{A}_2 : [\frac{\tau_1, \tau_5}{\tau_3, \tau_7} \frac{\tau_2, \tau_6}{\tau_4, \tau_8}]} \\
\text{(LET-GEN)} \quad \frac{C_1, \Gamma \vdash \mathcal{A} : T \quad C_2, \Gamma \cup \{x : S\} \vdash \mathcal{B} : T' \quad \Phi[C_1 \cup C_2]}{C_1 \cup C_2, \Gamma \vdash \text{let } x = \mathcal{A} \text{ in } \mathcal{B} : T'} \quad S = \text{gen}(C_1, \Gamma, T)
\end{array}$$

Note: let ψ be a substitution where $\text{dom}(\psi) = \bar{\alpha}$ throughout.

Figure 3.1: Alternate, syntax-directed typing rules for TRAFFIC(**X**).

$$\begin{array}{l}
\text{(VAR-ADD-INST)} \quad \frac{\Gamma(x) = \forall \bar{\alpha}(D).T \quad \Phi[C \cup D \cup \psi D]}{C \cup D \cup \psi D, \Gamma \vdash x : \psi T} \\
\text{(LOCAL-ADD-INST)} \quad \frac{\text{type}(A) = \forall \bar{\alpha}(D).T \quad \Phi[C \cup D \cup \psi D]}{C \cup D \cup \psi D, \Gamma \vdash A : \psi T}
\end{array}$$

Note: let ψ be a substitution where $\text{dom}(\psi) = \bar{\alpha}$ throughout.

Figure 3.2: Intermediate typing rules.

3.1 Equivalence

To assist in proving equivalence of TRAFFIC(**X**) and its syntax-directed typing judgments, we introduce two intermediate typing rules in Figure 3.2 which we use to construct an intermediate derivation system. The main result we are interested in is that the original, non syntax-directed system in Figure 2.2 is sound and complete with respect to the syntax-directed system in Figure 3.1.

Definition 3.4. The intermediate typing derivation systems are referred to by the judgments \vdash^1 , \vdash^2 , and \vdash^3 , where the rules are specified as follows:

- \vdash^1 is defined to be TRAFFIC(**X**), whose rules are specified in Figure 2.2. Judgments have the form $C, \Gamma \vdash^1 \mathcal{A} : S$.
- \vdash^2 has rules (VAR-ADD-INST) and (LOCAL-ADD-INST) in Figure 3.2; (SUB) in Figure 2.2; and (SEQ-ADD), (PAR-ADD), (LET-GEN) in Figure 3.1. This system eliminates (\forall -INTRO) and (\forall -ELIM), and judgments have the form $C, \Gamma \vdash^2 \mathcal{A} : T$.
- \vdash^3 is defined to be the syntax-directed rules of TRAFFIC(**X**) as specified in Figure 3.1. This system effectively eliminates (SUB), and judgments also have the form $C, \Gamma \vdash^3 \mathcal{A} : T$.

We proceed to show the equivalence of all four systems by showing the soundness of \vdash^1 to \vdash^2 , \vdash^2 to \vdash^3 , and finally \vdash^3 to \vdash^1 . The proofs are found in Appendix A.

It is useful to keep in mind that \vdash^3 greedily collects all entailed constraints and make them appear in the typing. These constraints may not appear in the typing of \vdash^1 but are entailed by the constraints in the typing. Therefore, the following soundness results are shown with respect to a set of constraints in the typing of \vdash^i that entails constraints in the typing of \vdash^j with $i < j$.

Lemma 3.5 (Soundness of \vdash^1 with regard to \vdash^2). *Given a derivation \mathcal{D} of the judgment $C, \Gamma \vdash^1 \mathcal{A} : S$,*

1. If $S = T$, then there is a derivation \mathcal{D}' of the judgment $C \cup E, \Gamma \vdash^2 \mathcal{A} : T$ for some E such that $C \Vdash E$.
2. If $S = \forall \bar{\alpha}(D).T$ and given a substitution ψ where $\text{dom}(\psi) = \bar{\alpha}$ such that $C \Vdash \psi D$, then there is a derivation \mathcal{D}' of the judgment $C \cup \psi D, \Gamma \vdash^2 \mathcal{A} : \psi T$.

Lemma 3.6 (Soundness of \vdash^2 with regard to \vdash^3). *If there is a derivation \mathcal{D} of the judgment $C, \Gamma \vdash^2 \mathcal{A} : T$, then there is a derivation \mathcal{D}' of the judgment $C \cup D, \Gamma \vdash^3 \mathcal{A} : T$ for some D such that $C \Vdash D$.*

Theorem 3.7 (Soundness of \vdash^1 with regard to \vdash^3). *Given a derivation \mathcal{D} of the judgment $C, \Gamma \vdash^1 \mathcal{A} : S$,*

- *If $S = T$, then there is a derivation \mathcal{D}' of the judgment $C \cup D, \Gamma \vdash^3 \mathcal{A} : T$ for some D such that $C \Vdash D$.*
- *If $S = \forall \bar{\alpha}(D).T$ and given a substitution ψ where $\text{dom}(\psi) = \bar{\alpha}$ such that $C \Vdash \psi D$, then there is a derivation \mathcal{D}' of the judgment $C \cup \psi D \cup E, \Gamma \vdash^3 \mathcal{A} : \psi T$ for some E such that $C \cup \psi D \Vdash E$.*

Proof. Apply Lemma 3.5 and 3.6. □

Theorem 3.8 (Soundness of \vdash^3 with regard to \vdash^1). *If there is a derivation \mathcal{D} of the judgment $C \cup E, \Gamma \vdash^3 \mathcal{A} : T$ such that $C \Vdash E$ for some constraint sets C and E , then there is a derivation \mathcal{D}' of the judgment $C, \Gamma \vdash^1 \mathcal{A} : T$.*

3.2 Type Preservation

Theorem 3.9 (Type Preservation of \vdash^3). *If there is a derivation \mathcal{D} for the judgment $C, \Gamma \vdash^3 \mathcal{A} : T$ and $\mathcal{A} \rightarrow \mathcal{A}'$ for some \mathcal{A}' according to any of the reduction rules in Definition 1.3, then there is a derivation \mathcal{D}' for the judgment $C, \Gamma \vdash^3 \mathcal{A}' : T$.*

Theorem 3.10 (Type Preservation of \vdash^1). *Let \mathcal{D} be a derivation for the judgment $C, \Gamma \vdash^1 \mathcal{A} : S$ and $\mathcal{A} \rightarrow \mathcal{A}'$ for some \mathcal{A}' according to any of the reduction rules in Definition 1.3. Then there is a derivation \mathcal{D}' for the judgment $C, \Gamma \vdash^1 \mathcal{A}' : S$.*

3.3 Relation to TRAFFIC

Definition 3.11. Let \vdash^0 be defined as TRAFFIC, whose rules are specified in Figure 1.2. Judgments have the form $\Gamma \vdash^0 \mathcal{A} : T$.

