Type Inference for Record Concatenation and Subtyping

Jens Palsberg
Purdue University*

Tian Zhao
Purdue University†

July 22, 2003

Abstract

Record concatenation, multiple inheritance, and multiple-object cloning are closely related and part of various language designs. For example, in Cardelli's untyped Obliq language, a new object can be constructed from several existing objects by cloning followed by concatenation; an error is given in case of field name conflicts. Type systems for record concatenation have been studied by Wand, Harper and Pierce, Remy, and others; and type inference for the combination of record concatenation and subtyping has been studied by Sulzmann and by Pottier.

In this paper we present a type inference algorithm for record concatenation, subtyping, and recursive types. Our example language is the Abadi-Cardelli object calculus extended with a concatenation operator. Our algorithm enables type checking of Obliq programs without changing the programs at all. We prove that the type inference problem is NP-complete.

*Purdue University, Dept. of Computer Science, W Lafayette, IN 47907, palsberg@cs.purdue.edu.

†Purdue University, Dept. of Computer Science, W Lafayette, IN 47907, tzhao@cs.purdue.edu.

1 Introduction

1.1 Background

In Cardelli's untyped Obliq language [4], the operation

$$clone(a_1, \ldots, a_n)$$

creates a new object that contains the fields and methods of all the argument objects $a_1, \ldots, a_n$. This is done by first cloning each of $a_1, \ldots, a_n$, and then concatenating the clones. An error is given in case of field name conflicts, that is, in case at least two of $a_1, \ldots, a_n$ have a common field. Cardelli notes that useful idioms are:

$$clone(a, \{l : v\})$$

to inherit the fields of a and add a new field l with initial value v, and:

$$clone(a_1, a_2)$$

to multiply inherit from a_1 and a_2.

Obliq's multiple-object cloning is an instance of the idea of concatenating two records of data. In a similar fashion, languages such as C++ [22] and Borning and Ingalls' [3] version of Smalltalk allow multiple inheritance of classes.

In this paper we focus on languages such as Obliq where concatenation is a run-time operation and where a field name conflict is considered an error; such concatenation is known as *symmetric concatenation*. There are several ways of handling field name conflicts. One idea is to do run-time checking, and thereby add some overhead to the execution time. Another idea, which we pursue here, is to statically detect field name errors by a type system. The main challenge for such a type system is to find out which objects will eventually be concatenated and give them types that support concatenation.

Type systems for record concatenation have been studied by Wand [25], Harper and Pierce [8], Remy [20], Shields and Meijer [21], Tsuiki [24], Zwanenburg [27, 28] and others. These type systems use ideas such as row variables, present-fields and absent-fields, type-indexed rows, second-order types, and intersection types. More recently, Sulzmann [23] and Pottier [19] have studied type inference with the combination of record concatenation and subtyping. None of these algorithms are, as far as we are aware, known to run in polynomial time.

In this paper we investigate the idea of using variance annotations [17, 1] together with subtyping and recursive types as the basis for typing record concatenation. Following Glew [7], we will use two forms of record types. The variance annotation 0, as in

$$[\ell_i : B_i \ ^{i \in 1..n}]^0,$$

denotes that records of that type *can* be concatenated, and that subtyping *cannot* be used. The variance annotation $\rightarrow$, as in

$$[\ell_i : B_i \ ^{i \in 1..n}]^{\rightarrow},$$

denotes that records of that type *cannot* be concatenated, and that subtyping *can* be used. For example, if we have

$$\begin{array}{rcl} [l : 5, m : true] & : & [l : int, m : boolean]^0 \\ {[n : 7]} & : & [n : int]^0 \end{array}$$

then for the concatenation (denoted by $+$) of the two records we would get

$$\begin{array}{rcl} [l : 5, m : true] + [n : 7] & : & [l : int, m : boolean]^0 \oplus [n : int]^0 \\ & = & [l : int, m : boolean, n : int]^0. \end{array}$$

where $\oplus$ is the symmetric concatenation operation on record types which is only defined when the labels sets are disjoint and the two types both have the variance annotation 0. The idea is that if an object has type $[l_i : t_i]^0$, then we know exactly which fields are in the object, and hence we know which other fields we can safely add without introducing a field name conflict. The more flexible types $[\ell_i : B_i \ ^{i \in 1..n}]^{\rightarrow}$ can be used to type objects that will not be concatenated with other objects.

We restrict our attention to width-subtyping for types with variance annotation $\rightarrow$, and we allow subtyping from variance annotation 0 to $\rightarrow$. Going from 0 to $\rightarrow$ is in effect to forget that a record of that type can be concatenated with other records. Our type system is simpler and less expressive than some previous type systems for record concatenation. Our goal is to analyze the computational complexity of type inference. That complexity may well be less than the complexity of type inference for some of the more expressive type systems.

1.2 Our Result

We present the design of a type inference algorithm for the Abadi-Cardelli object calculus extended with a concatenation operator. The type system supports subtyping and recursive types. Our algorithm enables type checking of Obliq programs without changing the programs at all; extending our results to Obliq is left for future work. We prove that the type inference problem is NP-complete.

Our NP algorithm works by reducing type inference to the problem of solving a set of *constraints*. A constraint is a pair (A, B), where A and B are types that may contain type variables and the concatenation operator $\oplus$; and the goal is to find a substitution S such that for each constraint (A, B), we have $S(A) \leq S(B)$ where $\leq$ is the subtype order. We will use R to range over sets of constraints; we will often refer to R as a relation on types. A key theorem states:

> **Theorem** A set of constraints is solvable if and only there exists a closed superset which is consistent.

Here, "closure" means that certain syntactic consequences of the constraints have been added to the constraint set, and "consistent" means that there are no obviously unsatisfiable constraints (e.g., $([m : V]^0, [l : U]^0)$). The algorithm constructs a solution from a closed, consistent constraint set. To solve a constraint set R generated from a program a, we first guess a superset R' of R. Next we check that R' is closed and consistent; this can be done in polynomial time. This framework has been used for solving subtype constraints for a variety of types [12, 9, 15, 13, 14, 16]. A key difference from these papers is that our constraint problem does not admit a *smallest* closed superset which is consistent. As a reflection of that, the algorithms in [12, 9, 15, 13, 14, 16] all run in polynomial time, while the type inference problem considered here is NP-complete. This is because in the referenced papers, the smallest closed superset of a given constraint set can be computed in polynomial time, while our algorithm has to guess a closed superset.

All type-inference algorithms based on this framework, including the one in this paper, can be viewed as whole-program analyses because they use a constraint set generated from the whole program. A whole-program analysis can be made modular in several ways [6]. For example, we can generalize to type inference with respect to a fixed (non-empty) typing environment. One would start the algorithm with an initial set of constraints for program variables, derived from that fixed environment. Thus, one could collect (or constrain) the substitution provided by a run of the algorithm as an interface to a further program fragment that uses the first one as a library.

Our algorithm uses a new notion of closure and a traditional notion of consistency. Our seven closure rules capture various aspects of the subtyping order. For example, one closure rule ensures that if

$$(V \oplus V', [l : U]^{\rightarrow})$$

is a constraint, then either V or V' must be forced to have an l-field, as illustrated in the example below. That closure rule highlights why the type inference problem is NP-complete: there is a choice which possibly later has to be undone.

In our proof of the main theorem we use the technique of Palsberg, Zhao, and Jim [16] that employs a convenient characterization of the subtyping order (Lemma 2.6). The characterization uses notions of subtype-closure and subtype-consistency that are different, yet closely related, to the already-mentioned notions of what we for clarity will call satisfaction-closure and satisfaction-consistency. The paper [16] concerns type inference with both covariant and invariant fields, and for types that all allow width-subtyping. In the present paper, all fields are invariant, but some types (those with the variance-annotation 0) do not admit non-trivial subtyping. While the type inference algorithms reported in the two papers are entirely different, their correctness proofs have the same basic structure.

1.3 Example

We now present an example that gives a taste of the definitions and techniques that are used later in the paper. Our example program a has two methods l and m:

$$a \quad = \quad [l = \varsigma(x)(x.l + x.m).k,\ m = \varsigma(y)y.m].$$

When running our type inference algorithm by hand on this program, the result is that a is typable with type

$$a \quad : \quad [l : \mu\alpha.[k : \alpha]^0, m : [\ ]^0]^0.$$

The goal of this section is to illustrate how the algorithm arrives at that conclusion.

We can use the rules in Section 4 to generate the following set of constraints, called R. In the left column are all occurrences of subterms in the program; in the right column are the constraints generated for each occurrence. We use $A \equiv B$ to denote the pair of constraints (A, B) and (B, A).

Occurrence	Constraints
x	(U_x, V_x)
y	(U_y, V_y)
a	$([l : V_{(x.l+x.m).k},\ m : V_{y.m}]^0, V_a)$
	$U_x \equiv [l : V_{(x.l+x.m).k},\ m : V_{y.m}]^0$
	$U_y \equiv [l : V_{(x.l+x.m).k},\ m : V_{y.m}]^0$
$(x.l + x.m).k$	$(V_{x.l+x.m}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$
	$(U_{(x.l+x.m).k}, V_{(x.l+x.m).k})$
$x.l + x.m$	$(V_{x.l} \oplus V_{x.m}, V_{x.l+x.m})$
$x.l$	$(V_x, [l : U_{x.l}]^{\rightarrow})$
	$(U_{x.l}, V_{x.l})$
$x.m$	$(V_x, [m : U_{x.m}]^{\rightarrow})$
	$(U_{x.m}, V_{x.m})$
$y.m$	$(V_y, [m : U_{y.m}]^{\rightarrow})$
	$(U_{y.m}, V_{y.m})$

Notice that, for each bound variable x, we have a type variable U_x. Moreover, for each occurrence of x, we have a type variable V_x. Intuitively, U_x stands for the type of x in the type environment, while V_x stands for the type of an occurrence of x after subtyping. Similarly, $U_{x.l}$ stands for the type of $x.l$ before subtyping, while $V_{x.l}$ stands for the type of $x.l$ after subtyping.

Next, our type inference algorithm will guess a so-called satisfaction-closed superset R' of R. We will here display and motivate some of the interesting constraints in a particular R'. First, from the constraints

$$(U_x, V_x)$$
$$(V_x, [l : U_{x.l}]^{\rightarrow})$$

and transitivity, we have

$$(U_x, [l : U_{x.l}]^{\rightarrow})$$

in R'. Second, from that constraint and

$$U_x \equiv [l : V_{(x.l+x.m).k},\ m : V_{y.m}]^0$$

and the observation that fields have invariant subtyping, we have

$$(V_{(x.l+x.m).k}, U_{x.l})$$

in R'. Third, from the constraints

$$\begin{array}{l}(V_{x.l} \oplus V_{x.m}, V_{x.l+x.m}) \\ (V_{x.l+x.m}, [k : U_{(x.l+x.m).k}]^{\rightarrow})\end{array}$$

and transitivity, we have

$$(V_{x.l} \oplus V_{x.m}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$$

in R'. At this point there is a choice. We can force either $V_{x.l}$ or $V_{x.m}$ to be mapped to a type with a k-field. Since there are no other significant constraints on either $V_{x.l}$ or $V_{x.m}$, both choices will be fine. Our algorithm chooses the first one, and so we have the constraint

$$(V_{x.l}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$$

in R'. After this constraint has been added, we can apply transitivity three times to:

$$\begin{array}{l}(U_{(x.l+x.m).k}, V_{(x.l+x.m).k}) \\ (V_{(x.l+x.m).k}, U_{x.l}) \\ (U_{x.l}, V_{x.l}) \\ (V_{x.l}, [k : U_{(x.l+x.m).k}]^{\rightarrow})\end{array}$$

so we have

$$(U_{(x.l+x.m).k}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$$

in R'. The last constraint makes it apparent that recursive types are needed to solve the constraint system and therefore to type the example program.

Note that the choice we made in applying closure rules to $(V_{x.l} \oplus V_{x.m}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$ implies that sometimes there is no unique solution to our type-inference problem.

Thus, if we want to do type inference for a program fragment without an initial type environment, the best we can do is to generate the constraints, perhaps simplify them [18], and delay solving them until the constraints for the other program fragments become available.

Once our type inference algorithm has guessed a sat-closed R', it checks whether R' is sat-consistent, that is, whether there is at least one constraint which obviously is unsolvable, e.g., $([m : V]^0, [l : U]^0)$. If R' is not sat-consistent, then R has no solution. In the case of the example program, R' is sat-consistent, and our type inference algorithm then derives the following solution from R'. Define

$$\begin{array}{rcl} P & \equiv & \mu\alpha.[k : \alpha]^0 \\ Q & \equiv & [l : P, m : [\ ]^0]^0 \\ E & \equiv & \emptyset[x : Q] \\ F & \equiv & \emptyset[y : Q], \end{array}$$

where P, Q are types, and E, F are type environments. Note that we use so-called equi-recursive types that satisfy a certain equation, rather than the kind of recursive types that have to be explicitly folded and unfolded.

