Automatic Discovery of Covariant Read-Only Fields

Jens Palsberg
Purdue University*

Tian Zhao
Purdue University†

Trevor Jim
AT&T Labs Research‡

December 10, 2001

Abstract

Read-only fields are useful in object calculi, pi calculi, and statically-typed intermediate languages because they admit covariant subtyping, unlike updateable fields. For example, Glew's translation of classes and objects to an intermediate calculus places the method tables of classes into read-only fields; covariant subtyping on the method tables is required to ensure that subclasses are translated to subtypes. In programs that use updateable fields, read-only fields can be either specified or discovered. For both cases we show that type inference is equivalent to solving type constraints, which in turn is P-complete and computable in $O(n^3)$ time. We also present an implementation for the Abadi-Cardelli object calculus. Perhaps surprisingly, specifying read-only fields explicitly does not make type inference easier.

1 Introduction

1.1 Background

In the quest for more expressive type systems, subtyping has become a common means for flexible matching of types. For example, for an assignment x=e, we need not require that x and e have the same type; it is sufficient that the type of e be a subtype of the type of x. Subtyping comes in many flavors. For object types of the form $[\ell : B, \ldots]$, there are several design choices. Abadi and Cardelli [1] explain that if the field ℓ can be both read and updated, then ℓ must be *invariant*, that is, if $[\ell : A, \ldots]$ is a subtype of $[\ell : B, \ldots]$, then $A = B$. Following Abadi and Cardelli, we will use the notation $[\ell^0 : B, \ldots]$ where the superscript 0 denotes that the object type has an invariant field ℓ.

A covariant read-only field (CROF) is a field which enjoys covariant subtyping and which cannot be updated. Again following Abadi and Cardelli, we will use the notation $[m^+ : B, \ldots]$ where the superscript + denotes that the object type has a covariant field m. Covariance means that if $[m^+ : A, \ldots]$ is a subtype of $[m^+ : B, \ldots]$, then A is a subtype of B, a weaker condition than $A = B$.

CROFs make a type system more expressive. For example, below is a program written in a variant of the Abadi-Cardelli object calculus [1]. Each method $\varsigma(x)b$ binds a name x which denotes the smallest enclosing object, much like "this" in Java. We apologize in advance that the methods do not exhibit any useful behavior; the method were chosen to make them difficult to type check.

*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.
‡AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932, trevor@research.att.com.

```
let
  Point       = [move = @(x)x]
  ColorPoint  = [move = @(y)y  setcolor = @(z)z]
  Circle      = [center = @(d)Point]
  ColorCircle = update Circle.center <=
     @(e) call (call ColorPoint.move).setcolor
in
  call (call ColorCircle.center).move
```

Palsberg and Jim [13] noted that this program is *not* typable in Abadi and Cardelli's type system $\mathbf{Ob}_{1<:\mu}$, which has recursive types, width subtyping for object types, and only invariant fields. The key reason for the untypability is that the body of the `ColorCircle`'s `center` method forces `ColorPoint` to have a type which is *not* a subtype of the type of `Point`, intuitively as follows.

$$\begin{array}{rcl}
\texttt{Point} & : & \mu(X)[\texttt{move}^0 : X] \\
\texttt{ColorPoint} & : & \mu(X)[\texttt{move}^0, \texttt{setcolor}^0 : X] \\
\mu(X)[\texttt{move}^0, \texttt{setcolor}^0 : X] & \not\leq & \mu(X)[\texttt{move}^0 : X] \\
\text{Moreover} & : & \texttt{call (call ColorCircle.center).move} \text{ is } \textit{not} \text{ typable.}
\end{array}$$