Theorem 3.12 (TRAFFIC is sound with respect to TRAFFIC(\mathbf{X})). *If there is a derivation \mathcal{D} for the judgment $\Gamma \vdash^0 \mathcal{A} : T$, then there is a derivation \mathcal{D}' for the judgment $\emptyset, \Gamma \vdash^1 \mathcal{A} : T$.*

Theorem 3.13 (TRAFFIC is incomplete with respect to TRAFFIC(\mathbf{X})). *There is a typing judgment $C, \Gamma \vdash^1 \mathcal{A} : T$ where \mathcal{A} is not typable using \vdash^0 .*

Proof. Suppose x is a flow variable that denotes identity flow in TRAFFIC. Let Δ be linear ordering of integers. For brevity, we write a flow type in place of a local flow that has that particular flow type. Consider the flow $[\frac{1}{0} \frac{2}{0}]; x; [\frac{3}{0} \frac{4}{0}]; x; [\frac{5}{0} \frac{6}{0}]; x; [\frac{7}{0} \frac{8}{0}]$. In order for this flow to type check, possible flow type assignments for x have the form $[\frac{\rho_1}{0} \frac{\rho_2}{0}]$ where $\rho_1 \geq 6$ and $\rho_2 \leq 3$. However, this implies the flow $[\frac{5}{0} \frac{6}{0}]; x; [\frac{3}{0} \frac{4}{0}]$ also type checks in TRAFFIC. If x is the identity, we would have $6 \leq 3$, which is a contradiction with subtyping assumptions, therefore x cannot be the identity.

In TRAFFIC(\mathbf{X}), an identity flow has the type $\forall \bar{\alpha}. [\frac{\alpha_1}{\alpha_2} \frac{\alpha_1}{\alpha_2}]$, and the first flow correctly type checks with this assignment, while the second flow is correctly rejected. □

4 Implementing Constraint Entailment

The type system TRAFFIC(\mathbf{X}) still lacks a missing piece \mathbf{X} —a customizable part that specifies TypeLit, TypeCons, Predicate, and the relation \Vdash , which must relate constraints over at least equality (\doteq) and subtyping ($<$) predicates. The relation must also satisfy Assumption 2.1. In this section, we explore several ways to define \mathbf{X} .

We require all constraint systems discussed in this section to have the following properties:

$$\frac{C \Vdash P \tau_{1,i} \dots \tau_{m,i} \quad \text{for } 1 \leq i \leq n}{C \Vdash P (F \tau_{1,1} \dots \tau_{1,n}) \dots (F \tau_{m,1} \dots \tau_{m,n})} \quad \frac{C \Vdash P (F \tau_{1,1} \dots \tau_{1,n}) \dots (F \tau_{m,1} \dots \tau_{m,n})}{C \Vdash P \tau_{1,i} \dots \tau_{m,i}} \quad \text{for } 1 \leq i \leq n$$

where P is an m -ary predicate, and F is an n -ary type constructor. A special case for this is subtyping with parallel flows.

$$\frac{C \Vdash \tau_1 <: \tau'_1 \quad C \Vdash \tau_2 <: \tau'_2}{C \Vdash (\tau_1 \cdot \tau_2) <: (\tau'_1 \cdot \tau'_2)}$$

The idea is that when a constraint predicate P is made out of types sharing a common type constructor F , we can lift the predicate by stripping F and check for the predicate against sub-expressions of types. This is required so we can solve constraints involving parallel flows.

4.1 Simple Constraint System with Subtyping

Given a \mathbf{Y} that defines TypeLit and a set of subtyping assumption Δ as described in Assumption 1.6, let SUB(\mathbf{Y}) define a constraint system such that both TypeCons and Predicate are empty sets, and the relation \Vdash is inferred by the rules listed in Assumption 2.1 in addition to the ones below:

$$\frac{\{t_1 <: t_2\} \subseteq \Delta}{\emptyset \Vdash t_1 <: t_2} \quad \frac{\{t_1 <: t_2\} \not\subseteq \Delta}{t_1 <: t_2 \Vdash \text{false}} \quad \frac{C \Vdash \tau_1 <: \tau_2 \quad C \Vdash \tau_2 <: \tau_1}{C \Vdash \tau_1 \doteq \tau_2} \quad \frac{\tau_1 \neq \tau_2}{\tau_1 <: \tau_2, \tau_2 <: \tau_1 \Vdash \text{false}}$$

Hence, TRAFFIC(SUB(\mathbf{Y})) denotes a complete type system with polymorphism and subtyping constraints. Some examples of \mathbf{Y} can be found in Example 1.7. An example using TRAFFIC(SUB(\mathbf{Y})) with polymorphism and subtyping can be found in Section 5.1.

4.2 Constraint System with Linear Programming

Let TypeLit range over real numbers, and we introduce three binary type constructors $+$, $-$ and $*$. The resulting Type allows us to construct linear polynomials whose variables range over TypeVar, but also non-linear polynomials among other non-sensical types. The meaning of subtyping ($<:$) is treated as if it were a less-than-or-equal-to ($\leq_{\mathbb{R}}$), and equality (\doteq) the same as algebraic equality ($=_{\mathbb{R}}$).

We are only interested to see if a set of constraints have a feasible solution, which can be determined using the initialization routine of the Simplex algorithm [2, pp. 812]. Special caution must be observed that variables of a feasible solution must have non-negative values.

A *feasible region* of a set of constraints is the space enclosed by all the inequalities in that set. We write $C \Vdash D$ to denote that the feasible region defined by constraints in C lies entirely inside the feasible region defined by constraints in D . If C is infeasible, then $C \Vdash \text{false}$.

To decide if $C \Vdash D$, we consider the base case $C \Vdash d$ supposing there is only one constraint d in D . Let I be a set of points that constraints in C intersect each other, and J be a set of points that constraints in C intersect d . If there is a point in $J - I$ that lies inside the feasible region of C , then we know d “cuts off” a part of C , so $C \not\Vdash d$.²

An example using this constraint system can be found in Section 5.2.

4.3 Constraint Handling Rules

So far, we showed two examples of constraint entailment that can be used with the TRAFFIC(\mathbf{X}) framework. There are many more possible implementations of constraint entailment. An attempt to generalize programming of constraint entailment results in a language that expresses constraint logic programs. Such treatment on HM(\mathbf{X}) has already been studied to provide dimension types and overloading [4].

A constraint logic program (CLP) is a collection of constraint handling rules (CHR). We use \mathbb{P} to denote a CLP, which is a customizable aspect of the type system TRAFFIC(CHR $_{\mathbb{P}}$).

A rule has two parts, each consisting of a set of constraints: *head*, and *goal*—let them be denoted as C_h and C_b respectively. Rules are written in the following form:

$$\begin{aligned} C_h &\iff C_b \quad (\text{simplification}) \\ C_h &\implies C_b \quad (\text{propagation}) \end{aligned}$$

Given a set of constraints C , in order to determine whether a rule is applicable, we check to see if constraints in C_h appear in C subject to type substitution. We have two kinds of rules, *simplification* and *propagation*, and they differ in

²Thanks to Stan Sclaroff, Murat Erdem, and Alexandra Stefan for the brainstorming session to design an algorithm for this.

$\begin{array}{l} \alpha \doteq \alpha \iff true \\ \alpha_1 \doteq \alpha_2, \alpha_2 \doteq \alpha_3 \implies \alpha_1 \doteq \alpha_3 \\ \alpha_1 \doteq \alpha_2 \implies \alpha_2 \doteq \alpha_1 \\ (\alpha_1 \cdot \alpha_2) \doteq (\alpha'_1 \cdot \alpha'_2) \iff \alpha_1 \doteq \alpha'_1, \alpha_2 \doteq \alpha'_2 \\ \begin{bmatrix} \alpha_1 & \alpha_2 \\ \alpha_3 & \alpha_4 \end{bmatrix} \doteq \begin{bmatrix} \alpha'_5 & \alpha'_6 \\ \alpha'_7 & \alpha'_8 \end{bmatrix} \iff \alpha_5 \doteq \alpha_1, \alpha_2 \doteq \alpha_6, \\ \alpha_3 \doteq \alpha_7, \alpha_8 \doteq \alpha_4 \end{array}$	$\begin{array}{l} \alpha <: \alpha \iff true \\ \alpha_1 <: \alpha_2, \alpha_2 <: \alpha_3 \implies \alpha_1 <: \alpha_3 \\ \alpha_1 <: \alpha_2 \iff \alpha_1 \doteq \alpha_2 \\ (\alpha_1 \cdot \alpha_2) <: (\alpha'_1 \cdot \alpha'_2) \iff \alpha_1 <: \alpha'_1, \alpha_2 <: \alpha'_2 \\ \begin{bmatrix} \alpha_1 & \alpha_2 \\ \alpha_3 & \alpha_4 \end{bmatrix} <: \begin{bmatrix} \alpha'_5 & \alpha'_6 \\ \alpha'_7 & \alpha'_8 \end{bmatrix} \iff \alpha_5 <: \alpha_1, \alpha_2 <: \alpha_6, \\ \alpha_3 <: \alpha_7, \alpha_8 <: \alpha_4 \end{array}$
(a) Constraints for “ \doteq ”	(b) Constraints for “ $<:$ ”

Figure 4.1: Built-in CHRs.

the manner in which they manipulate the constraints in C . A *simplification* rule replaces constraints in C that appear in C_h with C_b , but a *propagation* rule simply adds the constraints in C_b to C . This notion is made precise by Definition 4.1.