We can derive $\emptyset \vdash a : Q$ as follows.

$$\dfrac{\dfrac{\dfrac{E \vdash (x.l + x.m) : [k : P]^0}{E \vdash (x.l + x.m) : [k : P]^{\rightarrow}}}{E \vdash (x.l + x.m).k : P} \qquad \dfrac{\dfrac{F \vdash y : Q}{F \vdash y : [m : [\ ]^0]^{\rightarrow}}}{F \vdash y.m : [\ ]^0}}{\emptyset \vdash a : Q}$$

Notice the two uses of subsumption:

$$\begin{array}{rcl} [k : P]^0 & \leq & [k : P]^{\rightarrow} \\ Q & \leq & [m : [\ ]^0]^{\rightarrow}. \end{array}$$

We can derive $E \vdash (x.l + x.m) : [k : P]^0$ as follows. Notice that $[k : P]^0 = [k : P]^0 \oplus [\ ]^0$.

$$\dfrac{\dfrac{\dfrac{E \vdash x : Q}{E \vdash x : [l : [k : P]^0]^{\rightarrow}}}{E \vdash x.l : [k : P]^0} \qquad \dfrac{\dfrac{E \vdash x : Q}{E \vdash x : [m : [\ ]^0]^{\rightarrow}}}{E \vdash x.m : [\ ]^0}}{E \vdash (x.l + x.m) : [k : P]^0}$$

Notice the two uses of subsumption:

$$\begin{array}{rcl} Q & \leq & [l : [k : P]^0]^{\rightarrow} \\ Q & \leq & [m : [\ ]^0]^{\rightarrow}. \end{array}$$

We derive the first of these inequalities using the unfolding rule for recursive types to get

$$P = \mu\alpha.[k : \alpha]^0 = [k : \mu\alpha.[k : \alpha]^0]^0 = [k : P]^0,$$

and therefore

$$Q = [l : P, m : [\ ]^0]^0 = [l : [k : P]^0, m : [\ ]^0]^0.$$

Here is an alternative typing, which arises from forcing $V_{x.m}$ to be mapped to a type with a k-field:

$$\emptyset \vdash a : [l : [\ ]^0, m : [k : [\ ]^0]^0]^0.$$

2 Types and Subtyping

We will work with recursive types, and we choose to represent them by possibly infinite trees.

2.1 Defining types as infinite trees

We use U, V to range over the set $\mathcal{TV}$ of type variables; we use k, ℓ, m to range over labels drawn from some possibly infinite set Labels of method names; and we use v to range over the set $\mathsf{Variances} = \{0, \rightarrow\}$ of variance annotations. Variance annotations are ordered by the smallest partial order $\sqsubseteq$ such that $0 \sqsubseteq \rightarrow$.

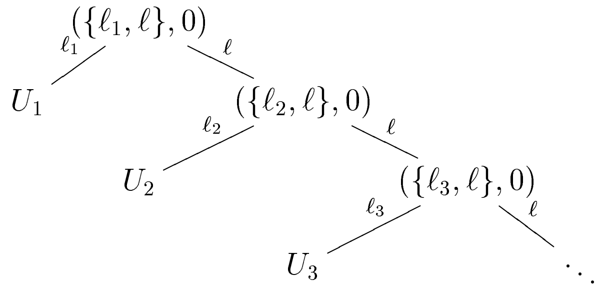
The alphabet Σ of our trees is defined

$$\Sigma = \mathcal{TV} \cup (\mathcal{P}(\mathsf{Labels}) \times \mathsf{Variances}).$$

A *path* is a finite sequence $\alpha \in \mathsf{Labels}^*$ of labels, with juxtaposition for concatenation of paths, and ϵ for the empty sequence. A *type* or *tree* A is a partial function from paths into Σ, whose domain is nonempty and prefix closed, and such that $A(\alpha) = (\{\ell_i \mid i \in I\}, v)$ if and only if $\forall i,\ A(\alpha\ell_i)$ is defined. We use A, B, C to range over the set $\mathcal{T}(\Sigma)$ of trees.

Note that trees need not be finitely branching or regular. A regular tree has finitely many distinct subtrees [5]. Of course, we will be particularly interested in two subsets of $\mathcal{T}(\Sigma)$, the finite trees $\mathcal{T}_{\mathrm{fin}}(\Sigma)$ and the finitely branching and regular trees $\mathcal{T}_{\mathrm{reg}}(\Sigma)$. Some definitions, results, and proofs are given in terms of $\mathcal{T}(\Sigma)$, in such a way that they immediately apply to $\mathcal{T}_{\mathrm{fin}}(\Sigma)$ and $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

An example tree is given below.



We now introduce some convenient notation. We write $A(\alpha) = \uparrow$ if A is undefined on α. If for all $i \in I$, B_i is a tree, ℓ_i is a distinct label, and

$v \in \mathsf{Variances}$, then $[\ell_i : B_i \;^{i \in I}]^v$ is the tree A such that

$$A(\alpha) \;=\; \begin{cases} (\{\ell_i \mid i \in I\}, v) & \text{if } \alpha = \epsilon \\ B_i(\alpha') & \text{if } \alpha = \ell_i \alpha' \text{ for some } i \in I \\ \uparrow & \text{otherwise.} \end{cases}$$

We abuse notation and write U for the tree A such that $A(\epsilon)$ is the type variable U and $A(\alpha) = \uparrow$ for all $\alpha \neq \epsilon$.

Recursive types are regular trees, and they can be presented by μ-expressions [5, 2] generated by the following grammar:

$$\begin{array}{lcll} A, B & ::= & U, V & \text{type variable} \\ & \mid & [\ell_i : B_i \;^{i \in 1..n}]^{\phi} & \text{object type } (\ell_i\text{'s distinct, } \phi ::= 0 \mid \rightarrow) \\ & \mid & \mu U.A & \text{recursive type} \end{array}$$

We can now define the concatenation operator $\oplus$. If

$$\begin{array}{rcl} A & = & [\ell_i : B_i \;^{i \in I}]^0 \\ A' & = & [\ell_i : B_i \;^{i \in I'}]^0 \end{array}$$

and $I \cap I' = \emptyset$, then

$$A \oplus A' = [\ell_i : B_i \;^{i \in I \cup I'}]^0,$$

and otherwise $A \oplus A'$ is undefined.

2.2 Defining Subtyping via Simulations

Our subtyping order supports width subtyping but not depth subtyping.

Definition 2.1 A relation R over $\mathcal{T}(\Sigma)$ is called a *simulation* if for all $(A, A') \in R$, we have the following conditions.

- For all U, $A = U$ if and only if $A' = U$.
- For all ℓ_i, $i \in I'$, B_i', if $A' = [\ell_i : B_i' \;^{i \in I'}]^{\phi'}$, then there exist B_i such that

$$\begin{array}{c} A = [\ell_i : B_i \;^{i \in I}]^{\phi}, \;\; I' \subseteq I, \;\; \phi' \sqsupseteq \phi \\ (B_i, B_i'), (B_i', B_i) \in R, \;\; \phi' = 0 \Rightarrow I' = I. \end{array}$$

$\square$

Notice that a simulation can contain pairs such as $([\ldots]^0, [\ldots]^{\rightarrow})$, but not $([\ldots]^{\rightarrow}, [\ldots]^0)$. Notice also that the last line of Definition 2.1 enforces no depth subtyping.

For example, the empty relation on $\mathcal{T}(\Sigma)$ and the identity relation on $\mathcal{T}(\Sigma)$ are both simulations. Simulations are closed under unions and intersections, and there is a largest simulation, which we call $\leq$ and use as our subtyping order:

$$\leq \; = \bigcup\{R \mid R \text{ is a simulation}\}. \tag{1}$$

Alternately, $\leq$ can be seen as the maximal fixed point of a monotone function on $\mathcal{P}(\mathcal{T}(\Sigma) \times \mathcal{T}(\Sigma))$. Then we immediately have the following result.

Lemma 2.2 *$A \leq A'$ if and only if*

- *For all U, $A = U$ if and only if $A' = U$.*
- *For all ℓ_i, B'_i, $i \in I'$, and ϕ', if $A' = [\ell_i : B'_i \;{}^{i \in I'}]^{\phi'}$, then there exist B_i, such that*

$$\begin{aligned} A = [\ell_i : B_i \;{}^{i \in I}]^{\phi}, \;\; I' \subseteq I, \;\; \phi' \sqsupseteq \phi, \;\; \textit{and} \\ \forall i \in I, B_i = B'_i, \;\; \phi' = 0 \Rightarrow I' = I. \end{aligned}$$

All of these results are standard in concurrency theory, and have easy proofs, c.f. [10]. Similarly, it is easy to show that $\leq$ is a preorder. Our simulations differ from the simulations typically found in concurrency in that they are all anti-symmetric (again, the proof is easy).

Lemma 2.3 *$\leq$ is a partial order.*

Proof. See Appendix A. $\square$

We may apply the principle of *co-induction* to prove that one type is a subtype of another:

> **Co-induction**: To show $A \leq B$, it is sufficient to find a simulation R such that $(A, B) \in R$.

2.3 A characterization of subtyping

We now give a characterization of subtyping (Lemma 2.6) which will be used in the proof of the main theorem (Theorem 5.15). Suppose R is a relation on types, and we want to know whether $A \leq B$ for every $(A, B) \in R$. By co-induction this is equivalent to the existence of a simulation containing R. And since simulations are closed under intersection, this is equivalent to the existence of a *smallest* simulation containing R. We can characterize this smallest simulation as follows.

Definition 2.4 We say a relation R on types is subtype-closed if $([\ell : B, \ldots]^{\phi}, [\ell : B', \ldots]^{\phi'}) \in R$ implies $(B, B'), (B', B) \in R$. □

Note that the subtype-closed relations on types are closed under intersection; therefore for any relation R on types, we may define its *subtype-closure* to be the smallest subtype-closed relation containing R. Every simulation is subtype-closed, and subtype-closure is a monotone operation.

Definition 2.5 We say a relation R on types is subtype-consistent if $[\ell_i : B_i{}^{i \in I}]^{\phi},\ [\ell_i : B_i'{}^{i \in I'}]^{\phi'}) \in R$, implies

- if $\phi' = 0$, then $\phi = 0$ and $I = I'$,
- if $\phi' = \rightarrow$, then $I \supseteq I'$.

□

Note that every simulation is subtype-consistent, and moreover, any subset of an subtype-consistent set is subtype-consistent.

Lemma 2.6 *Let R be a relation on types. The following statements are equivalent.*

1. *$A \leq B$ for every $(A, B) \in R$.*
2. *The subtype-closure of R is a simulation.*
3. *The subtype-closure of R is subtype-consistent.*

Proof.

- (2) $\Rightarrow$ (1): Immediate by co-induction.

- (1) $\Rightarrow$ (3): R is a subset of $\leq$, so by monotonicity and the fact that $\leq$ is subtype-closed, the subtype-closure of R is a subset of $\leq$. Then since $\leq$ is subtype-consistent, its subset, the subtype-closure of R, is subtype-consistent.

- (3) $\Rightarrow$ (2): Let R' be the subtype-closure of R, and suppose $(A, A') \in R'$.

  If $A = U$, by subtype-consistency $A' = U$; and similarly, if $A' = U$, then $A = U$.

  If $A' = [\ell_i : B'_i \ ^{i \in I'}]^{\phi'}$, by subtype-consistency A must be of the form $[\ell_i : B_i \ ^{i \in I}]^{\phi}$, where $\phi \sqsubseteq \phi'$. And since R' is subtype-closed, $(B_i, B'_i), (B'_i, B_i) \in R'$ and $I' \subseteq I$, and $\phi' = 0 \Rightarrow I' = I$, as desired.

$\square$

3 The Abadi-Cardelli Object Calculus

We now present an extension of the Abadi-Cardelli object calculus [1] and a type system. The types are recursive types as defined in the previous section.

We use x, y to range over term variables. Expressions are defined by the following grammar.

a, b, c	$::=$	x	variable
	$\mid$	$[\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$	object (ℓ_i distinct)
	$\mid$	$a.\ell$	field selection / method invocation
	$\mid$	$(a.\ell \Leftarrow \varsigma(x)b)$	field update / method update
	$\mid$	$a_1 + a_2$	object concatenation

An object $[\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$ has method names ℓ_i and methods $\varsigma(x_i)b_i$. The order of the methods does not matter. Each method binds a name x which denotes the smallest enclosing object, much like "this" in Java. Those names can be chosen to be different, so within a nesting of objects, one can refer to any enclosing object. A *value* is of the form $[\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$. A *program* is a closed expression.

A small-step operational semantics is defined by the following rules:

- If $a \equiv [\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, then, for $j \in 1..n$,

  - $a.\ell_j \ \rightsquigarrow \ b_j[x_j := a]$,

– $(a.\ell_j \Leftarrow \varsigma(y)b) \ \rightsquigarrow\ a[\ell_j \leftarrow \varsigma(y)b]$.

- If $a_1 \equiv [\ell_i = \varsigma(x_i)b_i\ ^{i \in I_1}]$, $a_2 \equiv [\ell_i = \varsigma(x_i)b_i\ ^{i \in I_2}]$, and $I_1 \cap I_2 = \emptyset$, then

$$a_1 + a_2 \ \rightsquigarrow\ [\ell_i = \varsigma(x_i)b_i\ ^{i \in I_1 \cup I_2}].$$

- If $b \rightsquigarrow b'$ then $a[b] \rightsquigarrow a[b']$.

Here, $b_j[x_j := a]$ denotes the ς-term b_j with a substituted for free occurrences of x_j (renaming bound variables to avoid capture); and $a[\ell_j \leftarrow \varsigma(y)b]$ denotes the expression a with the ℓ_j field replaced by $\varsigma(y)b$. A *context* is an expression with one hole, and $a[b]$ denotes the term formed by replacing the hole of the context $a[\cdot]$ by the term b (possibly capturing free variables in b).

An expression b is *stuck* if it is not a value and there is no expression b' such that $b \rightsquigarrow b'$. An expression b *goes wrong* if $\exists b' : b \rightsquigarrow^* b'$ and b' is stuck.

A type environment is a partial function with finite domain which maps term variables to types in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$. We use E to range over type environments. We use $E[x : A]$ to denote a partial function which maps x to A, and maps y, where $y \neq x$, to $E(y)$.