To increase expressiveness, we can add CROFs to $\mathbf{Ob}_{1<:\mu}$ such that each field can be either invariant or covariant:

$$\mathbf{Ob}_{1<:\mu+} = \mathbf{Ob}_{1<:\mu} \cup \text{CROFs}.$$

This is sufficient to make the above program type check with the following types:

$$\begin{array}{lcl}
\texttt{Point} & : & [\texttt{move}^0\texttt{: [ ]]} \\
\texttt{ColorPoint} & : & [\texttt{move}^0\texttt{: [setcolor}^+\texttt{: [move}^+\texttt{: [ ]]]; setcolor}^0\texttt{: [move}^+\texttt{: [ ]]]} \\
\texttt{Circle} & : & [\texttt{center}^0\texttt{: [move}^+\texttt{: [ ]]]} \\
\texttt{ColorCircle} & : & [\texttt{center}^0\texttt{: [move}^+\texttt{: [ ]]]}
\end{array}$$

These types were produced by our implementation of the algorithm presented in this paper. Notice that the program (the input to our algorithm) does not mention whether a field is read-only, or whether it can both be read and updated. Our algorithm automatically *discovers* which fields should be CROFs.

The above program is also typable with the so-called simple self-types of Palsberg and Jim [13]. Bugliesi and Pericas-Geertsen [3] observed that any program that can be typed with simple self-types can also be typed in $\mathbf{Ob}_{1<:\mu+}$. Perhaps interestingly, their encoding of simple self-types uses recursive types to type the above program, while our more direct algorithm produces nonrecursive types in this case. More importantly, type inference with simple self-types is NP-complete [13], while our type inference algorithm for the more expressive type system $\mathbf{Ob}_{1<:\mu+}$ runs in polynomial time.

A programmer can specify that some of the fields are read-only. For example, in the above program, a programmer may specify that the `setcolor` field is read-only by inserting a `+` annotation:

```
let
  Point       = [move = @(x)x]
  ColorPoint  = [move = @(y)y  setcolor^+ = @(z)z]
  Circle      = [center = @(d)Point]
  ColorCircle = update Circle.center <=
     @(e) call (call ColorPoint.move).setcolor
in
  call (call ColorCircle.center).move
```

For this program, our implementation produces the following types:

`Point`	:	[move^0: [ ]]
`ColorPoint`	:	[move^0: [setcolor$^+$: [move$^+$: [ ]]]; setcolor$^+$: [move$^+$: [ ]]]
`Circle`	:	[center^0: [move$^+$: [ ]]]
`ColorCircle`	:	[center^0: [move$^+$: [ ]]]

Notice that the second occurrence of `setcolor` in the type for `ColorPoint` now has the annotation `+` instead of `0`.

If, in addition, the program were changed to make the `center` method of `Circle` read-only, then our implementation would correctly decide that the resulting program is not typable (because `Center` is updated in `ColorCircle`).

Perhaps surprisingly, adding annotations, such as the one for `setcolor`, seems not to make the type inference problem easier. In this paper we show that even if all fields are explicitly specified as either updateable or read-only, the type inference problem is P-complete. If the programmer leaves a field unspecified, then our algorithm will *discover* whether it is advantageous to make it read-only.

CROFs play an essential role in Glew's translation of objects and classes to a typed intermediate language [5]. Like most implementations of object-oriented languages, Glew's translation uses method tables. One of Glew's insights is that the method table can conveniently be placed in a CROF. For example, let a and b be two source-language objects such that the type of b is a subtype of the type of a. The type system for the source language supports that b may have *more* methods than a (width subtyping). This means that the method table in the translation of b will be *longer* than the method table in the translation of a:

$$\begin{aligned} \text{translation } (a) &= \ldots [\text{mt} = m_a, \ldots] \ldots \\ \text{translation } (b) &= \ldots [\text{mt} = m_b, \ldots] \ldots \end{aligned}$$

where mt is the field name for the method table. Glew's translation of b has a subtype of the type of his translation of a; he makes mt a CROF, and he gives the following types to the translations of a and b:

$$\begin{aligned} \text{type-of (translation } (a)) &= \ldots [\text{mt}^+ : \text{type-of } (m_a), \ldots] \ldots \\ \text{type-of (translation } (b)) &= \ldots [\text{mt}^+ : \text{type-of } (m_b), \ldots] \ldots \end{aligned}$$

Glew's translation produces typed intermediate code, including the annotations 0 and +.