Definition 4.1 (Operational Semantics of CHR). Given a CLP \mathbb{P} , a transition from a set of constraints to another, $C \mapsto_{\mathbb{P}} C'$, is carried out according to the following rules.

SOLVE	$\{\alpha \doteq \tau\} \cup C \mapsto [\alpha := \tau]C$
LABEL (\doteq)	$\{t_1 \doteq t_2\} \cup C \mapsto \{d\} \cup C$ If $\{t_1 <: t_2, t_2 <: t_1\} \subseteq \Delta$, then $d = true$, else $d = false$.
LABEL ($<:$)	$\{t_1 <: t_2\} \cup C \mapsto \{d\} \cup C$ If $\{t_1 <: t_2\} \in \Delta$, then $d = true$, else $d = false$.
SIMPLIFY	$C \mapsto (C - [\bar{\alpha} := \bar{\tau}]C_h) \cup [\bar{\alpha} := \bar{\tau}]C_b$ for rule $C_h \iff C_b$ in \mathbb{P} and there exists a substitution such that $[\bar{\alpha} := \bar{\tau}]C_h \subseteq C$.
PROPAGATE	$C \mapsto C \cup [\bar{\alpha} := \bar{\tau}]C_b$ for rule $C_h \implies C_b$ in \mathbb{P} and there exists a substitution such that $[\bar{\alpha} := \bar{\tau}]C_h \subseteq C$.

For the purpose of built-in type constructors (parallel composition and flow type) and built-in constraints (equality and subtyping), we always have the built-in CHRs listed in Figure 4.1.

Definition 4.2 (Reflexive, Transitive Closure). We write $\mapsto_{\mathbb{P}}^*$ to denote the reflexive, transitive closure of $\mapsto_{\mathbb{P}}$. That is, $C \mapsto_{\mathbb{P}}^* D$ iff C reduces to D in a finite number of steps, possibly zero.

Definition 4.3 (Constraint Entailment with CHR). We define $\Vdash_{\mathbb{P}}$ in the following sense: given a CLP \mathbb{P} , we have $C \Vdash_{\mathbb{P}} D$ iff $C \mapsto_{\mathbb{P}}^* D$.

4.4 Combining Constraint Systems

It may be useful to combine a number of constraint systems so several unrelated properties of a flow can be checked at once. This, for example, can be used to combine all of the applications described in Section 5.

Suppose we are given constraint systems $(\mathbf{Y}_1, \dots, \mathbf{Y}_n)$ that define TypeLit_i and the relation \Vdash_i for $1 \leq i \leq n$ respectively. We assume without loss of generality that TypeLit_i are disjoint for $1 \leq i \leq n$. Furthermore, we assume that all constraint systems define the same set of TypeCons and Predicate .

We define $\text{TRAFFIC}(\langle \mathbf{Y}_1, \dots, \mathbf{Y}_n \rangle)$ in the following way:

$$\begin{aligned} \text{TypeLit}' &= \prod_{1 \leq i \leq n} \text{TypeLit}_i \\ \text{TypeCons}' &= \text{TypeCons} \cup \{\langle \cdot \rangle_n\} \\ \text{Predicate}' &= \text{Predicate} \end{aligned}$$

where $\langle \cdot \rangle_n$ denotes an n -ary *tuple* constructor.

We now have individual constraint entailment relations \Vdash_i , one for each $1 \leq i \leq n$, as well as a combined constraint entailment relation \Vdash . The interaction between the individual relations \Vdash_i and the combined relation \Vdash is defined by the following rule:

$$\frac{C_i \Vdash_i P \tau_{1,i} \dots \tau_{m,i} \quad \text{for } 1 \leq i \leq n}{C \Vdash P \langle \tau_{1,1} \dots \tau_{1,n} \rangle \dots \langle \tau_{m,1} \dots \tau_{m,n} \rangle}$$

where C_i is obtained from C by performing i -th projection on types in C .

5 Applications

In this section, we demonstrate several ways to use TRAFFIC(\mathbf{X}). These examples provide a motivation to use this system over the previous one by using quantified type variables in some interesting way.

5.1 Lossy Compression

Imagine that we have a network flow of streaming video and audio. A stream is compressed before it is sent out on a delivery network, and decompressed after a client receives it. In this example, we will use the constraint system with simple subtyping.

We consider lossless compression for now. Here are some basic concepts for this system:

- After a compressed stream is decompressed, we get the original stream back. This is true for all types of streams.
- A *source* is a flow that generates a stream, and a *client* is a flow that consumes it. We have a source and a client for both video and audio streams.
- Content delivery network never alters a stream. We assume reliable network connection, so there is no packet loss.

The following local flow type assignments describe these concepts:

$$\begin{aligned}
 \text{type}(\textit{Compress}) &= \forall\alpha. \begin{bmatrix} \alpha & F_{\text{compressed}}(\alpha) \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} & \text{type}(\textit{Decompress}) &= \forall\alpha. \begin{bmatrix} F_{\text{compressed}}(\alpha) & \alpha \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} \\
 \text{type}(\textit{VideoSource}) &= \begin{bmatrix} F_{\text{nil}} & F_{\text{video}} \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} & \text{type}(\textit{VideoClient}) &= \begin{bmatrix} F_{\text{video}} & F_{\text{nil}} \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} \\
 \text{type}(\textit{AudioSource}) &= \begin{bmatrix} F_{\text{nil}} & F_{\text{audio}} \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} & \text{type}(\textit{AudioClient}) &= \begin{bmatrix} F_{\text{audio}} & F_{\text{nil}} \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} \\
 \text{type}(\textit{Delivery}) &= \forall\bar{\alpha}. \begin{bmatrix} \alpha_1 & \alpha_1 \\ \alpha_2 & \alpha_2 \end{bmatrix}
 \end{aligned}$$

So far, the local flows *Compress* and *Decompress* are for lossless compression and decompression. The following flow basically connects, over a delivery network, lossless compressed video and audio streaming source to a client capable of decompressing it. We can easily verify that it type checks.

$$\begin{aligned}
 \text{let } x &= (\textit{VideoSource} ; \textit{Compress}) \parallel (\textit{AudioSource} ; \textit{Compress}) \text{ in} \\
 \text{let } y &= (\textit{Decompress} ; \textit{VideoClient}) \parallel (\textit{Decompress} ; \textit{AudioClient}) \text{ in} \\
 x ; \textit{Delivery} ; y
 \end{aligned}$$

Most applications for streaming media use lossy compression and decompression, however. To make our example more interesting, we introduce the following new concepts:

- After decrypting an encrypted stream, we get the original stream back. This is true for all types of streams.
- Only video and audio streams may be lossy compressed. After decompressing a lossy compressed video stream, we get a video stream back. Same for an audio stream.
- Any other data streams (e.g., encrypted streams of any sort) cannot be lossy compressed.