The typing rules below allow us to derive judgments of the form $E \vdash a : A$, where E is a type environment, a is an expression, and A is a type in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

$$E \vdash x : A \quad \text{(provided } E(x) = A\text{)} \tag{2}$$

$$\frac{E[x_i : A] \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}] : A} \quad \text{(where } A = [\ell_i : B_i\ ^{i \in 1..n}]^0\text{)} \tag{3}$$

$$\frac{E \vdash a : A}{E \vdash a.\ell : B} \quad \text{(where } A \leq [\ell : B]^{\rightarrow}\text{)} \tag{4}$$

$$\frac{E \vdash a : A \quad E[x : A] \vdash b : B}{E \vdash a.\ell \Leftarrow \varsigma(x)b : A} \quad \text{(where } A \leq [\ell : B]^{\rightarrow}\text{)} \tag{5}$$

$$\frac{E \vdash a_1 : A_1 \quad E \vdash a_2 : A_2}{E \vdash a_1 + a_2 : A_1 \oplus A_2} \tag{6}$$

$$\frac{E \vdash a : A}{E \vdash a : B} \quad \text{(where } A \leq B\text{)} \tag{7}$$

The first five rules express the typing of each of the four constructs in the object calculus and the last rule is the rule of subsumption. We say that a term a is *well-typed* if $E \vdash a : A$ is derivable for some E and A. The following result can be proved by a well-known technique [11, 26].

Theorem 3.1 (Type Soundness) *Well-typed programs cannot go wrong.*

The type inference problem for our extension of the Abadi-Cardelli calculus is: given a term a, find a type environment E and a type A such that $E \vdash a : A$, or decide that this is impossible.

4 From Type Inference to Constraint Solving

A *substitution* S is a finite partial function from type variables to types in $\mathcal{T}_{\text{reg}}(\Sigma)$, written $\{U_1 := A_1, \ldots, U_n := A_n\}$. The set $\{U_1, \ldots, U_n\}$ is called the *domain* of the substitution. We identify substitutions with their graphs, and write $(S_1 \cup S_2)$ for the union of two substitutions S_1 and S_2; by convention, we assume that S_1 and S_2 agree on variables in their common domain, so $(S_1 \cup S_2)$ is a substitution. Substitutions are extended to total functions from types to types in the usual way.

Definition 4.1 A relation R is solvable if and only if there is a substitution S such that for all $(A, B) \in R$, we have $S(A) \leq S(B)$. $\square$

Definition 4.2 We will here focus on so-called C-relations (which we also refer to as *constraint sets*) which contain only pairs (A, B), where A, B are of the forms

- $[\ell : V, \ldots]^{\phi}$,
- V, or
- $V_1 \oplus V_2$,

where V, V_1, V_2 are type variables, and $\phi \in \{0. \rightarrow\}$. $\square$

While $V_1 \oplus V_2$ is not a type, it will become a type once we apply a substitution and get $S(V_1) \oplus S(V_2)$, provided the concatenation is defined. Note that if $V_1 \oplus V_2$ is in R, and R is solvable, then the solution, say S, must make $S(V_1) \oplus S(V_2)$ well-defined. To avoid introducing special terminology for the left-hand sides and right-hand sides of constraints, we will abuse the word type and call $V_1 \oplus V_2$ a type in the remainder of the paper.

We now prove that the type inference problem is equivalent to solving constraints in the form of C-relations.

We write $E' \leq E$ if, whenever $E(x) = A$, there is an $A' \leq A$ such that $E'(x) = A'$. The following standard result can be proved by induction on typings.

Lemma 4.3 (Weakening) *If $E \vdash c : C$ and $E' \leq E$, then $E' \vdash c : C$.*

By a simple induction on typing derivations, we obtain the following syntax-directed characterization of typings. The proof uses only the reflexivity and transitivity of $\leq$ which can be derived from Lemma 2.2.

Lemma 4.4 (Characterization of Typings) *$E \vdash c : C$ if and only if one of the following cases holds:*

- *$c = x$ and $E(x) \leq C$;*
- *$c = a.\ell$, and for some A and B, $E \vdash a : A$, $A \leq [\ell : B]^{\rightarrow}$, and $B \leq C$;*
- *$c = [\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, and for some A, and some B_i for $i \in 1..n$, $E[x_i : A] \vdash b_i : B_i$, and $A = [\ell_i : B_i \ ^{i \in 1..n}]^0 \leq C$; or*
- *$c = (a.\ell \Leftarrow \varsigma(x)b)$, and for some A and B, $E \vdash a : A$, $E[x : A] \vdash b : B$, $A \leq [\ell : B]^{\rightarrow}$, and $A \leq C$.*
- *$c = a_1 + a_2$, and for some A_1, A_2, $E \vdash a_1 : A_1$, $E \vdash a_2 : A_2$, and $A_1 \oplus A_2 \leq C$.*

We now show how to generate a C-relation from a given program.

Definition 4.5 Let c be a ς-term in which all free and bound variables are pairwise distinct. We define X_c, Y_c, E_c, and $\mathcal{C}(c)$ as follows.

- X_c is a set of fresh type variables. It consists of a type variable U_x for every term variable x appearing in c.
- Y_c is a set of fresh type variables. It consists of a type variable $V_{c'}$ for each occurrence of a subterm c' of c, and a type variable $U_{c'}$ for each occurrence of a select subterm $c' = a.\ell$ of c. (If c' occurs more than once in c, then $U_{c'}$ and $V_{c'}$ are ambiguous. However, it will always be clear from context which occurrence is meant.)

- E_c is a type environment, defined by

$$E_c = \{x : U_x \mid x \text{ is free in } c\}.$$

- $\mathcal{C}(c)$ is the set of the following constraints over X_c and Y_c:
  - For each occurrence in c of a variable x, the constraint
$$(U_x, V_x). \tag{8}$$
  - For each occurrence in c of a subterm of the form $a.\ell$, the two constraints
$$(V_a, [\ell : U_{a.\ell}]^{\rightarrow}) \tag{9}$$
$$(U_{a.\ell}, V_{a.\ell}). \tag{10}$$
  - For each occurrence in c of a subterm of the form $[\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]$, the constraint
$$([\ell_i : V_{b_i}\ ^{i \in 1..n}]^0, V_{[\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]}) \tag{11}$$
    and for each $j \in 1..n$, the constraints
$$U_{x_j} \equiv [\ell_i : V_{b_i}\ ^{i \in 1..n}]^0. \tag{12}$$
  - For each occurrence in c of a subterm of the form $(a.\ell \Leftarrow \varsigma(x)b)$, the constraints
$$(V_a, V_{(a.\ell \Leftarrow \varsigma(x)b)}) \tag{13}$$
$$V_a \equiv U_x \tag{14}$$
$$(V_a, [\ell : V_b]^{\rightarrow}). \tag{15}$$
  - For each occurrence in c of a subterm of the form $(a_1 + a_2)$, the constraint
$$(V_{a_1} \oplus V_{a_2}, V_{(a_1+a_2)}), \tag{16}$$

$\square$

In the definition of $\mathcal{C}(c)$, each equality $A \equiv B$ denotes the two inequalities (A, B) and (B, A).

Theorem 4.6 *$E \vdash c : C$ if and only if there is a solution S of $\mathcal{C}(c)$ such that $S(V_c) = C$ and $S(E_c) \subseteq E$.*

Each direction of the theorem can be proved separately. However, the proofs share a common structure, so for brevity we will prove them together. The two directions follow immediately from the two parts of the next lemma.

Lemma 4.7 *Let c_0 be a ς-term. For every subterm c of c_0,*

1. *if $E \vdash c : C$, then there is a solution S_c of $\mathcal{C}(c)$ such that $S_c(V_c) = C$ and $S_c(E_c) \subseteq E$; and*
2. *if S is a solution of $\mathcal{C}(c_0)$, then $S(E_c) \vdash c : S(V_c)$.*

Proof. The proof is by induction on the structure of c. In (2), we will often use the fact that any solution to $\mathcal{C}(c_0)$ (in particular, S) is a solution to $\mathcal{C}(c) \subseteq \mathcal{C}(c_0)$.

- If $c = x$, then $E_c = \{x : U_x\}$ and $\mathcal{C}(c) = \{(U_x, V_x)\}$.
  1. Define $S_c = \{U_x := E(x), V_x := C\}$. Then $S_c(V_c) = S_c(V_x) = C$, and $S_c(E_c) = \{x : E(x)\} \subseteq E$.
     Furthermore, by Lemma 4.4, $E(x) \leq C$, so S_c is a solution to $\mathcal{C}(c)$.
  2. By (2), $S(E_c) \vdash c : S(U_x)$.
     And since $S(U_x) \leq S(V_x) = S(V_c)$, we have $S(E_c) \vdash c : S(V_c)$ by (7).

- If $c = a.\ell$, then $E_c = E_a$ and $\mathcal{C}(c) = \mathcal{C}(a) \cup \{(V_a, [\ell : U_{a.\ell}]^{\rightarrow}),\ (U_{a.\ell}, V_{a.\ell})\}$.
  1. By Lemma 4.4, for some A and B, $E \vdash a : A$, $A \leq [\ell : B]^{\rightarrow}$, and $B \leq C$.
     By induction there is a solution S_a of $\mathcal{C}(a)$ such that $S_a(V_a) = A$ and $S_a(E_a) \subseteq E$.
     Define $S_c = S_a \cup \{U_{a.\ell} := B, V_{a.\ell} := C\}$. Then S_c solves $\mathcal{C}(c)$, $S_c(V_c) = S_c(V_{a.\ell}) = C$, and $S_c(E_c) = S_a(E_a) \subseteq E$.

2. By induction, $S(E_a) \vdash a : S(V_a)$.
   Since $S(V_a) \leq S([\ell : U_{a.\ell}]^{\rightarrow})$, by (7) we have $S(E_a) \vdash a : S([\ell : U_{a.\ell}]^{\rightarrow})$.
   Then by (4), $S(E_a) \vdash a.\ell : S(U_{a.\ell})$.
   Since $S(U_{a.\ell}) \leq S(V_{a.\ell}) = S(V_c)$, by (7) we have $S(E_a) \vdash a.\ell : S(V_c)$.
   Finally, $E_c = E_a$ and $c = a.\ell$, so $S(E_c) \vdash c : S(V_c)$ as desired.

- If $c = [\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, then $E_c = \bigcup_{i \in 1..n}(E_{b_i} \backslash x_i)$, and

$$\begin{aligned} \mathcal{C}(c) = \{ \ & ([\ell_i : V_{b_i} \ ^{i \in 1..n}]^0, V_c \ )\} \\ \cup \{ \ & U_{x_j} \equiv [\ell_i : V_{b_i} \ ^{i \in 1..n}]^0 \mid j \in 1..n \ \} \\ \cup \ & (\textstyle\bigcup_{i \in 1..n} \mathcal{C}(b_i)). \end{aligned}$$

1. By Lemma 4.4, for some A, and some B_i for $i \in 1..n$, we have $E[x_i : A] \vdash b_i : B_i$ and $A = [\ell_i : B_i \ ^{i \in 1..n}]^0 \leq C$.
   By induction, for every $i \in 1..n$ there is a substitution S_{b_i} such that S_{b_i} solves $\mathcal{C}(b_i)$, $S_{b_i}(V_{b_i}) = B_i$, and $S_{b_i}(E_{b_i}) \subseteq E[x_i : A]$.
   We first assume that the domain of any S_{b_i} is $X_{b_i} \cup Y_{b_i}$ (else restrict S_{b_i} to this set). Let $S_c = (\bigcup_{i \in 1..n} S_{b_i}) \cup \{V_c := C\}$
   Clearly, if S_c is well-defined, then it is a solution to $\mathcal{C}(c)$, $S_c(V_c) = C$, and $S_c(E_c) \subseteq E$.
   To show that S_c is well-defined, it suffices to show that for any distinct $j, k \in 1..n$, the substitutions S_{b_j} and S_{b_k} agree on all type variables in their common domain. And if U is in the domain of both S_{b_j} and S_{b_k}, it must have the form U_y for some term variable y free in both b_j and b_k.
   Then y must be assigned a type by E, so the conditions $S_{b_j}(E_{b_j}) \subseteq E[x_j : A]$ and $S_{b_k}(E_{b_k}) \subseteq E[x_k : A]$ guarantee that $S_{b_j}(U_y) = E(y) = S_{b_k}(U_y)$. Therefore S_c is well-defined, as desired.
2. By induction, $S(E_{b_j}) \vdash b_j : S(V_{b_j})$ for all $j \in 1..n$.
   By weakening, $S(E_c[x_j : U_{x_j}]) \vdash b_j : S(V_{b_j})$ for all $j \in 1..n$.
   Since S solves $\mathcal{C}(c)$, $S(U_{x_j}) = S([\ell_i : V_{b_i} \ ^{i \in 1..n}]^0)$ for all $j \in 1..n$.
   Then by (3), $S(E_c) \vdash c : S([\ell_i : V_{b_i} \ ^{i \in 1..n}]^0)$.
   Finally, since S solves $\mathcal{C}(c)$, $S([\ell_i : V_{b_i} \ ^{i \in 1..n}]^0) \leq S(V_c)$, so we have $S(E_c) \vdash c : S(V_c)$ by (7).

- If $c = (a.\ell \Leftarrow \varsigma(x)b)$, then $E_c = E_a \cup (E_b \backslash x)$, and

$$\mathcal{C}(c) = \mathcal{C}(a) \cup \mathcal{C}(b) \cup \{(V_a, V_c), V_a \equiv U_x, (V_a, [\ell : V_b]^{\rightarrow})\}.$$

  1. By Lemma 4.4, for some A and B, $E \vdash a : A$, $E[x : A] \vdash b : B$, $A \leq [\ell : B]^{\rightarrow}$, and $A \leq C$.