Is type inference possible for an implicitly-typed version of Glew's intermediate language? In this paper we present a type-inference algorithm for a fragment of Glew's type system. Our long-term goal is to extend the algorithm to cover a larger fragment. Such an algorithm would make it possible to omit bulky type annotations, and to automatically discover the CROFs.

A $O(n^3)$ type inference algorithm for Abadi and Cardelli's type system $\mathbf{Ob}_{1<:\mu}$ (only invariant fields) was given in [12]. There is a similar $O(n^3)$ type inference algorithm for the calculus with only CROFs (only covariant fields). Surprisingly, there seems to be no easy way to "merge" the two algorithms to obtain an algorithm for the combined type system, $\mathbf{Ob}_{1<:\mu+}$. Both algorithms work 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 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 about both algorithms states:

Theorem A set of constraints is solvable if and only if its closure 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., $(\text{int}, [\ell : \text{boolean}])$). Both algorithms construct a solution from a closed, consistent constraint set. This framework has been used for solving subtype constraints for a variety of types [12, 7, 16, 14, 15]. 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.

Finding an algorithm thus rests on finding a correct definition of "closure." For the type system with only covariant fields, there are three closure rules, all operating on a constraint set R:

- if $(A, B) \in R$, then $(A, A), (B, B) \in R$ (reflexivity);
- if $(A, B), (B, C) \in R$, then $(A, C) \in R$ (transitivity);
- if $([\ell^+ : B, \ldots], [\ell^+ : B', \ldots]) \in R$, then $(B, B') \in R$ (propagation of subtyping to fields).

Computing the closure takes $O(n^3)$ time. For $\mathbf{Ob}_{1<:\mu}$ (only invariant fields), there are also three closure rules:

- if $(A, B) \in R$, then $(A, A), (B, B) \in R$ (reflexivity);
- if $(A, B), (B, C) \in R$, then $(A, C) \in R$ (transitivity);
- if $(A, [\ell^0 : B, \ldots]), (A, [\ell^0 : B', \ldots]) \in R$, then $(B, B') \in R$ (propagation of subtyping to fields).

Notice that the last rule also can be used to give (B', B), so it actually forces B and B' to be unified. Now, can we solve constraints over the types in $\mathbf{Ob}_{1<:\mu+}$ by taking the union of the two sets of closure rules? As it happens, a notion of closure based on the union of the rules does *not* support the result mentioned above. For example, consider the constraint set that consists of the following two constraints:

$$(V, [\ell^+ : \text{int}]) \qquad (V, [\ell^+ : \text{boolean}])$$

where V is a type variable. Apart from reflexivity, this constraint set is closed under the four closure rules above. Moreover, the constraint set is consistent: there are no obviously unsatisfiable constraints. So, if there was a theorem of the form mentioned above, this constraint set should be solvable. However, it is not solvable. To see that, consider the following informal argument. In any solution, V must be assigned a type of the form $[\ell^+ : A, \ldots]$, for some A. (Actually, it might also be possible to annotate ℓ with 0, but that will not help). Now, because ℓ is a CROF, we must be able to satisfy the constraints:

$$(A, \mathrm{int}) \qquad (A, \mathrm{boolean})$$

This is not possible: there is no subtype of both int and boolean in this system. Conclusion: either there are too few closure rules, or else there is something wrong with the notion of consistency.

The example suggests that in a setting with both covariant and invariant fields, a new technique is called for.