We can use constraints on the type variable to express the restriction on lossy compression and decompression. Local flow type assignments that reflect these concepts are found below:

$$\begin{aligned}
 \text{type}(\textit{LossyCompress}) &= \forall\alpha(D). \begin{bmatrix} \alpha & F_{\text{lossycompressed}}(\alpha) \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} & \text{type}(\textit{Encrypt}) &= \forall\alpha. \begin{bmatrix} \alpha & F_{\text{encrypted}}(\alpha) \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} \\
 \text{type}(\textit{LossyDecompress}) &= \forall\alpha(D). \begin{bmatrix} F_{\text{lossycompressed}}(\alpha) & \alpha \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix} & \text{type}(\textit{Decrypt}) &= \forall\alpha. \begin{bmatrix} F_{\text{encrypted}}(\alpha) & \alpha \\ F_{\text{nil}} & F_{\text{nil}} \end{bmatrix}
 \end{aligned}$$

where $D = \{\alpha <: F_{\text{lossycompressable}}\}$. We also have the following subtyping assumptions:

$$\Delta = \{F_{\text{video}} <: F_{\text{lossycompressable}}, F_{\text{audio}} <: F_{\text{lossycompressable}}\}$$

Then obviously, the previous example replacing *Compress* with *LossyCompress* and *Decompress* with *LossyDecompress* still type checks. The following flow also type checks:

$$(VideoSource ; Encrypt ; Compress ; Decompress ; Decrypt ; VideoClient)$$

But not this flow:

$$(VideoSource ; Encrypt ; LossyCompress ; LossyDecompress ; Decrypt ; VideoClient)$$

Because according to Δ , the constraint $F_{\text{encrypted}(F_{\text{video}})} <: F_{\text{lossycompressable}}$ does not hold. This is because an encrypted video stream is not lossy compressible.

5.2 Computing Round Trip Time

In this section, we demonstrate how latency can be expressed in flow types, and how this information can be used to compute round trip time using constraint system with linear programming.

Suppose the following local flows have latencies that are asymmetric for forward and backward directions:

Local Flow	Forward Lat.	Backward Lat.
A_1	10 ms.	6 ms.
A_2	12 ms.	15 ms.
A_3	8 ms.	9 ms.

For each local flow X listed above, we have

$$\text{type}(X) = \forall x, y. \left[\begin{array}{cc} x & x+\delta_1 \\ y+\delta_2 & y \end{array} \right]$$

where δ_1 is the forward latency, and δ_2 the backward latency. Types describe the wall clock time a signal enters or leaves a flow, in milliseconds. The flow type is quantified over two variables, x , which denotes the time a signal enters the forward input, and y , which denotes the time a signal enters the backward input. Forward signal exits at time $x + \delta_1$, and backward signal at time $y + \delta_2$.

Note that we consider “+” as a binary type constructor. We use simplex algorithm for the constraint solver. All constraints are linear.

To compute round trip time of the flow $A_1 ; A_2 ; A_3$, we instantiate x and y variables to distinct variables and obtain the following flow type diagram:

$$\begin{array}{ccc} \left[\begin{array}{cc} x_1 & x_1+10 \\ y_1+6 & y_1 \end{array} \right] & - & \left[\begin{array}{cc} x_2 & x_2+12 \\ y_2+15 & y_2 \end{array} \right] & - & \left[\begin{array}{cc} x_3 & x_3+8 \\ y_3+9 & y_3 \end{array} \right] \\ A_1 & & A_2 & & A_3 \end{array}$$

This flow is a sequential flow, so (SEQ) rule is used to produce a typing of this flow. The rule requires us to solve the following constraints: $\{x_1 + 10 \doteq x_2, x_2 + 12 \doteq x_3, y_3 + 9 \doteq y_2, y_2 + 15 \doteq y_1\}$. After solving these constraints, the constraint solver effectively computes total forward latency and total backward latency. In order to relate the total forward and backward latency for round trip time, we need the additional constraint $\{x_3 + 8 \doteq y_3\}$. We also use type variables a and b to denote the start time and end time respectively, so $\{a \doteq x_1, b \doteq y_1 + 6\}$. These additional constraints can be added by prepending the local flow *Ping* and appending the local flow *Pong*, whose flow types are:

$$\text{type}(Ping) = \forall a, b(D). \left[\begin{array}{cc} F_{\text{nil}} & a \\ F_{\text{nil}} & b \end{array} \right] \quad \text{type}(Pong) = \forall x, y(x \doteq y). \left[\begin{array}{cc} x & F_{\text{nil}} \\ y & F_{\text{nil}} \end{array} \right]$$

Notice that we leave the constraint set D unspecified. For now, let it be the empty set.

Given all the constraints listed above, the constraint solver goes through the following iterations (not necessarily

in this exact order) to obtain:

$$\begin{array}{ll}
 b & = y_1 + 6 & [y_1 := y_2 + 15] \\
 & = y_2 + 15 + 6 = y_2 + 21 & [y_2 := y_3 + 9] \\
 & = y_3 + 9 + 21 = y_3 + 30 & [y_3 := x_3 + 8] \\
 & = x_3 + 8 + 30 = x_3 + 38 & [x_3 := x_2 + 12] \\
 & = x_2 + 12 + 38 = x_2 + 50 & [x_2 := x_1 + 10] \\
 & = x_1 + 10 + 50 = x_1 + 60 & [x_1 := a] \\
 & = a + 60 &
 \end{array}$$

At the end of constraint solving, we have the constraint $\{b \doteq a + 60\}$. This constraint expresses the exact relation between start time and end time. In this case, the round trip time is 60 milliseconds.

To say something to the effect of “restrict total round trip time to be less than or equal to 50 milliseconds,” let $D = \{b - a \leq 50\}$. Then, when we type check the flow $Ping ; A_1 ; A_2 ; A_3 ; Pong$, the constraint solver would indicate that $\{b \doteq a + 60, b - a \leq 50\}$ is not solvable.

Propagation of latency works similarly in a parallel flow setting if the timings of each flow are independent. If we have a flow that splits or a flow that joins two parallel flows, then the timings become dependent, and latencies of the split and join need to be disambiguated.

6 Conclusion

The main contributions of this paper is a framework for TRAFFIC with polymorphism and constraints. We showed the correctness of the framework, provided a few ways to customize the framework, and presented verification examples that use these customizations.

Related Work There are two approaches for formal system verification: syntactic and semantic. Syntactic approach means the verification algorithm is driven by syntax of input. Semantic approach means the verification algorithm is driven by a desired model. Our work uses a mixed approach. Analysis based on cases of flow composition is syntactic; however, implementation of constraint entailment is semantic.

It is also possible to take the semantic approach entirely: [5] encodes IPsec policies as a boolean formula and check for policy conflicts by trying to resolve a normal form of the formula. A boolean formula that reduces to false indicates a policy conflict.

Future Work In this paper, we presented a syntax-directed method for type-checking. However, it is desirable to produce a type inference algorithm for TRAFFIC with principal typing property like that in [7]. Such algorithm enjoys modular verification, and the results can be combined efficiently without having to go through detailed analysis. We also hope to build more applications that exercise this type system extensively.

References

- [1] 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.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, Cambridge, Mass., 2nd edition, 2001.
- [3] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.
- [4] K. Glynn, P.J. Stuckey, and M. Sulzmann. The hm(chr) framework. Technical report, Department of Computer Science, The University of Melbourne, 2001.
- [5] Hazem Hamed, Ehab Al-Shaer, and Will Marrero. Modeling and verification of ipsec and vpn security policies. *icnp*, 0:259–278, 2005.