     By induction there is a solution S_a of $\mathcal{C}(a)$ such that $S_a(V_a) = A$ and $S_a(E_a) \subseteq E$, and a solution S_b of $\mathcal{C}(b)$ such that $S_b(V_b) = B$ and $S_b(E_b) \subseteq E[x : A]$.

     Let $S_c = S_a \cup S_b \cup \{V_c := C, U_x := A\}$. (We omit a proof that S_c is well-defined; this can be shown just as in the previous case.)

     Then S_c is a solution to $\mathcal{C}(c)$, $S_c(V_c) = C$, and $S_c(E_c) \subseteq E$.

  2. Since S solves $\mathcal{C}(c)$, $S(V_a) \leq S[l : V_b]^{\rightarrow}$. By induction $S(E_a) \vdash a : S(V_a)$ and $S(E_b) \vdash b : S(V_b)$.

     By weakening, $S(E_c) \vdash a : S(V_a)$ and $S(E_c[x : U_x]) \vdash b : S(V_b)$.

     Then by (5), $S(E_c) \vdash c : S(V_a)$, and by (7), $S(E_c) \vdash c : S(V_c)$.

- If $c = (a_1 + a_2)$, then $E_c = E_{a_1} \cup E_{a_2}$ and

$$\mathcal{C}(c) = \mathcal{C}(a_1) \cup \mathcal{C}(a_2) \cup \{(V_{a_1} \oplus V_{a_2}, V_c)\}.$$

  1. By Lemma 4.4, for some A_1 and A_2, $E \vdash a_1 : A_1$, $E \vdash a_2 : A_2$, and $A_1 \oplus A_2 \leq C$.

     By induction there is a solution S_{a_i} of $\mathcal{C}(a_i)$ such that $S_{a_i}(V_{a_i}) = A_i$, and $S_{a_i}(E_{a_i}) \subseteq E$, for $i = 1, 2$.

     Let $S_c = S_{a_1} \cup S_{a_2} \cup \{V_c := C\}$. (We omit a proof that S_c is well-typed; this can be shown as above.) Then S_c is a solution to $\mathcal{C}(c)$, $S_c(V_c) = C$, and $S_c(E_c) \subseteq E$.

  2. By induction $S(E_{a_1}) \vdash a_1 : S(V_{a_1})$ and $S(E_{a_2}) \vdash a_2 : S(V_{a_2})$.

     By weakening, $S(E_c) \vdash a_1 : S(V_{a_1})$ and $S(E_c) \vdash a_2 : S(V_{a_2})$.

     Then by (6), $S(E_c) \vdash c : S(V_{a_1}) \oplus S(V_{a_2})$, and by (7), $S(E_c) \vdash c : S(V_c)$.

$\square$

5 Solving Constraints

In this section we present an algorithm for deciding whether a C-relation R is solvable. We first list the terminology used in the later definitions.

$$\begin{array}{rcl} \mathsf{Types} & = & \text{the set of types} \\ \mathsf{States} & = & \mathrm{P}(\mathsf{Types}) \\ \mathsf{RelTypes} & = & \mathrm{P}(\mathsf{Types} \times \mathsf{Types}) \\ \mathsf{RelStates} & = & \mathrm{P}(\mathsf{States} \times \mathsf{States}) \end{array}$$

We use $\mathcal{T}$ to range over sets of types. For any type A such that $A(\epsilon) = (S, \phi)$, we write $labs(A) = S$. For any type A and label ℓ, $A.\ell$ is B if $A = [\ell : B \ldots]^\phi$, and is undefined otherwise. Notice that $A(\ell\alpha) = (A.\ell)(\alpha)$. We also make the following definitions.

$$\begin{array}{rcl} \mathcal{T}.\ell & = & \{B \mid \exists A \in \mathcal{T}.\ A = [\ell : B, \ldots]^\phi\}. \\ \mathrm{above}_R(\mathcal{T}) & = & \{B \mid \exists A \in \mathcal{T}.\ (A, B) \in R\}. \\ \mathrm{ABOVE}_R(R') & = & \{(\mathrm{above}_R(\{A\}), \mathrm{above}_R(\{B\})) \mid (A, B) \in R'\} \end{array}$$

We define function Var_R such that

- if type A is of the form $[\ldots]^\phi$, then $\mathrm{Var}_R(A) = \phi$;
- $\mathrm{Var}_R(V \oplus V') = 0$;
- if $V \oplus V'$ or $V' \oplus V$ is in R, then $\mathrm{Var}_R(V) = 0$; and
- $\mathrm{Var}_R(\mathcal{T}) = \sqcap\{\mathrm{Var}_R(A) \mid A \in \mathcal{T}\}$,

where $\sqcap$ is the greatest lower bound of a nonempty set of variances; $\sqcap\emptyset$ is undefined.

The types of the above definitions are

$$\begin{array}{rcl} \mathcal{T}.\ell & : & \mathsf{States} \to \mathsf{States} \\ \mathrm{above}_R & : & \mathsf{States} \to \mathsf{States} \\ \mathrm{ABOVE}_R & : & \mathsf{RelTypes} \to \mathsf{RelStates} \\ \mathrm{Var}_R & : & \mathsf{States} \to \mathsf{Variances} \end{array}$$

For any set $\mathcal{T}$ of types we define $\mathrm{LV} : \mathsf{States} \to \mathcal{P}(\mathsf{Labels})$, the labels implied by $\mathcal{T}$, by

$$\mathrm{LV}(\mathcal{T}) = \bigcup_{A \in \mathcal{T}} labs(A(\epsilon))$$

In the rest of the section, we first define the notions of satisfaction-closure (Section 5.1) and satisfaction-consistency (Section 5.2), and we then prove that a C-relation R is solvable if and only if there exists a satisfaction-closed superset which is satisfaction-consistent (Theorem 5.15).

5.1 Satisfaction-closure

Definition 5.1 A C-relation R on types is satisfaction-closed (abbreviated sat-closed) if and only if the following are true:

0 if type A of the form $[\ell : U, \ldots]^\phi$ is in R, then $(A, [\ell : U]^\rightarrow) \in R$.

A if $(A, B), (B, C) \in R$, then $(A, C) \in R$;

B if $(A, B) \in R$, then $(A, A), (B, B) \in R$;

C if $(A, B) \in R$, and $\mathrm{Var}_R(B) = 0$, then $(B, A) \in R$;

D if $(A, [\ell : U]^\rightarrow), (A, [\ell : U']^\rightarrow) \in R$, then $(U, U') \in R$;

E if $(V, [\ell : U]^\rightarrow) \in R$ and $V \oplus V'$ is in R, then $(V \oplus V', [\ell : U]^\rightarrow) \in R$.

F for all $(V \oplus V', [\ell : U]^\rightarrow) \in R$, we have either $(V, [\ell : U]^\rightarrow)$ or $(V', [\ell : U]^\rightarrow)$ in R.

□

Notice that rule **D** is symmetric in the two hypotheses.

Lemma 5.2 *For every solvable C-relation R, there exists a solvable, sat-closed superset R' of R.*

Proof. For a substitution S, define a function

$$
\begin{aligned}
G_S \;&:\; \mathsf{RelTypes} \to \mathsf{RelTypes} && (17)\\
G_S(R) &= R && (18)\\
&\quad \cup \{ (A, [\ell : U]^{\to}) \mid \text{type } A \text{ of the form } [\ell : U, \ldots]^{\phi} \text{ is in } R \} && (19)\\
&\quad \cup \{ (A, C) \mid (A, B), (B, C) \in R \} && (20)\\
&\quad \cup \{ (A, A), (B, B) \mid (A, B) \in R \} && (21)\\
&\quad \cup \{ (B, A) \mid (A, B) \in R \wedge \mathrm{Var}_R(B) = 0 \} && (22)\\
&\quad \cup \{ (U, U') \mid (A, [\ell : U]^{\to}), (A, [\ell : U']^{\to}) \in R \} && (23)\\
&\quad \cup \{ (V \oplus V', [\ell : U]^{\to}) \mid (V, [\ell : U]^{\to}) \in R \ \wedge\ V \oplus V' \text{ is in } R \} && (24)\\
&\quad \cup \{ (V, [\ell : U]^{\to}) \mid (V \oplus V', [\ell : U]^{\to}) \in R \ \wedge\ S(V) \text{ has an } \ell\text{-field} \} && (25)\\
&\quad \cup \{ (V', [\ell : U]^{\to}) \mid (V \oplus V', [\ell : U]^{\to}) \in R \ \wedge\ S(V') \text{ has an } \ell\text{-field} \} && (26)
\end{aligned}
$$

Given a C-relation R with solution S, define R' as follows:

$$R' \;=\; \bigcup_{n=0}^{\infty} G_S^n(R).$$

It is straightforward to show that $R \subseteq R'$ and that R' is sat-closed. It remains to be shown that R' is solvable. It is sufficient to show that $G_S^n(R)$ has solution S, for all n. We proceed by induction on n. In the base of $n = 0$, we have $G_S^0(R) = R$ and that R has solution S by assumption.

In the induction step, suppose $G_S^n(R)$ has solution S. We will now show that $G_S^{n+1}(R) = G_S(G_S^n(R))$ has solution S. We proceed by case analysis on the definition of G_S.

Let $R_n = G_S^n(R)$ and $R_{n+1} = G_S^{n+1}(R)$. We have from the definition of G_S that the constraints in $R_{n+1} \backslash R_n$ belongs to the union of the sets (19) to (26). For each of the sets, we need to show that the constraints in it preserve that S is a solution. In each case, S is preserved because:

(19) Straightforward from the definition of $\leq$.

(20) If $(A, B), (B, C) \in R_n$, then by induction hypothesis, we have $S(A) \leq S(B) \leq S(C)$ and since the $\leq$ is transitive, we have $S(A) \leq S(C)$. Hence, S is a solution to $\{(A, C)\}$.

(21) Since the $\leq$ is reflexive, we have $S(A) \leq S(A)$ and $S(B) \leq S(B)$. Hence, S is a solution to $\{(A, A), (B, B)\}$.

(22) If $(A, B) \in R_n$ and $\mathrm{Var}_{R_n}(B) = 0$, then by induction hypothesis, $S(A) \leq S(B)$ and by definition of $\leq$, we have $S(A) = S(B)$ as well, which implies $S(B) \leq S(A)$. Hence, S is a solution to $\{(B, A)\}$.

(23) If $(A, [\ell : U]^{\rightarrow}), (A, [\ell : U']^{\rightarrow}) \in R_n$, then by induction hypothesis, $S(A) \leq S([\ell : U]^{\rightarrow})$ and $S(A) \leq S([\ell : U']^{\rightarrow})$. By definition of $\leq$, $\exists B$, such that $S(A) = [\ell : B, \ldots]^{\phi}$ and $B = S(U) = S(U')$, which implies $S(U) \leq S(U')$. Hence, S is a solution to $\{(U, U')\}$.

(24) If $(V, [\ell : U]^{\rightarrow}) \in R_n$, then by induction hypothesis, $S(V) \leq S([\ell : U]^{\rightarrow})$. From the definition of $V \oplus V'$, we have $S(V \oplus V').\ell_i = S(V).\ell_i, \forall \ell_i \in \mathrm{LV}(S(V))$. Since $S(V) \leq S([\ell : U]^{\rightarrow})$, we have $S(V \oplus V') \leq S([\ell : U]^{\rightarrow})$. Hence, S is a solution to $\{(V \oplus V', [\ell : U]^{\rightarrow})\}$.

(25) Since $\ell \in \mathrm{LV}(S(V))$, there exists B such that $S(V) = [\ell : B, \ldots]^0$. By definition of $\leq$ and $S(V) \oplus S(V') \leq [\ell : S(U)]^{\rightarrow}$, we have that $B = S(U)$ and $S(V) \leq [\ell : S(U)]^{\rightarrow}$. Therefore, S is a solution to $\{(V, [\ell : U]^{\rightarrow})\}$.

(26) The proof is similar to the previous case.

$\square$

5.2 Satisfaction-consistency

Definition 5.3 A C-relation R on types is satisfaction-consistent (abbreviated sat-consistent) if and only if the following are true:

1. if $([\ell_i : U_i \ {}^{i \in I}]^{\phi}, [\ell_i : U_i' \ {}^{i \in I'}]^{\phi'}) \in R$, then $I \supseteq I'$ and $\phi \sqsubseteq \phi'$;
2. if $([\ell : U, \ldots]^{\phi}, V) \in R$, and $V \oplus V'$ is in R, then $\phi = 0$;
3. if $V \oplus V'$ is in R, then $\mathrm{LV}(\mathrm{above}_R(\{V\})) \cap \mathrm{LV}(\mathrm{above}_R(\{V'\})) = \emptyset$;

$\square$

Lemma 5.4 *If a C-relation R is solvable, then R is sat-consistent.*

Proof. Immediate. $\square$

5.3 Main Result

In this section, we will show that if a C-relation is sat-closed and sat-consistent, then it is solvable.

For a C-relation R we build an automaton with states consisting of sets of types appearing in R, and the following one-step transition function:

$$\delta_R(\mathcal{T})(\ell) = \begin{cases} \text{above}_R(\mathcal{T}.\ell) & \text{if } \mathcal{T}.\ell \neq \emptyset \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We write $\text{States}(R)$ for the set of states of the automaton, and use g, h to range over states.