1.2 Our Results

We present the design and implementation of a type inference algorithm for $\mathbf{Ob}_{1<:\mu+}$. The algorithm automatically discovers CROFs. It is based on a theorem of the form discussed above, with a new notion of closure and a traditional notion of consistency. Type inference is equivalent to solving type constraints, which in turn is P-complete and computable in $O(n^3)$ time. The novel aspect of our definition of closure is that it keeps track of both subtype relations and which pairs of types must have a lower bound. For the example constraint set above, our closure rules will note that since $[\ell^+ : \mathrm{int}]$ and $[\ell^+ : \mathrm{boolean}]$ must have a lower bound, it must also be the case that int and boolean have a lower bound, which is obviously false. Our nine closure rules describe the interaction between a set of subtype constraints and a set of lower-bound constraints. In our proof of the main theorem (of the form mentioned above), we use a new technique 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.

Our prototype implementation, already showcased above, works with a version of the Abadi-Cardelli object calculus. The implementation is freely available from:

http://www.cs.purdue.edu/homes/tzhao/type-inference/inference.htm

Future work includes the addition of atomic subtyping [10, 18, 6, 4, 2].

One of the first uses of annotations such as +, often called *variance annotations*, can be found in Pierce and Sangiorgi's paper [17] on typing and subtyping for mobile processes. They used annotations of types in a type system for the π-calculus to enforce that some channels are for input only or for output only.

The variance annotation "$-$" is sometimes used to denote that a field is contravariant and write-only. We know of no easy way of extending the results of this paper to cover "$-$".

1.3 Examples

We now present two examples that give a taste of the definitions and techniques that are used later in the paper. In the first example we return to the program with points and colorpoints. We use the program to illustrate the reduction of the type inference problem to a constraint problem. In the abstract syntax of the Abadi-Cardelli object calculus, we can write the program as follows:

$$\begin{array}{rcl}
\mathrm{Point} & = & [\mathrm{move} = \varsigma(x)x] \\
\mathrm{ColorPoint} & = & [\mathrm{move} = \varsigma(y)y,\ \mathrm{setcolor} = \varsigma(z)z] \\
\mathrm{Circle} & = & [\mathrm{center} = \varsigma(d)\mathrm{Point}] \\
\mathrm{ColorCircle} & = & \mathrm{Circle.center} \Leftarrow \varsigma(e)\mathrm{ColorPoint.move.setcolor} \\
\mathrm{Main} & = & \mathrm{ColorCircle.center.move}
\end{array}$$

We can use the rules in Section 4.1 to generate the following set of constraints. 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)
Point	$([\text{move}^0 : V_x], V_{Point})$
	$U_x \equiv [\text{move}^0 : V_x]$
y	(U_y, V_y)
z	(U_z, V_z)
ColorPoint	$([\text{move}^0 : V_y,\ \text{setcolor}^0 : V_z], V_{ColorPoint})$
	$U_y \equiv [\text{move}^0 : V_y,\ \text{setcolor}^0 : V_z]$
	$U_z \equiv [\text{move}^0 : V_y,\ \text{setcolor}^0 : V_z]$
Circle	$([\text{center}^0 : V_{Point}], V_{Circle})$
	$U_d \equiv [\text{center}^0 : V_{Point}]$
ColorCircle	$(V_{Circle}, V_{ColorCircle})$
	$V_{Circle} \equiv U_e$
	$(V_{Circle}, [\text{center}^0 : V_{ColorPoint.move.setcolor}])$
ColorPoint.move	$(V_{ColorPoint}, [\text{move}^+ : U_{ColorPoint.move}])$
	$(U_{ColorPoint.move}, V_{ColorPoint.move})$
ColorPoint.move.setcolor	$(V_{ColorPoint.move}, [\text{setcolor}^+ : U_{ColorPoint.move.setcolor}])$
	$(U_{ColorPoint.move.setcolor}, V_{ColorPoint.move.setcolor})$
ColorCircle.center	$(V_{ColorCircle}, [\text{center}^+ : U_{ColorCircle.center}])$
	$(U_{ColorCircle.center}, V_{ColorCircle.center})$
ColorCircle.center.move	$(V_{ColorCircle.center}, [\text{move}^+ : U_{ColorCircle.center.move}])$
	$(U_{ColorCircle.center.move}, V_{ColorCircle.center.move})$