- [6] 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.
- [7] Likai Liu, Assaf Kfoury, Azer Bestavros, Yarom Gabay, Adam Bradley, and Ibrahim Matta. Safe Compositional Specification of Networking Systems: A Compositional Analysis Approach. Technical Report BUCS-TR-2005-033, CS Department, Boston University, December 28 2005.
- [8] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [9] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

A Proofs

In this section, we revisit the theorems and lemmas cited earlier and complete them with proofs.

A.1 Equivalence

Lemma (3.5). *Given a derivation \mathcal{D} of the judgment $C, \Gamma \vdash^1 \mathcal{A} : S$,*

1. *If $S = T$, then there is a derivation \mathcal{D}' of the judgment $C \cup E, \Gamma \vdash^2 \mathcal{A} : T$ for some E such that $C \Vdash E$.*
2. *If $S = \forall \bar{\alpha}(D).T$ and given a substitution ψ where $\text{dom}(\psi) = \bar{\alpha}$ such that $C \Vdash \psi D$, then there is a derivation \mathcal{D}' of the judgment $C \cup \psi D, \Gamma \vdash^2 \mathcal{A} : \psi T$.*

Proof. By Lem 2.6, we have $\Phi[C]$. We proceed by induction on the structure of \mathcal{D} according to the rule that is used last. For those cases that require $C \Vdash \psi D$, we also have $\Phi[C \cup \psi D]$ because of Remark 2.3.

- case (VAR-ADD), where $S = \forall \bar{\alpha}(D).T$, $C = C_0 \cup D$, and \mathcal{D} has the form:

$$\text{(VAR-ADD)} \frac{\Gamma(x) = \forall \bar{\alpha}(D).T \quad \Phi[C_0 \cup D]}{C_0 \cup D, \Gamma \vdash^1 x : \forall \bar{\alpha}(D).T}$$

A substitution ψ is given so that $C \Vdash \psi D$. Then \mathcal{D}' has the form:

$$\text{(VAR-ADD-INST)} \frac{\Gamma(x) = \forall \bar{\alpha}(D).T \quad \Phi[C_0 \cup D \cup \psi D]}{C_0 \cup D \cup \psi D, \Gamma \vdash^2 x : \psi T}$$

- case (LOCAL-ADD) is similar to (VAR-ADD), resulting in a derivation \mathcal{D}' with the rule (LOCAL-ADD-INST).
- case (\forall -ELIM), \mathcal{D} has the form:

$$\text{(\forall-ELIM)} \frac{\mathcal{E} \frac{\vdots}{C, \Gamma \vdash^1 \mathcal{A} : \forall \bar{\alpha}(D).T_0} \quad C \Vdash \psi D}{C, \Gamma \vdash^1 \mathcal{A} : \psi T_0}$$

by induction hypothesis on \mathcal{E} , there is a derivation \mathcal{E}' of the judgment $C \cup \psi D, \Gamma \vdash^2 \mathcal{A} : \psi T_0$. Let $E = \psi D$, so $C \Vdash E$.

- case (\forall -INTRO) where $C = C_0 \cup D$, then \mathcal{D} has the form:

$$\text{(\forall-INTRO)} \frac{\mathcal{E} \frac{\vdots}{C_0 \cup D, \Gamma \vdash^1 \mathcal{A} : T} \quad \bar{\alpha} \# \text{ftv}(C_0, \Gamma)}{C_0 \cup D, \Gamma \vdash^1 \mathcal{A} : \forall \bar{\alpha}(D).T}$$

By substitution on \mathcal{E} , there is a derivation \mathcal{E}' such that $\psi(C_0 \cup D), \psi \Gamma \vdash^1 \mathcal{A} : \psi T_0$ is derivable. Note since $\text{dom}(\psi) = \bar{\alpha}$ and $\bar{\alpha} \# \text{ftv}(C_0, \Gamma)$, so $\psi(C_0 \cup D) = C_0 \cup \psi D$ and $\psi \Gamma = \Gamma$. Then let \mathcal{D}' be derivation \mathcal{E}' where the final judgment is $C_0 \cup \psi D, \Gamma \vdash^2 \mathcal{A} : \psi T_0$.

- case (LET). \mathcal{D} has the form:

$$\text{(LET)} \frac{\mathcal{E}_1 \frac{\vdots}{C, \Gamma \vdash^1 \mathcal{A} : S_0} \quad \mathcal{E}_2 \frac{\vdots}{C, \Gamma \cup \{x : S_0\} \vdash^1 \mathcal{B} : T}}{C, \Gamma \vdash^1 \text{let } x = \mathcal{A} \text{ in } \mathcal{B} : T}$$

Without loss of generality, we assume that $S_0 = \forall \bar{\alpha}(D).T_0$ (if S_0 is a flow type without quantifier, then we simply let $\bar{\alpha} = \varepsilon$ and constraint D be empty).

We use induction hypothesis on \mathcal{E}_1 in the following manner. Let ψ be the identity. By Lemma 2.6 on \mathcal{E}_1 , we have $\Phi[C]$ and $C \supseteq D$, and therefore $C \Vdash \psi D$ by Assumption 2.1 (iv). By induction hypothesis, there is a derivation \mathcal{E}'_1 of the judgment $C \cup D, \Gamma \vdash^2 \mathcal{A} : T_0$.

By induction hypothesis on \mathcal{E}_2 , there is a derivation \mathcal{E}'_2 of the judgment $C, \Gamma \cup \{x : S_0\} \vdash^2 \mathcal{B} : T$. Let $S'_0 = \forall \bar{\alpha}'(D').T_0 = \text{gen}(C \cup D, \Gamma, T_0)$. Because deterministic generalization yields a flow type scheme that is maximally quantified, there is a derivation \mathcal{E}''_2 of the judgment $C, \Gamma \cup \{x : S'_0\} \vdash^2 \mathcal{B} : T$.

Then let \mathcal{D}' be the following derivation:

$$\text{(LET)} \frac{\mathcal{E}'_1 \frac{\vdots}{C \cup D, \Gamma \vdash^2 \mathcal{A} : T_0} \quad \mathcal{E}''_2 \frac{\vdots}{C, \Gamma \cup \{x : S'_0\} \vdash^2 \mathcal{B} : T}}{C \cup D, \Gamma \vdash^2 \text{let } x = \mathcal{A} \text{ in } \mathcal{B} : T}$$

- case (SUB) is shown by straightforward application of induction hypothesis.
- case (SEQ), where $T = [\frac{\tau_1 \ \tau_6}{\tau_3 \ \tau_8}]$, and \mathcal{D} has the following form:

$$\text{(SEQ)} \frac{\mathcal{E}_1 \frac{\vdots}{C, \Gamma \vdash^1 \mathcal{A}_1 : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}]} \quad \mathcal{E}_2 \frac{\vdots}{C, \Gamma \vdash^1 \mathcal{A}_2 : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}]} \quad C \Vdash \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}}{C, \Gamma \vdash^1 \mathcal{A}_1 ; \mathcal{A}_2 : [\frac{\tau_1 \ \tau_6}{\tau_3 \ \tau_8}]}$$

By induction hypothesis on \mathcal{E}_1 and \mathcal{E}_2 , there are derivations \mathcal{E}'_1 and \mathcal{E}'_2 with the final judgments $C, \Gamma \vdash^2 \mathcal{A}_1 : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}]$ and $C, \Gamma \vdash^2 \mathcal{A}_2 : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}]$ respectively. Then let \mathcal{D}' be the following derivation:

$$\text{(SEQ-ADD)} \frac{\mathcal{E}'_1 \frac{\vdots}{C, \Gamma \vdash^2 \mathcal{A}_1 : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}]} \quad \mathcal{E}'_2 \frac{\vdots}{C, \Gamma \vdash^2 \mathcal{A}_2 : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}]} \quad \Phi[C \cup \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}]}{C \cup \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}, \Gamma \vdash^2 \mathcal{A}_1 ; \mathcal{A}_2 : [\frac{\tau_1 \ \tau_6}{\tau_3 \ \tau_8}]}$$

Let $E = \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}$. Since $C \Vdash E$, we have $\Phi[C \cup E]$ by Remark 2.3 (3).