The one-step transition function is extended to a many-step transition function in the usual way.

$$\begin{array}{rcl} \delta^*_R(g)(\epsilon) & = & g, \\ \delta^*_R(g)(\ell\alpha) & = & \delta^*_R(\delta_R(g)(\ell))(\alpha). \end{array}$$

Any g defines a type, $\text{Type}_R(g)$, and any relation $\mathcal{R}$ on $\text{States}(R)$ defines a constraint set on types $\text{TYPE}_R(\mathcal{R})$, as follows:

$$\begin{array}{rcl} \text{Type}_R(g)(\alpha) & = & (\text{LV}, \text{Var}_R)(\delta^*_R(g)(\alpha)), \\ \text{TYPE}_R(\mathcal{R}) & = & \{(\text{Type}_R(g), \text{Type}_R(h)) \mid (g,h) \in \mathcal{R}\} \end{array}$$

Notice that we use $(\text{LV}, \text{Var}_R)(g)$ to denote $(\text{LV}(g), \text{Var}_R(g))$. We have that

$$\begin{array}{rcl} \text{Type}_R & : & \mathsf{States} \to \mathsf{Types} \\ \text{TYPE}_R & : & \mathsf{RelStates} \to \mathsf{RelTypes} \end{array}$$

Lemma 5.5 *If* $g = \delta_R(g')(\ell)$, *then* $\text{Type}_R(g) = \text{Type}_R(g').\ell$.

Proof.

$$\begin{array}{rcl} (\text{Type}_R(g').\ell)(\alpha) & = & \text{Type}_R(g')(\ell\alpha) \\ & = & (\text{LV}, \text{Var}_R)(\delta^*_R(g')(\ell\alpha)) \\ & = & (\text{LV}, \text{Var}_R)(\delta^*_R(\delta_R(g')(\ell))(\alpha)) \\ & = & (\text{LV}, \text{Var}_R)(\delta^*_R(g)(\alpha)) \\ & = & \text{Type}_R(g)(\alpha). \end{array}$$

□

Definition 5.6 For any C-relation R on types, we define S_R to be the least substitution such that for every U appearing in R we have

$$S_R(U) = \text{Type}_R(\text{above}_R(\{U\})).$$

Note that if $A = [\ell : U, \ldots]^{\phi}$, then $S_R(A) = [\ell : S_R(U), \ldots]^{\phi}$. $\square$

We claim that if R is sat-closed and sat-consistent, then S_R is a solution to R.

To prove this claim, the first step is to develop a connection between subtype-closure and δ. Define the function $\mathcal{A} : \mathsf{RelTypes} \to \mathsf{RelTypes}$ by $(A, B) \in \mathcal{A}(R)$ if and only if one of the following conditions holds:

- $(A, B) \in R$.
- For some ℓ, ϕ, and ϕ', we have $([\ell : A, \ldots]^{\phi}, [\ell : B, \ldots]^{\phi'}) \in R$, or $([\ell : B, \ldots]^{\phi'}, [\ell : A, \ldots]^{\phi}) \in R$.

Note, the subtype-closure (Definition 2.4) of a C-relation R is the least fixed point of $\mathcal{A}$ containing R.

Define the function $\mathcal{B}_R : \mathsf{RelStates} \to \mathsf{RelStates}$ by $(g, h) \in \mathcal{B}_R(\mathcal{R})$, where $g, h \neq \emptyset$, if and only if one of the following conditions holds:

- $(g, h) \in \mathcal{R}$.
- For some ℓ and (g', h') or $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$.

The next four lemmas (Lemma 5.7, 5.8, 5.10, and 5.11) are key ingredients in the proof of Lemma 5.12. Lemma 5.7 states the fundamental relationship between TYPE_R, $\mathcal{A}$, and $\mathcal{B}_R$. We will use the notation

$$f \circ g(x) = f(g(x)).$$

Lemma 5.7 *The following diagram commutes:*

$$\begin{array}{ccc}
\mathsf{RelStates} & \xrightarrow{\text{TYPE}_R} & \mathsf{RelTypes} \\
\Big\downarrow \mathcal{B}_R & & \Big\downarrow \mathcal{A} \\
\mathsf{RelStates} & \xrightarrow{\text{TYPE}_R} & \mathsf{RelTypes}
\end{array}$$

Proof. Suppose $\mathcal{R} \in \mathsf{RelStates}$. To prove $\mathrm{TYPE}_R \circ \mathcal{B}_R \subseteq \mathcal{A} \circ \mathrm{TYPE}_R$, suppose $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$. There must be a pair of states $(g, h) \in \mathcal{B}_R(\mathcal{R})$ such that $A = \mathrm{Type}_R(g)$ and $B = \mathrm{Type}_R(h)$. We reason by cases on how $(g, h) \in \mathcal{B}_R(\mathcal{R})$. From the definition of $\mathcal{B}_R$ we have that there are three cases.

1. suppose $(g, h) \in \mathcal{R}$. We have $(\mathrm{Type}_R(g), \mathrm{Type}_R(h)) \in \mathrm{TYPE}_R(\mathcal{R})$, so from the definition of $\mathcal{A}$ we have $(\mathrm{Type}_R(g), \mathrm{Type}_R(h)) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$.

2. suppose for some ℓ and $(g', h') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. From $(g', h') \in \mathcal{R}$, we have $(\mathrm{Type}_R(g'), \mathrm{Type}_R(h')) \in \mathrm{TYPE}_R(\mathcal{R})$. We have, from Lemma 5.5,

$$(\mathrm{Type}_R(g').\ell)(\alpha) = \mathrm{Type}_R(g)(\alpha) = A(\alpha),$$

   so $\mathrm{Type}_R(g').\ell = A$. Similarly, $\mathrm{Type}_R(h').\ell = B$. From these two observations, and $(\mathrm{Type}_R(g'), \mathrm{Type}_R(h')) \in \mathrm{TYPE}_R(\mathcal{R})$, and the definition of $\mathcal{A}$, we conclude $(A, B) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$.

3. Suppose for some ℓ and $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. The proof is similar to the previous case.

To prove $\mathcal{A} \circ \mathrm{TYPE}_R \subseteq \mathrm{TYPE}_R \circ \mathcal{B}_R$, suppose $(A, B) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$. We reason by cases on how $(A, B) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$. From the definition of $\mathcal{A}$ we have that there are three cases.

1. suppose $(A, B) \in \mathrm{TYPE}_R(\mathcal{R})$. There must exist g and h such that $A = \mathrm{Type}_R(g)$, $B = \mathrm{Type}_R(h)$, and $(g, h) \in \mathcal{R}$. From $(g, h) \in \mathcal{R}$ and the definition of $\mathcal{B}_R$, we have that $(g, h) \in \mathcal{B}_R(\mathcal{R})$, so $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R$.

2. suppose for some ℓ, ϕ, ϕ', we have $([\ell : A, \ldots]^{\phi}, [\ell : B, \ldots]^{\phi'}) \in \mathrm{TYPE}_R(\mathcal{R})$. There must exist g' and h' such that $\mathrm{Type}_R(g') = [\ell : A, \ldots]^{\phi}$, $\mathrm{Type}_R(h') = [\ell : B, \ldots]^{\phi'}$, and $(g', h') \in \mathcal{R}$. Then $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$ are well defined, and $(g, h) \in \mathcal{B}_R(\mathcal{R})$ by the definition of $\mathcal{B}_R$. From $\mathrm{Type}_R(g') = [\ell : A, \ldots]^{\phi}$, $g = \delta_R(g')(\ell)$, and Lemma 5.5, we have $\mathrm{Type}_R(g) = \mathrm{Type}_R(g').\ell = A$. Similarly, $\mathrm{Type}_R(h) = B$, so $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$ as desired.

3. Suppose for some ℓ and $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. The proof is similar to the previous case.

$\square$

Lemma 5.8 *Suppose R is sat-closed. If $(g, h) \in \text{ABOVE}_R(R)$, then $g \supseteq h$.*

Proof. Suppose $(g, h) \in \text{ABOVE}_R(R)$. From the definition of ABOVE_R we have that we can choose A, B such that $(A, B) \in R$, $g = \text{above}_R(\{A\})$, and $h = \text{above}_R(\{B\})$. To prove $g \supseteq h$, suppose $C \in h$. We have $(B, C), (A, B) \in R$. Since R is sat-closed and by closure Rule **A**, we have $(A, C) \in R$ and $C \in g$. Hence, $g \supseteq h$. $\square$

The following lemma reflects that $\leq$ does not support depth subtyping. As a consequence, we have designed the sat-closure rules such that, intuitively, if $(A', B') \in R$ and R is sat-closed, then the types constructed from $\{A'\}$ and $\{B'\}$ have the same ℓ field type.

Lemma 5.9 *If R is sat-closed, $(A', B') \in R$, and* $\text{above}_R(\text{above}_R(\{B'\}).\ell) \neq \emptyset$*, then* $\text{above}_R(\text{above}_R(\{A'\}).\ell) = \text{above}_R(\text{above}_R(\{B'\}).\ell)$.

Proof. From $(A', B') \in R$ and Lemma 5.8, we have $\text{above}_R(\{A'\}) \supseteq \text{above}_R(\{B'\})$, so $\text{above}_R(\text{above}_R(\{A'\}).\ell) \supseteq \text{above}_R(\text{above}_R(\{B'\}).\ell)$.

To prove $\text{above}_R(\text{above}_R(\{A'\}).\ell) \subseteq \text{above}_R(\text{above}_R(\{B'\}).\ell)$, suppose $A \in \text{above}_R(\text{above}_R(\{A'\}).\ell)$. So, there exists $[\ell : U_1, \ldots]^{\phi_1}$ such that

$$\begin{array}{rcl} (A', [\ell : U_1, \ldots]^{\phi_1}) & \in & R \\ (U_1, A) & \in & R. \end{array}$$

From $\text{above}_R(\text{above}_R(\{B'\}).\ell) \neq \emptyset$, we have $B \in \text{above}_R(\text{above}_R(\{B'\}).\ell)$. So, there exists $[\ell : U_2, \ldots]^{\phi_2}$ such that

$$\begin{array}{rcl} (B', [\ell : U_2, \ldots]^{\phi_2}) & \in & R \\ (U_2, B) & \in & R. \end{array}$$

From $(A', B'), (B', [\ell : U_2, \ldots]^{\phi_2}) \in R$, and closure rule A (transitivity), we have $(A', [\ell : U_2, \ldots]^{\phi_2}) \in R$. From

$$\begin{array}{rcl} (A', [\ell : U_1, \ldots]^{\phi_1}) & \in & R \\ (A', [\ell : U_2, \ldots]^{\phi_2}) & \in & R, \end{array}$$

and closure rule $\mathsf{0,A,D}$, we have $(U_2, U_1) \in R$. From $(U_2, U_1), (U_1, A) \in R$ and closure rule A (transitivity), we have $(U_2, A) \in R$. From

$$\begin{array}{rcl} (B', [\ell : U_2, \ldots]^{\phi_2}) & \in & R \\ (U_2, A) & \in & R, \end{array}$$

we have $A \in \text{above}_R(\text{above}_R(\{B'\}).\ell)$.

$\square$

Lemma 5.10 *If* $(g, h) \in (\mathcal{B}_R^n \circ \text{ABOVE}_R(R)) \backslash \text{ABOVE}_R(R)$*, then* $g = h$*,* $\forall n \geq 1$*, where* R *is sat-closed.*

Proof. We proceed by induction on n.

In the base case of $n = 1$, suppose $(g, h) \in (\mathcal{B}_R^1 \circ \text{ABOVE}_R(R)) \backslash \text{ABOVE}_R(R)$. From the definition of $\mathcal{B}_R$, there are two cases.

- Suppose for some ℓ and $(g', h') \in \text{ABOVE}_R(R)$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. By the definition of ABOVE_R, there exist types A', B' such that $g' = \text{above}_R(\{A'\})$, $h' = \text{above}_R(\{B'\})$, and $(A', B') \in R$. We have

$$\begin{array}{rcl} \text{above}_R(\text{above}_R(\{B'\}).\ell) & = & \delta_R(\text{above}_R(\{B'\}))(\ell) \\ & = & \delta_R(h')(\ell) \\ & = & h, \end{array}$$

  and from $(g, h) \in (\mathcal{B}_R^n \circ \text{ABOVE}_R(R))$, and the definition of $\mathcal{B}_R$, we have $h \neq \emptyset$. From $(A', B') \in R$, $\text{above}_R(\text{above}_R(\{B'\}).\ell) \neq \emptyset$, and Lemma 5.9, we have

$$\begin{array}{rcl} g & = & \text{above}_R(\text{above}_R(\{A'\}).\ell) \\ & = & \text{above}_R(\text{above}_R(\{B'\}).\ell) = h. \end{array}$$

- Suppose for some ℓ and $(h', g') \in \text{ABOVE}_R(R)$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. The proof is similar as in the previous case.

In the induction step, suppose

$$(g, h) \in (\mathcal{B}_R^{n+1} \circ \text{ABOVE}_R(R)) \backslash \text{ABOVE}_R(R).$$

From the definition of $\mathcal{B}_R$, there exist ℓ such that (g', h') or $(h', g') \in (\mathcal{B}_R^n \circ \text{ABOVE}_R(R)) \backslash \text{ABOVE}_R(R)$ and $g = \delta_R(g')(\ell), h = \delta_R(h')(\ell)$. From the induction hypothesis, we have $g' = h'$. From the definition of δ_R, it is immediate that $g = h$. $\square$

Lemma 5.11 *Suppose R is sat-closed. If $(g,h) \in \mathrm{ABOVE}_R(R)$, then* $\mathrm{Var}_R(h) = 0 \Rightarrow \mathrm{LV}(g) = \mathrm{LV}(h)$.

Proof. Suppose $(g,h) \in \mathrm{ABOVE}_R(R)$. From the definition of ABOVE_R, $\exists A, B$ such that $g = \mathrm{above}_R(\{A\}), h = \mathrm{above}_R(\{B\})$ and $(A,B) \in R$. Therefore, $\forall A' \in g, B' \in h$, we have $(A,A'),(A,B') \in R$. Since $\mathrm{Var}_R(h) = 0$, there exists a type $B'' \in h$ such that $\mathrm{Var}_R(B'') = 0$. From closure rule **A**, we have that $\mathrm{LV}(\mathrm{above}_R\{A'\}) \subseteq \mathrm{LV}(\mathrm{above}_R\{A\})$; and from closure rule **C**, we have that $\mathrm{LV}(\mathrm{above}_R\{A\}) \subseteq \mathrm{LV}(\mathrm{above}_R\{B''\})$. Hence, $\mathrm{LV}(g) \subseteq \mathrm{LV}(\mathrm{above}_R(\{B''\})) \subseteq \mathrm{LV}(h)$.

From Lemma 5.8, we have $g \supseteq h$ which implies that $\mathrm{LV}(g) \supseteq \mathrm{LV}(h)$. Therefore, $\mathrm{LV}(g) = \mathrm{LV}(h)$. □

Lemma 5.12 *If R is sat-closed, then the subtype-closure of* $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ *is subtype-consistent.*

Proof.

$$\begin{aligned}
&\text{The subtype-closure of } \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)\\
&= \bigcup_{0 \le n < \infty} \mathcal{A}^n \circ \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R) \qquad \text{(Definition of subtype-closure)}\\
&= \bigcup_{0 \le n < \infty} \mathrm{TYPE}_R \circ \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R) \qquad \text{(Lemma 5.7)}\\
&= \bigcup_{0 \le n < \infty} \ \bigcup_{(g,h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)} \{(\mathrm{Type}_R(g), \mathrm{Type}_R(h))\} \ \text{(Definition of } \mathrm{TYPE}_R\text{)}.
\end{aligned}$$

Suppose $(g,h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$. From Lemma 5.8 and Lemma 5.10, and a case analysis on why (g,h) is in $\mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, we have that $g \supseteq h$. From Lemma 5.11 and Lemma 5.10, and a case analysis on why (g,h) is in $\mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, we have that $\mathrm{Var}_R(h) = 0 \Rightarrow \mathrm{LV}(g) = \mathrm{LV}(h)$. Thus, it is immediate from the definition of Type_R that $\{(\mathrm{Type}_R(g), \mathrm{Type}_R(h))\}$ is subtype-consistent.

Thus, the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is the union of a family of subtype-consistent C-relations. Since the union of a family of subtype-consistent C-relations is itself subtype-consistent, we conclude that the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is subtype-consistent. □

The following lemma is a key ingredient in the proof of Lemma 5.14. Lemma 5.14 is the place where it is needed that a relation is satisfaction-consistent.

Lemma 5.13 *If A of the form $[\ell : B, \ldots]^{\phi}$ is in R and R is sat-closed, then*

$$\text{above}_R((\text{above}_R(\{A\})).\ell) = \text{above}_R(\{B\}).$$

Proof. To prove the direction $\supseteq$, notice that from sat-closure rule **B** and A appearing in R, we have $(A, A) \in R$, so $A \in \text{above}_R\{A\}$, hence $B \in (\text{above}_R(\{A\})).\ell$, and thus $\text{above}_R((\text{above}_R(\{A\})).\ell) \supseteq \text{above}_R(\{B\})$.

To prove the direction $\subseteq$, suppose $C \in \text{above}_R((\text{above}_R(\{A\})).\ell)$. From that we have there exists $C' \in (\text{above}_R(\{A\})).\ell$ such that $(C', C) \in R$. From $C' \in (\text{above}_R(\{A\})).\ell$ we have that there exists type D of the form $[\ell : C', \ldots]^{\phi'}$ such that $(A, D) \in R$. Together with closure rule **0**, **A**, **B**, and **D**, we have that $(B, C') \in R$. From transitivity of R (sat-closure rule **A**) and $(B, C'), (C', C) \in R$, we have $(B, C) \in R$, and $C \in \text{above}_R(\{B\})$. □

Lemma 5.14 *If R is sat-closed and sat-consistent, then*

1. *for any type A appearing in R, $S_R(A) = \text{Type}_R \circ \text{above}_R(\{A\})$; and*
2. *$S_R(R) = \text{TYPE}_R \circ \text{ABOVE}_R(R)$.*

Proof. The second property is an immediate consequence of the first property.

To prove the first property, we will, by induction on α, show that for all α, for all A appearing in R, $S_R(A)(\alpha) = \text{Type}_R \circ \text{above}_R(\{A\})(\alpha)$.

If $\alpha = \epsilon$ and A is an ordinary type variable, the result follows by definition of S_R.

If $\alpha = \epsilon$ and A is of the form $V \oplus V'$, $S_R(V) = [\ell_i : B'_i \ {}^{i\in I}]^0, S_R(V') = [\ell_i : B'_i \ {}^{i\in I'}]^0$, and $\text{Type}_R \circ \text{above}_R(\{A\}) = [\ell_i : B_i \ {}^{i\in J}]^0$, we need to show that $J = I \cup I'$, $B_i = B'_i, \forall i \in J$, and $I \cap I' = \emptyset$. From R being sat-closed and closure rules **0**, **E**, we have $\text{LV}(\text{above}_R(\{V, V'\})) \subseteq \text{LV}(\text{above}_R(\{A\}))$. From R being sat-closed and closure rules **0**, **F**, we have $\text{LV}(\text{above}_R(\{A\})) \subseteq \text{LV}(\text{above}_R(\{V, V'\}))$. We conclude $\text{LV}(\text{above}_R(\{A\})) = \text{LV}(\text{above}_R(\{V, V'\}))$. Thus, $J = I \cup I'$ and by sat-consistency rule 3, we have $I \cap I' = \emptyset$. Because of closure rules **0**, **D**, **E**, and **F**, we have that $B_i = B'_i$, $\forall i \in J$.

If $\alpha = \epsilon$ and $A = [\ell_i : B_i \ {}^{i\in\{1..n\}}]^{\phi}$, then $S_R(A)(\alpha) = (\{\ell_i \mid i \in 1..n\}, \phi)$ and $\text{Type}_R \circ \text{above}_R(\{A\})(\alpha) = (\text{LV}(\text{above}_R(\{A\})), \phi)$. From closure rule **B** and A appearing in R, we have $(A, A) \in R$, so $A \in \text{above}_R(\{A\})$. From $A \in \text{above}_R(\{A\})$, we have $\text{LV}(\{A\}) \subseteq \text{LV}(\text{above}_R(\{A\}))$. From $A \in \text{above}_R(\{A\})$ and sat-consistency rules 1 and 2, we have $\text{LV}(\text{above}_R(\{A\})) \subseteq$

LV($\{A\}$). We conclude LV($\{A\}$) = LV(above$_R$($\{A\}$)). From the definition of Var$_R$, we have that $\mathrm{Var}_R(A) = \phi$. By sat-consistency rule 1, we have $\mathrm{Var}_R(\mathrm{above}_R(\{A\}) = \phi$, as desired.

If $\alpha = \ell\alpha'$ and A is a type variable, the result follows by definition of S_R.

If $\alpha = \ell\alpha'$ and A is of the form $V \oplus V'$, then either $S_R(V)$ or $S_R(V')$ has an ℓ field. Suppose it is $S_R(V)$ that has an ℓ field:

$$\begin{array}{lcll} S_R(A)(\alpha) & = & (S_R(V) \oplus S_R(V'))(\alpha) & \text{(Definition of } S_R\text{)} \\ & = & S_R(V)(\alpha) & (S_R(V) \text{ has an } \ell \text{ field)} \\ & = & \mathrm{Type}_R \circ \mathrm{above}_R(\{V\})(\alpha) & \text{(Definition of } S_R\text{)} \\ & = & \mathrm{Type}_R \circ \mathrm{above}_R(\{V, V'\})(\alpha) & (S_R(V') \text{ has no } \ell \text{ field)} \\ & = & \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha). & \text{(from the proof of the base case)} \end{array}$$

The case where it is $S_R(V')$ that has an ℓ field is similar, we omit the details.

If $\alpha = \ell\alpha'$ and $A = [\ell : B, \ldots]^\phi$, then

$$\begin{array}{l} S_R(A)(\alpha) \\ = S_R(B)(\alpha') \qquad \text{(Definition of } S_R\text{)} \\ = \mathrm{Type}_R \circ \mathrm{above}_R(\{B\})(\alpha') \qquad \text{(Induction hypothesis)} \\ = (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\mathrm{above}_R(\{B\}))(\alpha')) \qquad \text{(Definition of Type}_R\text{)} \\ = (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell))(\alpha')) \qquad \text{(Lemma 5.13)} \\ = (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\delta_R(\mathrm{above}_R(\{A\}))(\ell))(\alpha')) \qquad \text{(Definition of } \delta_R\text{)} \\ = (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\mathrm{above}_R(\{A\}))(\ell\alpha')) \qquad \text{(Definition of } \delta_R^*\text{)} \\ = \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha) \qquad \text{(Definition of Type}_R \text{ and } \alpha = \ell\alpha'\text{).} \end{array}$$

If $\alpha = \ell\alpha'$ and A is a record without an ℓ field, then $S_R(A)(\alpha)$ is undefined. By sat-consistency rule 1, no $C \in \mathrm{above}_R(\{A\})$ has an ℓ field, so from the definition of Type$_R$ we have that $\mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\ell\alpha')$ is undefined, as desired.

□

Theorem 5.15 *R is solvable if and only if there exists a sat-closed superset R' of R, such that R' is sat-consistent.*

Proof. If R is solvable, then we have from Lemma 5.2 that there exists solvable, sat-closed superset R' of R, so from Lemma 5.4, we have that R' is sat-consistent.

Conversely, let R' be a sat-closed superset of R, and assume that R' is sat-consistent. From Lemma 5.12 and Lemma 5.14, we have that the subtype-closure of $S_{R'}(R')$ is subtype-consistent. From the subtype-closure of $S_{R'}(R')$ being subtype-consistent and Lemma 2.6, we have $A \leq B$ for every $(A, B) \in S_{R'}(R')$, so $S_{R'}(A') \leq S_{R'}(B')$ for every $(A', B') \in R'$, and hence R' has solution $S_{R'}$. From $R \subseteq R'$ and that R' is solvable, we have that R is solvable. $\square$

Theorem 5.16 *The type inference problem is in NP.*

Proof. From Theorem 4.6 we have the type inference problem is polynomial-time reducible to the constraint problem. To solve a constraint set R generated from a program a, we first guess a superset R' of R. Notice that we only need to consider an R' which has a size which is polynomial in the size of a. Next we check that R' is sat-closed and sat-consistent. It is straightforward to see that this can be done in polynomial time. $\square$

6 NP-hardness

In this section we prove that the type inference problem is NP-hard. We do this in two steps. First we prove that solvability of so-called simple constraints can be reduced to the type inference problem, and then we prove that solving simple constraints is NP-hard.

6.1 From Constraints to Types

For any ς-term c, the the constraint set $\mathcal{C}(c)$ is defined as follows.

Definition 6.1 Given a denumerable set of variables, a *simple* constraint set is a finite set of constraints of the forms

$$\begin{array}{rcl} (V & , & [\ell_i : V_i \ ^{i \in 1..n}]^0) \\ (V \oplus V' & , & [\ell_i : V_i \ ^{i \in 1..n}]^0) \end{array}$$

where $V, V', V_1, \ldots, V_n$ are variables. □

Lemma 6.2 *Solvability of simple constraint sets is polynomial-time reducible to the type inference problem.*

Proof. Let $\mathcal{C}$ be a simple constraint set. Define

$$\begin{array}{lllll}
a^{\mathcal{C}} = & [& \ell_V & = & \varsigma(x)(x.\ell_V) \\
 & & & & \text{for each variable } V \text{ in } \mathcal{C} \\
 & & \ell_Q & = & \varsigma(x)[\ell_i = \varsigma(y)(x.\ell_{V_i}) \ ^{i \in 1..n}] \\
 & & & & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
 & & m_{Q,\ell_j} & = & \varsigma(x)((x.\ell_{V_j} \Leftarrow \varsigma(y)(x.\ell_Q.\ell_j)).\ell_Q) \\
 & & & & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
 & & & & \text{and for each } j \in 1..n \\
 & & k_Q & = & \varsigma(x)(x.\ell_Q + [\,]) \\
 & & & & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
 & & \ell_{(V,Q)} & = & \varsigma(x)((x.\ell_Q \Leftarrow \varsigma(y)(x.\ell_V)).\ell_V) \\
 & & & & \text{for each constraint } (V, Q) \text{ in } \mathcal{C} \\
 & & & & \text{where } Q \text{ is of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
 & & \ell_{(V \oplus V',Q)} & = & \varsigma(x)((x.\ell_Q \Leftarrow \varsigma(y)(x.\ell_V + x.\ell_{V'})).\ell_Q) \\
 & & & & \text{for each constraint } (V \oplus V', Q) \text{ in } \mathcal{C} \\
 & & & & \text{where } Q \text{ is of the forms } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
 &] & & &
\end{array}$$

Notice that $a^{\mathcal{C}}$ can be generated in polynomial time.