- case (PAR), this case is similar to (SEQ), resulting in a derivation \mathcal{D}' whose last rule is (PAR-ADD), and that $E = \emptyset$.

□

Lemma A.1 (Typability preservation from \vdash^2 to \vdash^3). *If there is a derivation \mathcal{D} of the judgment $C, \Gamma \vdash^2 \mathcal{A} : T$ and given a T' such that $C \Vdash T <: T'$, then there is a derivation \mathcal{D}' of the judgment $C \cup \{T <: T'\}, \Gamma \vdash^3 \mathcal{A} : T'$.*

Proof. Induction on the structure of \mathcal{D} according to the rule that is used last.

- case (VAR-ADD-INST), then \mathcal{D} has the following form:

$$\text{(VAR-ADD-INST)} \frac{\Gamma(x) = \forall \bar{\alpha}(D).T_0 \quad \Phi[C_0 \cup D \cup \psi D]}{C_0 \cup D \cup \psi D, \Gamma \vdash^2 x : \psi T_0}$$

Then let \mathcal{D}' be the following derivation:

$$\text{(VAR-ADD-INST-SUB)} \frac{\Gamma(x) = \forall \bar{\alpha}(D).T_0 \quad \Phi[C_0 \cup D \cup \psi D \cup \{\psi T_0 <: T'\}]}{C_0 \cup D \cup \psi D \cup \{\psi T_0 <: T'\}, \Gamma \vdash^3 x : T'}$$

Note that $\Phi[C_0 \cup D \cup \psi D \cup \{\psi T_0 <: T'\}]$ by Remark 2.3 (3) because $C_0 \cup D \cup \psi D \Vdash \{\psi T_0 <: T'\}$.

- case (LOCAL-INST). This is similar to the previous case, generating a derivation with the last rule (LOCAL-INST-SUB) instead.
- case (SUB). Use induction hypothesis.
- all other cases are shown using straightforward application of induction hypothesis.

□

Lemma (3.6). *If there is a derivation \mathcal{D} of the judgment $C, \Gamma \vdash^2 \mathcal{A} : T$, then there is a derivation \mathcal{D}' of the judgment $C \cup D, \Gamma \vdash^3 \mathcal{A} : T$ for some D such that $C \Vdash D$.*

Proof. Use Lemma A.1 with $T = T'$ and $D = \{T <: T\}$. By reflexivity of $<:$, we have $\emptyset \Vdash T <: T$, so $C \Vdash T <: T$. □

Lemma A.2 (Admissibility of Adding Constraints). *If there is a derivation \mathcal{D} of the judgment $C, \Gamma \vdash^1 \mathcal{A} : S$, and given some C' such that $\Phi[C \cup C']$, then there is a derivation \mathcal{D}' of the judgment $C \cup C', \Gamma \vdash^1 \mathcal{A} : S$.*

Proof. Induction on the structure of \mathcal{D} according to the rule that is used last.

- base cases (VAR-ADD), (LOCAL-ADD) are trivial. Use $\Phi[C \cup C']$ in the premise.
- case (\forall -INTRO), where $C = C_0 \cup D$, and \mathcal{D} has the form:

$$(\forall\text{-INTRO}) \frac{\mathcal{E} \frac{\vdots}{C_0 \cup D, \Gamma \vdash^1 \mathcal{A} : T} \quad \bar{\alpha} \# \text{ftv}(C_0, \Gamma)}{C_0 \cup D, \Gamma \vdash^1 \mathcal{A} : \forall \bar{\alpha}(D).T}$$

By induction hypothesis on \mathcal{E} , we get a derivation \mathcal{E}' of the final judgment $C_0 \cup D \cup C', \Gamma \vdash^1 \mathcal{A} : T$, then we simply reconstruct \mathcal{D}' using (\forall -INTRO) accordingly. Disjoint condition on $\bar{\alpha}$ can be achieved by renaming $\bar{\alpha}$ to fresh variables.

- case (\forall -ELIM), where $C = C_0 \cup \psi D$, and \mathcal{D} has the form:

$$(\forall\text{-ELIM}) \frac{\mathcal{E} \frac{\vdots}{C_0, \Gamma \vdash^1 \mathcal{A} : \forall \bar{\alpha}(D).T} \quad \Phi[C_0 \cup \psi D]}{C_0 \cup \psi D, \Gamma \vdash^1 \mathcal{A} : \psi T}$$

By induction hypothesis on \mathcal{E} , we get a derivation \mathcal{E}' of the final judgment $C_0 \cup C', \Gamma \vdash^1 \mathcal{A} : \forall \bar{\alpha}(D).T$ (note: $\Phi[C_0 \cup C']$ because $C_0 \cup C' \subseteq C \cup C'$ and we know $\Phi[C \cup C']$; see Remark 2.3).

Then construct \mathcal{D}' in the same manner as \mathcal{D} , using \mathcal{E}' instead.

- cases (LET), (PAR), (SEQ), and (SUB), apply induction hypothesis in a straightforward manner.

□

Theorem (3.8). *If there is a derivation \mathcal{D} of the judgment $C \cup E, \Gamma \vdash^3 \mathcal{A} : T$ such that $C \Vdash E$ for some constraint sets C and E , then there is a derivation \mathcal{D}' of the judgment $C, \Gamma \vdash^1 \mathcal{A} : T$.*

Proof. Induction on the structure of \mathcal{D} according to the rule that is used last.

- case (VAR-ADD-INST-SUB), where $C \cup E = C_0 \cup D \cup \psi D \cup \{\psi T_0 <: T\}$. Suppose $E = \psi D \cup \{\psi T_0 <: T\}$, then \mathcal{D} has the form:

$$(\text{VAR-ADD-INST-SUB}) \frac{\Gamma(x) = \forall \bar{\alpha}(D).T_0 \quad \Phi[C_0 \cup D \cup E]}{C_0 \cup D \cup E, \Gamma \vdash^3 x : T}$$

Then let \mathcal{D}' be the following derivation:

$$\begin{array}{c} \text{(VAR-ADD)} \frac{\Gamma(x) = \forall \bar{\alpha}(D).T_0 \quad \Phi[C]}{C, \Gamma \vdash^1 x : \forall \bar{\alpha}(D).T_0} \quad C \Vdash \psi D \\ \text{(}\forall\text{-ELIM)} \frac{}{C, \Gamma \vdash^1 x : \psi T_0} \quad C \Vdash \{\psi T_0 <: T\} \\ \text{(SUB)} \frac{}{C, \Gamma \vdash^1 x : T} \end{array}$$

The constraint entailment conditions $C \Vdash \psi D$ and $C \Vdash \{\psi T_0 <: T\}$ both satisfy because of $C \Vdash E$ by Assumption 2.1 (vi).