We first prove that if $\mathcal{C}$ is solvable then $a^{\mathcal{C}}$ is typable. Suppose $\mathcal{C}$ has solution S. Define

$$
\begin{array}{lllll}
A = & [& \ell_V & : S(V) & \text{for each variable } V \text{ in } \mathcal{C} \\
& & \ell_Q & : S(Q) & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
& & m_{Q,\ell_j} & : S(Q) & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
& & & & \text{and for each } j \in 1..n \\
& & k_Q & : S(Q) & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
& & \ell_{V \leq Q} & : S(V) & \text{for each constraint } (V, Q) \text{ in } \mathcal{C} \\
& & & & \text{where } Q \text{ is of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
& & \ell_{V \oplus V' \leq Q} & : S(Q) & \text{for each constraint } (V \oplus V', Q) \text{ in } \mathcal{C} \\
& & & & \text{where } Q \text{ is of the form } [\ell_i : V_i \ ^{i \in 1..n}]^0 \\
&]^0 & & &
\end{array}
$$

Clearly $\emptyset \vdash a^{\mathcal{C}} : A$ is derivable.

We now prove that if $a^{\mathcal{C}}$ is typable, then $\mathcal{C}$ is solvable. Suppose $a^{\mathcal{C}}$ is typable. From Theorem 4.6 we get a solution S of $\mathcal{C}(a^{\mathcal{C}})$. Notice that each method in $a^{\mathcal{C}}$ binds a variable x. Each of these variables corresponds to a distinct type variable in $\mathcal{C}(a^{\mathcal{C}})$. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, and $\mathcal{C}(a^{\mathcal{C}})$ contains constraints of the form $U_x = [\ldots]^0$ for each method in $a^{\mathcal{C}}$ (from rule (12)), all those type variables are mapped by S to the same type. Thus, we can think of all the bound variables of methods of $a^{\mathcal{C}}$ as being related to the same type variable, which we will write as U_x.

The solution S has the following two properties.

- **Property 1** If V is a variables in $\mathcal{C}$, then $S(U_x)\downarrow\ell_V$ is defined.
- **Property 2** For each Q in $\mathcal{C}$ of the form $[\ell_i : V_i \ ^{i \in 1..n}]^0$, we have $S(U_x)\downarrow\ell_Q = [\ell_i : (S(U_x)\downarrow\ell_{V_i}) \ ^{i \in 1..n}]^0$.

To see Property 1, notice that in the body of the method ℓ_V we have the expression $x.\ell_V$. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (8) and (9) that S satisfies

$$(U_x, V_x) \text{ and } (V_x, [\ell_V : U_{x.\ell_V}]).$$

We conclude that $S(U_x)\downarrow\ell_V = S(U_{x.\ell_V})$ is defined.

To see Property 2, let Q be an occurrence in $\mathcal{C}$ of the form $[\ell_i : V_i \ ^{i \in 1..n}]^0$. For each $j \in 1..n$, in the body of the method m_{Q,ℓ_j}, we have the expression $x'.\ell_{V_j} \Leftarrow \varsigma(y)(x.\ell_Q.\ell_j)$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (8), (15), (8), (9), (10), (9), and (10) that S satisfies

$$
\begin{array}{rcll}
(U_x & , & V_{x'}) \text{ and } (V_{x'},\ [\ell_{V_j} : V_{x.\ell_Q.\ell_j}]^{\rightarrow}) & (27)\\
(U_x & , & V_x) \text{ and } (V_x,\ [\ell_Q : U_{x.\ell_Q}]^{\rightarrow}) & (28)\\
(U_{x.\ell_Q} & , & V_{x.\ell_Q}) & (29)\\
(V_{x.\ell_Q} & , & [\ell_j : U_{x.\ell_Q.\ell_j}]^{\rightarrow}) & (30)\\
(U_{x.\ell_Q.\ell_j} & , & V_{x.\ell_Q.\ell_j}) & (31)
\end{array}
$$

Thus,

$$
\begin{array}{rcll}
S(U_x){\downarrow}\ell_Q & = & S(U_{x.\ell_Q}) & \text{from (28) and Lemma 2.2}\\
& \leq & S(V_{x.\ell_Q}) & \text{from (29)}\\
& \leq & [\ell_j : S(U_{x.\ell_Q.\ell_j})]^{\rightarrow} & \text{from (30)}\\
S(U_x){\downarrow}\ell_Q{\downarrow}\ell_j & = & S(U_{x.\ell_Q.\ell_j}) & \text{from Lemma 2.2}\\
& \leq & S(V_{x.\ell_Q.\ell_j}) & \text{from (31)}\\
& = & S(U_x{\downarrow}\ell_{V_j}) & \text{from (27) and Lemma 2.2}
\end{array}
$$

In the body of the method k_Q, we have the expression $(x.\ell_Q + [\ ])$. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (8), (9), (10), and (16) that S satisfies

$$
\begin{array}{rcll}
(U_x & , & V_x) \text{ and } (V_x,\ [\ell_Q : U_{x.\ell_Q}]^{\rightarrow}) & (32)\\
(U_{x.\ell_Q} & , & V_{x.\ell_Q}) & (33)\\
(V_{x.\ell_Q} \oplus V_{[\ ]} & , & V_{x.\ell_Q+[\ ]}) & (34)
\end{array}
$$

Thus, from (32), Lemma 2.2, (33), (34) and the definition of $\oplus$, we have

$$S(U_x){\downarrow}\ell_Q = S(U_{x.\ell_Q}) \leq S(V_{x.\ell_Q}) = [\ldots]^0. \tag{35}$$

In the body of the method ℓ_Q we have the expression $[\ell_i = \varsigma(y)(x.\ell_{V_i})\ \ ^{i\in 1..n}]$. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (8), (9), (10), (11) and (12) that S satisfies

$$
\begin{array}{rcll}
\forall j \in 1..n,\ (U_x & , & V_x) \text{ and } (V_x,\ [\ell_{V_j} : U_{x.\ell_{V_j}}]^{\rightarrow}) & (36)\\
(U_{x.\ell_{V_j}} & , & V_{x.\ell_{V_j}}) & (37)\\
([\ell_i^0 : V_{x.\ell_{V_i}}\ \ ^{i\in 1..n}]^0 & , & V_{[\ell_i=\varsigma(y)(x.\ell_{V_i})\ \ ^{i\in 1..n}]}) & (38)\\
U_x & \equiv & [\ldots \ell_Q : V_{[\ell_i=\varsigma(y)(x.\ell_{V_i})\ \ ^{i\in 1..n}]} \ldots]^0 & (39)
\end{array}
$$

Thus, from (38) and (39), we have

$$[\ell_i : S(V_{x.\ell_{V_i}})\ ^{i\in 1..n}]^0 \leq S(V_{[\ell_i=\varsigma(y)(x.\ell_{V_i})\ ^{i\in 1..n}]}) = S(U_x){\downarrow}\ell_Q$$

and together with (36), Lemma 2.2 and (37), we have

$$\forall j \in 1..n,\ \ S(U_x){\downarrow}\ell_{V_j} = S(U_{x.\ell_{V_j}}) \leq S(V_{x.\ell_{V_j}}) = S(U_x){\downarrow}\ell_Q{\downarrow}\ell_j.$$

Since we have both

$$\begin{array}{rcl} S(U_x){\downarrow}\ell_Q{\downarrow}\ell_j & \leq & S(U_x){\downarrow}\ell_{V_j} \text{ and} \\ S(U_x){\downarrow}\ell_Q{\downarrow}\ell_j & \geq & S(U_x){\downarrow}\ell_{V_j}, \end{array}$$

we have

$$S(U_x){\downarrow}\ell_Q{\downarrow}\ell_j \quad = \quad S(U_x){\downarrow}\ell_{V_j} \tag{40}$$

and together (40) and (35) give that $S(U_x){\downarrow}\ell_Q = [\ell_i : S(U_x){\downarrow}\ell_{V_j}{}^{i\in 1..n}]^0$, that is, Property 2.

From Property 1 we have that we can define

$$S_{\mathcal{C}}(V) \quad = \quad S(U_x){\downarrow}\ell_V \qquad \text{for each variable } V \text{ in } \mathcal{C}. \tag{41}$$

With this definition, we can restate Property 2 as

$$S_{\mathcal{C}}(Q) \quad = \quad S(U_x){\downarrow}\ell_Q \qquad \text{where } Q = [\ell_i : V_i\ ^{i\in 1..n}]^0. \tag{42}$$

We will now show that $\mathcal{C}$ has solution $S_{\mathcal{C}}$.

Consider first a constraint (V, Q) in $\mathcal{C}$, where $Q = [\ell_i : V_i\ ^{i\in 1..n}]^0$. The body of the method $\ell_{(V,Q)}$ contains the expression $x'.\ell_Q \Leftarrow \varsigma(y)(x.\ell_V)$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (8), (15), (8), (9), and (10) that S satisfies

$$(U_x \quad , \quad V_{x'}) \ \text{ and } \ (V_{x'},\ [\ell_Q : V_{x.\ell_V}]^{\rightarrow}) \tag{43}$$

$$(U_x \quad , \quad V_x) \ \text{ and } \ (V_x,\ [\ell_V : U_{x.\ell_V}]^{\rightarrow}) \tag{44}$$

$$(U_{x.\ell_V} \quad , \quad V_{x.\ell_V}) \tag{45}$$

We conclude

$$
\begin{array}{rcll}
S_{\mathcal{C}}(V) & = & S(U_x)\downarrow \ell_V & \text{from (41)} \\
& = & S(U_{x.\ell_V}) & \text{from (44) and Lemma 2.2} \\
& \leq & S(V_{x.\ell_V}) & \text{from (45)} \\
& = & S(U_x)\downarrow \ell_Q & \text{from (43) and Lemma 2.2} \\
& = & S_{\mathcal{C}}(Q) & \text{from (42).}
\end{array}
$$

Consider next a constraint $(V \oplus V', Q)$ in $\mathcal{C}$, where Q is of the form $[\ell_i : V_i \ ^{i \in 1..n}]^0$. The body of the method $\ell_{(V \oplus V', Q)}$ contains the expression $x'.\ell_Q \Leftarrow \varsigma(y)(x.\ell_V + x.\ell_{V'})$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (8), (15), (8), (9), (10), and (16) that S satisfies

$$
\begin{array}{rcl}
(U_x & , & V_{x'}) \text{ and } (V_{x'},\, [\ell_Q : V_{x.\ell_V + x.\ell_{V'}}]^{\rightarrow}) \\
(U_x & , & V_x) \text{ and } (V_x,\, [\ell_V : U_{x.\ell_V}]^{\rightarrow}) \\
(U_x & , & V_x) \text{ and } (V_x,\, [\ell_{V'} : U_{x.\ell_{V'}}]^{\rightarrow}) \\
(U_{x.\ell_V} & , & V_{x.\ell_V}) \\
(U_{x.\ell_{V'}} & , & V_{x.\ell_{V'}}) \\
(V_{x.\ell_V} \oplus V_{x.\ell_{V'}} & , & V_{x.\ell_V + x.\ell_{V'}})
\end{array}
\begin{array}{r}
(46) \\ (47) \\ (48) \\ (49) \\ (50) \\ (51)
\end{array}
$$

We conclude

$$
\begin{array}{rcll}
S_{\mathcal{C}}(V) & = & S(U_x)\downarrow \ell_V & \text{from (41)} \\
& = & S(U_{x.\ell_V}) & \text{from (47) and Lemma 2.2} \\
& = & S(V_{x.\ell_V}) & \text{from (49) and (51)} \\
S_{\mathcal{C}}(V') & = & S(U_x)\downarrow \ell_{V'} & \text{from (41)} \\
& = & S(U_{x.\ell_{V'}}) & \text{from (48) and Lemma 2.2} \\
& = & S(V_{x.\ell_{V'}}) & \text{from (50) and (51)} \\
S_{\mathcal{C}}(V) \oplus S_{\mathcal{C}}(V') & = & S(V_{x.\ell_V}) \oplus S(V_{x.\ell_{V'}}) & \text{from above} \\
& \leq & S(V_{x.\ell_V + x.\ell_{V'}}) & \text{from (51)} \\
& = & S(U_x)\downarrow \ell_Q & \text{from (46) and Lemma 2.2} \\
& = & S_{\mathcal{C}}(Q) & \text{from (42).}
\end{array}
$$

□

6.2 Solving Simple Constraints is NP-hard

In this section we show that solving simple constraint systems is NP-hard.

Suppose we are given a Boolean expression

$$\psi = \bigwedge_{i=1}^{n} (l_{i1} \vee l_{i2} \vee l_{i3})$$

where X_ψ is the set of variables occurring in ψ, and each literal l_{ij} is of the form x or $\bar{x}$, where $x \in X_\psi$. We will use the notation I_x for the set of positions (ij) for which $l_{ij} = x$ or $l_{ij} = \bar{x}$. Furthermore, if $l_{ij} = x$ or $l_{ij} = \bar{x}$, then we use I_{ij} to denote I_x. We will use the abbreviations

$$\mathsf{False} = [\ ]^0 \qquad \mathsf{True} = [q : [\ ]^0]^0.$$

Their only significance is that $\mathsf{False} \neq \mathsf{True}$. We will construct a simple constraint system $\mathcal{C}_\psi$ over the variables

$$\begin{array}{rl} & \{\ U_x, U_{\bar{x}}, V_x, V_{\bar{x}}, T_x, T_{\bar{x}}, R_x \ \mid \ x \in X_\psi\ \} \\ \cup & \{\ P_{ij} \ \mid \ i \in 1..n, j \in 1..3\} \\ \cup & \{\ A_{ij} \ \mid \ i \in 1..n, j \in 0..3\}. \end{array}$$

The constraint system $\mathcal{C}_\psi$ consists of:

- for each $x \in X_\psi$, the constraints

$$\begin{array}{rcl} (U_x \oplus U_{\bar{x}} & , & [k : R_x]^0) \\ (U_x \oplus T_x & , & [k : V_x]^0) \\ (U_{\bar{x}} \oplus T_{\bar{x}} & , & [k : V_{\bar{x}}]^0) \\ (R_x & , & [m_{ij} : A_{ij}{}^{(ij)\in I_x}]^0) \\ (V_x \oplus V_{\bar{x}} & , & [m_{ij} : A_{ij}{}^{(ij)\in I_x}]^0) \end{array} \begin{array}{r} (52) \\ (53) \\ (54) \\ (55) \\ (56) \end{array}$$

- for all $i \in 1..n$ and for all $j \in 1..3$, the constraints:

$$(V_{l_{ij}} \oplus P_{ij} \quad , \quad [m_{ij} : A_{i(j-1)}, \quad m_{i'j'} : A_{i'j'}{}^{(i'j')\in I_{ij}\setminus(ij)}]^0) \qquad (57)$$

- for all $i \in 1..n$, the constraints:

$$\begin{array}{rcl} (A_{i0} & , & \mathsf{False}) \\ (A_{i3} & , & \mathsf{True}). \end{array}$$

In the last constraint, we use the abbreviation (A_{i3}, True) to denote the two constraints $(A_{i3}, [q : B]^0)$, $(B, [\ ]^0)$, where B is a fresh variable.

Lemma 6.3 *Solving simple constraint systems is NP-hard.*

Proof. Given that 3-SAT is NP-hard, it is sufficient to show that ψ is satisfiable if and only if $\mathcal{C}_\psi$ is solvable.

Suppose first that ψ has solution f. Here is a mapping S_f from the variables of $\mathcal{C}_\psi$ to types. If $f(x)$ is true, then we have:

v	$S_f(v)$
U_x	$[\ ]^0$
$U_{\bar{x}}$	$[k : S_f(R_x)]^0$
V_x	$[\ ]^0$
$V_{\bar{x}}$	$S_f(R_x)$
T_x	$[k : [\ ]^0]^0$
$T_{\bar{x}}$	$[\ ]^0$
R_x	$[m_{ij} : S_f(A_{ij})\ ^{(ij)\in I_x}]^0$.

If $f(x)$ is false, then we have:

v	$S_f(v)$
U_x	$[k : S_f(R_x)]^0$
$U_{\bar{x}}$	$[\ ]^0$
V_x	$S_f(R_x)$
$V_{\bar{x}}$	$[\ ]^0$
T_x	$[\ ]^0$
$T_{\bar{x}}$	$[k : [\ ]^0]^0$
R_x	$[m_{ij} : S_f(A_{ij})\ ^{(ij)\in I_x}]^0$.

For $i \in 1..n$ and $j \in 1..3$, define

$$S_f(P_{ij}) = \begin{cases} [m_{ij} : S_f(A_{i(j-1)}),\quad m_{i'j'} : S_f(A_{i'j'})\ ^{(i'j')\in I_{ij}\setminus(ij)}]^0 & f(l_{ij}) \text{ is true} \\ [\ ]^0 & \text{otherwise.} \end{cases}$$

Define the function g from Booleans to $\{\mathsf{False}, \mathsf{True}\}$ such that $g(\text{false}) = \mathsf{False}$ and $g(\text{true}) = \mathsf{True}$. For $i \in 1..n$,

v	$S_f(v)$
A_{i0}	False
A_{i1}	$g \circ f(l_{i1})$
A_{i2}	$g \circ f(l_{i1} \vee l_{i2})$
A_{i3}	True.

It is straightforward to check that S_f is a solution to the constraints in $\mathcal{C}_\psi$ of the forms (52)–(56), we omit the details. Here we will focus on showing that S_f is a solution to the constraints in $\mathcal{C}_\psi$ of the form (57). Suppose we are given $i \in 1..n$ and $j \in 1..3$. There are two cases. First, if $f(l_{ij})$ is true, then $S_f(P_{ij}) = [m_{ij} : S_f(A_{i(j-1)}),\ \ m_{i'j'} : S_f(A_{i'j'})\ ^{(i'j') \in I_{ij} \setminus (ij)}]^0$ and $S_f(V_{l_{ij}}) = [\ ]^0$. Hence, the constraint (57) is satisfied.

Second, if $f(l_{ij})$ is false, then $S_f(P_{ij}) = [\ ]^0$ and $S_f(V_{l_{ij}}) = S_f(R_{l_{ij}})$. Hence, we must show that $S_f(A_{ij}) = S_f(A_{i(j-1)})$. There are three cases.

- If $j = 1$, then $S_f(A_{i1}) = g \circ f(l_{i1}) = g(\text{false}) = \mathsf{False} = S_f(A_{i0})$.
- If $j = 2$, then $S_f(A_{i2}) = g \circ f(l_{i1} \vee l_{i2}) = g \circ (f(l_{i1}) \vee f(l_{i2})) = g \circ (f(l_{i1}) \vee \text{false}) = g \circ f(l_{i1}) = S_f(A_{i1})$.
- If $j = 3$, then $S_f(A_{i3}) = \mathsf{True}$. Since ψ is satisfiable and $f(l_{i3})$ is false, we have that $f(l_{i1} \vee l_{i2})$ is true, so $S_f(A_{i2}) = g \circ f(l_{i1} \vee l_{i2}) = g(\text{true}) = \mathsf{True}$. We conclude that $S_f(A_{i3}) = \mathsf{True} = S_f(A_{i2})$.

Conversely, suppose S is a solution to $\mathcal{C}_\psi$.

Property 1: For every $x \in X_\psi$, we have either $S(V_x) = S(R_x)$ and $S(V_{\bar{x}}) = [\ ]^0$, or we have $S(V_x) = [\ ]^0$ and $S(V_{\bar{x}}) = S(R_x)$.

To prove Property 1, notice that from (52) we have exactly one of $S(U_x) = [k : S(R_x)]^{\rightarrow}$ or $S(U_{\bar{x}}) = [k : S(R_x)]^{\rightarrow}$. From that and (53)–(54), we have that either $S(V_x) = S(R_x)$ or $S(V_{\bar{x}}) = S(R_x)$. From that and (56) we get Property 1.

Define

$$f_S(x) = \begin{cases} \text{false} & S(V_x) = S(R_x) \\ \text{true} & \text{otherwise.} \end{cases}$$

Going for a contradiction, let us suppose that f_S does not satisfy ψ. That means that must exist i such that, for all $j \in 1..3$, $f_S(l_{ij}) = \text{false}$. From the definition of f_S and Property 1 we have that, for $j \in 1..3$, there is a variable x such that $(ij) \in I_x$ and $S(V_{l_{ij}}) = S(R_x)$. From that and (55) and (57), we conclude

$$\mathsf{False} = S(A_{i0}) = S(A_{i1}) = S(A_{i2}) = S(A_{i3}) = \mathsf{True},$$

a contradiction. □

Theorem 6.4 *The type inference problem is NP-complete.*

Proof. We have that type inference is in NP from Theorem 5.16. NP-hardness follows from Lemma 6.2 and Lemma 6.3. □

7 Conclusion

Type inference with record concatenation, subtyping, and recursive types is NP-complete. Future work includes implementing the algorithm for a language such as Obliq, and to attempt to combine our technique with our algorithm for type inference with both covariant and invariant fields [16].

The construction used in our NP-hardness proof may be applicable to other types systems. In particular, our notion of simple constraint systems may be reducible to even more restrictive type inference problems than the one we have considered.

Acknowledgments. A preliminary version of this paper was presented at LICS'02, IEEE Symposium on Logic in Computer Science, July 2002. In that version of the paper, we mistakenly claimed that the type inference problem can be solved in polynomial time. Two reviewers for Information and Computation spotted a problem in the proof of a crucial lemma. The lemma was indeed false and, as a consequence, we realized that the type inference problem is NP-complete, and not polynomial. We thank the reviewers for identifying that problem and for numerous other helpful comments.

Our work is supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265.

A Proof of Lemma 2.3

Here we give a full proof that $\leq$ is a partial order.

First, $\leq$ is reflexive because the identity on $\mathcal{T}(\Sigma)$ is a simulation.

Lemma A.1 *If R is a reflexive simulation, then $(R \circ R)$ is a simulation.*

Proof. Suppose $(A, A') \in (R \circ R)$. Then there is an A'' such that $(A, A'') \in R$ and $(A'', A') \in R$.

- If $A' = U$, then $A'' = U$ because $(A'', A') \in R$; and then $A = U$ because $(A, A'') \in R$.
- Similarly, if $A = U$, then $A' = U$.
- Otherwise $A' = [\ell : B'^{i \in I'}]^{\phi'}$. Then since R is a simulation, we have

- $A'' = [\ell_i : B''_i \ ^{i \in I''}]^{\phi''}$,
- $A = [\ell_i : B_i \ ^{i \in I}]^{\phi}$,
- $\phi \sqsubseteq \phi'' \sqsubseteq \phi'$,
- $(B_i, B''_i), (B''_i, B'_i) \in R \Rightarrow (B_i, B'_i) \in R \circ R$, and
- $(B'_i, B''_i), (B''_i, B_i) \in R \Rightarrow (B'_i, B_i) \in R \circ R$.

Since $\sqsubseteq$ is transitive we have $\phi \sqsubseteq \phi'$.

If $\phi = \phi' = 0$, then $I = I'' = I'$ as desired.

□

Corollary A.2 $\leq$ *is transitive.*

Proof. Just note that $\leq$ is reflexive, and $\leq \supseteq (\leq \circ \leq)$ because $\leq$ is the largest simulation. □

Lemma A.3 *Every simulation is antisymmetric.*

Proof. Let R be a simulation. We prove the following statement by induction on α:

If $(A, A') \in R$ and $(A', A) \in R$, then $A = A'$, that is, $A(\alpha) = A'(\alpha)$ for every α.

- If $\alpha = \epsilon$, we show $A(\alpha) = A'(\alpha)$ by cases on the structure of A.
  - If $A = U$, then by the definition of simulation, $A' = U$. Therefore $A(\alpha) = U = A'(\alpha)$.
  - If A is a record type, then by the definition of simulation and the antisymmetry of $\sqsubseteq$, A' is a record type with exactly the same labels and variances; that is, $A = [\ell_i : B_i \ ^{i \in I}]^{\phi}$ and $A' = [\ell_i : B'_i \ ^{i \in I}]^{\phi}$. Therefore $A(\alpha) = (\{\ell_i : i \in 1..n\}, \phi) = A'(\alpha)$ as desired.
- If $\alpha = \ell\alpha'$, we consider two cases.
  - If $A(\ell)$ is undefined, then either $A = U$ for some U, or A is a record type with no ℓ field. In the first case, $A' = U$ because $(A', A) \in R$. In the second case, A' has no ℓ field (otherwise $(A, A') \in R$ would imply A has an ℓ field, contradiction). In either case, $A'(\ell)$ is undefined, so both $A(\alpha)$ and $A'(\alpha)$ are undefined.

– If $A = [\ell : B, \ldots]^\phi$, then by the definition of simulation and the antisymmetry of $\sqsubseteq$, we have $A' = [\ell : B', \ldots]^\phi$ and $(B, B'), (B', B) \in R$. Then by induction, $B(\alpha') = B'(\alpha')$. So $A(\alpha) = B(\alpha') = B'(\alpha') = A'(\alpha)$ as desired.

$\square$

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proceedings of the 19th ACM Symposium on Principles of Programming Languages.

[3] Alan H. Borning and Daniel H. H. Ingalls. Multiple inheritance in Smalltalk80. In *Proceedings of the Second National Conference on Artificial Intelligence*, 1982.

[4] Luca Cardelli. A language with distributed scope. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, 1995.

[5] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(1):95–169, 1983.

[6] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Proceedings of CC'02, International Conference on Compiler Construction*, pages 159–178. Springer-Verlag (*LNCS* 2304), 2002.

[7] Neal Glew. An efficient class and object encoding. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 311–324, Minneapolis, Minnesota, October 2000.

[8] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages, Orlando*, pages 131–142. ACM, 1991.

[9] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. Preliminary version in Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[10] Robin Milner. Operational and algebraic semantics of concurrent processes. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. The MIT Press, New York, 1990.

[11] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of the Second International Parallel Architectures and Languages Europe Conference*, pages 357–373, April 1989.

[12] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proceedings of the Nineth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

[13] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Proceedings of the 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.

[14] Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5):519–527, September 1996.

[15] Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[16] Jens Palsberg, Tian Zhao, and Trevor Jim. Automatic discovery of covariant read-only fields. In *Proceedings of the Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, January 2002.

[17] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science*, pages 376–385, 1993.

[18] Francois Pottier. Simplifying subtyping constraints. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 1996.

[19] Francois Pottier. A 3-part type inference engine. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming*, volume 1782, pages 320–335. Springer Verlag, 2000.

[20] Didier Remy. Typing record concatenation for free. In *ACM Symposium on Principles of Programming Languages*, pages 166–176, 1992.

[21] Mark Shields and Erik Meijer. Type-indexed rows. In *Symposium on Principles of Programming Languages*, pages 261–275, 2001.

[22] Bjarne Stroustrup. *A History of C++: 1979–1991.* 1993. Manuscript.

[23] Martin Sulzmann. Designing Record Systems. Research Report YALEU/DCS/RR-1128, Yale University, Department of Computer Science, April 1997.

[24] Hideki Tsuiki. On typed calculi with a merge operator. In *Foundations of Software Technology and Theoretical Computer Science*, pages 101–112, 1994.

[25] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.

[26] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[27] Jan Zwanenburg. Record concatenation with intersection types. Technical Report 95–34, Eindhoven University of Technology, 1995.

[28] Jan Zwanenburg. A type system for record concatenation and subtyping. In Kim Bruce and Giuseppe Longo, editors, *Informal proceedings of Third Workshop on Foundations of Object-Oriented Languages*, 1996.