- case (LOCAL-ADD-INST-SUB). This is similar to the previous case.
- case (SEQ-ADD), where $C \cup E = C_1 \cup C_2 \cup \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}$. Suppose $E = \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}$ and both $C_1 \subseteq C$ and $C_2 \subseteq C$, then \mathcal{D} is of the form:

$$\text{(SEQ-ADD)} \frac{\mathcal{E}_1 \frac{\vdots}{C_1, \Gamma \vdash^3 \mathcal{A}_1 : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}]} \quad \mathcal{E}_2 \frac{\vdots}{C_2, \Gamma \vdash^3 \mathcal{A}_2 : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}]} \quad \Phi[C_1 \cup C_2 \cup E]}{C_1 \cup C_2 \cup E, \Gamma \vdash^3 \mathcal{A}_1 ; \mathcal{A}_2 : [\frac{\tau_1 \ \tau_6}{\tau_3 \ \tau_8}]}$$

By induction hypothesis on \mathcal{E}_1 and \mathcal{E}_2 , we have a derivation \mathcal{E}'_1 of the judgment $C_1, \Gamma \vdash^1 \mathcal{A}_1 : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}]$ and a derivation \mathcal{E}'_2 of the judgment $C_2, \Gamma \vdash^1 \mathcal{A}_2 : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}]$. By Lemma A.2, we have \mathcal{E}''_1 and \mathcal{E}''_2 with the final judgments $C, \Gamma \vdash^1 \mathcal{A}_1 : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}]$ and $C, \Gamma \vdash^1 \mathcal{A}_2 : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}]$ respectively. Then let \mathcal{D}' be the following derivation:

$$\text{(SEQ)} \frac{\mathcal{E}''_1 \frac{\vdots}{C, \Gamma \vdash^1 \mathcal{A}_1 : [\frac{\tau_1 \ \tau_2}{\tau_3 \ \tau_4}]} \quad \mathcal{E}''_2 \frac{\vdots}{C, \Gamma \vdash^1 \mathcal{A}_2 : [\frac{\tau_5 \ \tau_6}{\tau_7 \ \tau_8}]} \quad C \Vdash \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}}{C, \Gamma \vdash^1 \mathcal{A}_1 ; \mathcal{A}_2 : [\frac{\tau_1 \ \tau_6}{\tau_3 \ \tau_8}]}$$

- case (PAR-ADD). This is similar to the previous case.
- case (LET-GEN), where \mathcal{D} is of the form:

$$\text{(LET-GEN)} \frac{\mathcal{E}_1 \frac{\vdots}{C_1 \cup E, \Gamma \vdash^3 \mathcal{A} : T_0} \quad \mathcal{E}_2 \frac{\vdots}{C_2 \cup E, \Gamma \cup \{x : S\} \vdash^3 \mathcal{B} : T} \quad \Phi[C_1 \cup C_2 \cup E]}{C_1 \cup C_2 \cup E, \Gamma \vdash^3 \text{let } x = \mathcal{A} \text{ in } \mathcal{B} : T}$$

where $S = \text{gen}(C_1, \Gamma, T_0)$ and S has the form $\forall \alpha(D).T_0$.

By induction hypothesis on \mathcal{E}_1 and \mathcal{E}_2 , there is a derivation \mathcal{E}'_1 of the judgment $C_1, \Gamma \vdash^1 \mathcal{A} : T_0$ and a derivation \mathcal{E}'_2 of the judgment $C_2, \Gamma \cup \{x : S\} \vdash^1 \mathcal{B} : T$. By Lemma A.2, we get \mathcal{E}''_1 and \mathcal{E}''_2 with the final judgments $C, \Gamma \vdash^1 \mathcal{A} : T_0$ and $C, \Gamma \cup \{x : S\} \vdash^1 \mathcal{B} : T$ respectively with $C = C_1 \cup C_2$. Then let \mathcal{D}' be the following derivation:

$$\begin{array}{c} \text{(}\forall\text{-INTRO)} \frac{\mathcal{E}''_1 \frac{\vdots}{C, \Gamma \vdash^1 \mathcal{A} : T_0} \quad \bar{\alpha} \# \text{ftv}(C - D, \Gamma)}{C, \Gamma \vdash^1 \mathcal{A} : \forall \bar{\alpha}(D).T_0} \quad \mathcal{E}''_2 \frac{\vdots}{C, \Gamma \cup \{x : S\} \vdash^1 \mathcal{B} : T} \\ \text{(LET)} \frac{}{C, \Gamma \vdash^1 \text{let } x = \mathcal{A} \text{ in } \mathcal{B} : T} \end{array}$$

Note: the disjoint condition $\bar{\alpha} \# \text{ftv}(C - D, \Gamma)$ can be satisfied by renaming bound variable.

□

A.2 Type Preservation

Lemma A.3 (Substitution). *Given the following:*

- a derivation \mathcal{D}_1 with the judgment $C_1, \Gamma \vdash^3 \mathcal{A}_1 : T_0$,
- a derivation \mathcal{D}_2 with the judgment $C_2, \Gamma \cup \{x : S\} \vdash^3 \mathcal{A}_2 : T$,
- $S = \forall \bar{\alpha}(D).T_0 = \text{gen}(C_1, \Gamma, T_0)$
- $\Phi[C_1 \cup C_2]$

Then there is a derivation \mathcal{D}' with the final judgment $C_1 \cup C_2, \Gamma \vdash^3 [x := \mathcal{A}_1]\mathcal{A}_2 : T$.

Proof. Induction on the cases of \mathcal{A}_2 .

- case $\mathcal{A}_2 = y$ for some $y \neq x$, then $[x := \mathcal{A}_1]\mathcal{A}_2 = y = \mathcal{A}_2$. In this case, \mathcal{D}_2 has the form:

$$\text{(VAR-ADD-INST-SUB)} \frac{\Gamma'(y) = \forall \bar{\alpha}'(D').T' \quad \Phi[C_2' \cup \psi D' \cup \{\psi T' <: T\}]}{C_2' \cup \psi D' \cup \{\psi T' <: T\}, \Gamma' \vdash^3 y : T}$$

where $C_2 = C_2' \cup \psi D' \cup \{\psi T' <: T\}$ and $\Gamma' = \Gamma \cup \{x : S\}$.

Construct \mathcal{E} as follows:

$$\text{(VAR-ADD-INST-SUB)} \frac{\Gamma'(y) = \forall \bar{\alpha}'(D').T' \quad \Phi[C_1 \cup C_2]}{C_1 \cup C_2, \Gamma' \vdash^3 y : T}$$

and since the binding $x : S$ is not used in the derivation, we could have a derivation \mathcal{E}' constructed the same way, but with Γ instead of Γ' . Then let \mathcal{D}' be \mathcal{E}' .

- case $\mathcal{A}_2 = x$, then $[x := \mathcal{A}_1]\mathcal{A}_2 = \mathcal{A}_1$. In this case, we construct \mathcal{D}' by propagating the additional constraints in C_2 to one of the leaves of derivation \mathcal{D}_1 , and we obtain a derivation of the final judgment $C_1 \cup C_2, \Gamma \vdash^3 \mathcal{A}_1 : T$.
- case $\mathcal{A}_2 = A$, then $[x := \mathcal{A}_1]\mathcal{A}_2 = A = \mathcal{A}_2$. This case is similar to the case where $\mathcal{A}_2 = y$ for some $y \neq x$; that is, we construct a derivation similar to \mathcal{D}_2 but with additional constraints C_1 and the environment Γ .
- all other cases are done by straightforward usage of induction hypothesis.

□

Theorem (3.9). *If there is a derivation \mathcal{D} for the judgment $C, \Gamma \vdash^3 \mathcal{A} : T$ and $\mathcal{A} \rightarrow \mathcal{A}'$ for some \mathcal{A}' according to any of the reduction rules in Definition 1.3, then there is a derivation \mathcal{D}' for the judgment $C, \Gamma \vdash^3 \mathcal{A}' : T$.*

Proof. By induction on the structure of \mathcal{D} according to the rule used last.

- base cases (VAR-ADD-INST-SUB) and (LOCAL-ADD-INST-SUB) are already in normal form, so there is no \mathcal{A}' .
- case (SEQ-ADD) where $\mathcal{A} = \mathcal{A}_1 ; \mathcal{A}_2$, and \mathcal{D} has the form:

$$\text{(SEQ-ADD)} \frac{\mathcal{E}_1 \frac{\vdots}{C_1, \Gamma \vdash^3 \mathcal{A}_1 : [\frac{\tau_1}{\tau_3} \frac{\tau_2}{\tau_4}]} \quad \mathcal{E}_2 \frac{\vdots}{C_2, \Gamma \vdash^3 \mathcal{A}_2 : [\frac{\tau_5}{\tau_7} \frac{\tau_6}{\tau_8}]} \quad \Phi[C_1 \cup C_2 \cup \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}]}{C_1 \cup C_2 \cup \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}, \Gamma \vdash^3 \mathcal{A}_1 ; \mathcal{A}_2 : [\frac{\tau_1}{\tau_3} \frac{\tau_6}{\tau_8}]}$$

- subcase $\mathcal{A}' = \mathcal{A}'_1 ; \mathcal{A}_2$ for some \mathcal{A}'_1 such that $\mathcal{A}_1 \rightarrow \mathcal{A}'_1$. By induction hypothesis on \mathcal{E}_1 , we get a derivation \mathcal{E}'_1 of the judgment $C_1, \Gamma \vdash^3 \mathcal{A}'_1 : [\frac{\tau_1}{\tau_3} \frac{\tau_2}{\tau_4}]$. Then let \mathcal{D}' be the same as \mathcal{D} except with \mathcal{E}_1 replaced by \mathcal{E}'_1 .
- subcase $\mathcal{A}' = \mathcal{A}_1 ; \mathcal{A}'_2$ for some \mathcal{A}'_2 such that $\mathcal{A}_2 \rightarrow \mathcal{A}'_2$. This is similar to the about subcase, using straightforward application of induction hypothesis.

- case (PAR-ADD) is similar to (SEQ-ADD).
- case (LET-GEN), $\mathcal{A} = \text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}_2$, and \mathcal{D} has the form:

$$\text{(LET-GEN)} \frac{\mathcal{E}_1 \frac{\vdots}{C_1, \Gamma \vdash^3 \mathcal{A} : T_0} \quad \mathcal{E}_2 \frac{\vdots}{C_2, \Gamma \cup \{x : S\} \vdash^3 \mathcal{A}_2 : T} \quad \Phi[C_1 \cup C_2]}{C_1 \cup C_2, \Gamma \vdash^3 \text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}_2 : T}$$

where $S = \forall \bar{\alpha}(D).T_0 = \text{gen}(C_1, \Gamma, T_0)$.

- subcase $\mathcal{A}' = \text{let } x = \mathcal{A}'_1 \text{ in } \mathcal{A}_2$ for some \mathcal{A}'_1 such that $\mathcal{A}_1 \rightarrow \mathcal{A}'_1$. Straightforward application of induction hypothesis.
- subcase $\mathcal{A}' = \text{let } x = \mathcal{A}_1 \text{ in } \mathcal{A}'_2$ for some \mathcal{A}'_2 such that $\mathcal{A}_2 \rightarrow \mathcal{A}'_2$. Straightforward application of induction hypothesis.
- subcase $\mathcal{A}' = [x := \mathcal{A}_1]\mathcal{A}_2$. Use substitution lemma (Lemma ??) with the premises:
 1. a derivation \mathcal{E}_1 of the judgment $C_1, \Gamma \vdash^3 \mathcal{A}_1 : T_0$,
 2. a derivation \mathcal{E}_2 of the judgment $C_2, \Gamma \cup \{x : S\} \vdash^3 \mathcal{A}_2 : T$,
 3. $S = \forall \bar{\alpha}(D).T_0 = \text{gen}(C_1, \Gamma, T_0)$, and
 4. $\Phi[C_1 \cup C_2]$.

□

Theorem (3.10). *Let \mathcal{D} be a derivation for the judgment $C, \Gamma \vdash^1 \mathcal{A} : S$ and $\mathcal{A} \rightarrow \mathcal{A}'$ for some \mathcal{A}' according to any of the reduction rules in Definition 1.3. Then there is a derivation \mathcal{D}' for the judgment $C, \Gamma \vdash^1 \mathcal{A}' : S$.*

Proof. On two cases of S .

- case $S = T$, we proceed by the following:
 1. Given $C, \Gamma \vdash^1 \mathcal{A} : T$.
 2. $C \cup D, \Gamma \vdash^3 \mathcal{A} : T$ by Theorem 3.7 from (1). Note that $C \Vdash D$.
 3. $C \cup D, \Gamma \vdash^3 \mathcal{A}' : T$ by Theorem 3.9 from (2) given that $\mathcal{A} \rightarrow \mathcal{A}'$.
 4. $C, \Gamma \vdash^1 \mathcal{A}' : T$ by Theorem 3.8 from (3) and $C \Vdash D$ from (2).
- case $S = \forall \bar{\alpha}(D).T$, we proceed by the following:
 1. Given $C, \Gamma \vdash^1 \mathcal{A} : \forall \bar{\alpha}(D).T$. Assume $\bar{\alpha}$ is fresh, so $\bar{\alpha} \# \text{ftv}(C, \Gamma)$. We have $C \supseteq D$ and $\Phi[C]$ by Lemma 2.6, so $C \Vdash D$ by Assumption 2.1 (v).
 2. $C \cup D \cup E, \Gamma \vdash^3 \mathcal{A} : T$ by Theorem 3.7 from (1) while letting ψ be the identity. Note that $C \cup D \Vdash E$.
 3. $C \cup D \cup E, \Gamma \vdash^3 \mathcal{A}' : T$ by Theorem 3.9 from (2) given that $\mathcal{A} \rightarrow \mathcal{A}'$.
 4. $C \Vdash D \cup E$ by Assumption 2.1: given $C \Vdash D$ and $C \cup D \Vdash E$, we have $C \Vdash E$ by (iii); then $C \Vdash D$ and $C \Vdash E$ give us $C \Vdash D \cup E$ by (v).
 5. $C, \Gamma \vdash^1 \mathcal{A}' : T$ by Theorem 3.8 from (3) and $C \Vdash D \cup E$ from (4).
 6. $C, \Gamma \vdash^1 \mathcal{A}' : S$ by (\forall -INTRO) from (4). Note the type variable disjointness condition can be trivially satisfied by renaming in step (1).

□

A.3 Relation to TRAFFIC

Theorem (3.12). *If there is a derivation \mathcal{D} for the judgment $\Gamma \vdash^0 \mathcal{A} : T$, then there is a derivation \mathcal{D}' for the judgment $\emptyset, \Gamma \vdash^1 \mathcal{A} : T$.*

Proof. By induction on the structure of \mathcal{D} according to the rule that is used last.

- case (VAR), \mathcal{D} has the form:

$$\text{(VAR)} \frac{\Gamma(x) = T}{\Gamma \vdash^0 x : T}$$

Then let \mathcal{D}' be the following derivation:

$$\text{(VAR-ADD)} \frac{\Gamma(x) = T \quad \Phi[\emptyset]}{\emptyset, \Gamma \vdash^1 x : T}$$

- case (LOCAL) is similar to (VAR), resulting in a derivation using rule (LOCAL-ADD).
- cases (LET) and (PAR) are shown by straightforward induction hypothesis.
- cases (SEQ) and (SUB) are also shown by induction hypothesis, noting that we map $\Delta \Vdash C$ for some constraint set C to $\emptyset \Vdash C$ where Δ is built into the notion of constraint entailment. See Section 4.1 for more details.

□