The above constraint set is solvable and a solution can be found by running our constraint solving algorithm. The solution that will be generated was displayed earlier in this section; it corresponds to the following type derivation. Define:

$$
\begin{aligned}
P &= [\text{move}^+ : [\,]] \\
Q &= [\text{move}^0 : [\text{setcolor}^+ : P],\ \text{setcolor}^0 : P] \\
E &= \emptyset[d : [\text{center}^0 : P]] \\
F &= \emptyset[e : [\text{center}^0 : P]].
\end{aligned}
$$

We can derive $\emptyset \vdash \text{ColorCircle.center.move} : [\,]$ as follows.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E[x : [\text{move}^0 : [\,]]] \vdash x : [\text{move}^0 : [\,]]}{E \vdash \text{Point} : [\text{move}^0 : [\,]]}}{E \vdash \text{Point} : P}}{\emptyset \vdash \text{Circle} : [\text{center}^0 : P]} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{F[y : Q] \vdash y : Q}{F[y : Q] \vdash y : [\text{setcolor}^+ : P]} \qquad \dfrac{F[z : Q] \vdash z : Q}{F[z : Q] \vdash z : P}}{F \vdash \text{ColorPoint} : Q}}{F \vdash \text{ColorPoint} : [\text{move}^+ : [\text{setcolor}^+ : P]]}}{F \vdash \text{ColorPoint.move} : [\text{setcolor}^+ : P]}}{F \vdash \text{ColorPoint.move.setcolor} : P}}{\emptyset \vdash \text{ColorCircle} : [\text{center}^0 : P]}}{\emptyset \vdash \text{ColorCircle.center} : P}}{\emptyset \vdash \text{ColorCircle.center.move} : [\,]}
$$

Notice the four uses of subsumption:

$$\begin{aligned} [\text{move}^0 : [\,]] &\leq P \\ Q &\leq [\text{setcolor}^+ : P] \\ Q &\leq P \\ Q &\leq [\text{move}^+ : [\text{setcolor}^+ : P]]. \end{aligned}$$

Our second example illustrates our algorithm for solving constraints, particularly the role of the closure operation. Let R consist of the following three constraints:

$$\begin{aligned} &(U, [\ell^+ : [\ell^+ : [m^+ : [\,]]]]) \\ &(U, V) \\ &(V, [\ell^+ : [\ell^0 : W]]), \end{aligned}$$

where U, V, W are type variables and ℓ, m are labels of fields. The satisfaction-closure of R and the accompanying lower-bound relation are shown below. A pair (A, B) is in the lower-bound relation when it has been deduced that A and B must have a lower bound in the subtype ordering.

Satisfaction-closure of R (excerpt)	Lower-bound relation (excerpt)
$(U \;,\; [\ell^+ : [\ell^+ : [m^+ : [\,]]]])$	$([\ell^+ : [\ell^0 : W]] \;,\; [\ell^+ : [\ell^+ : [m^+ : [\,]]]])$
$(U \;,\; V)$	$([\ell^0 : W] \;,\; [\ell^+ : [m^+ : [\,]]])$
$(V \;,\; [\ell^+ : [\ell^0 : W]])$	
$(U \;,\; [\ell^+ : [\ell^0 : W]])$	
$(W \;,\; [m^+ : [\,]])$.	

Let us now explain how the rules for satisfaction-closure (Definition 5.1) generate the constraints in the table above. Since (U, V) and $(V, [\ell^+ : [\ell^0 : W]])$ are both in R and therefore in the sat-closure of R, we have from transitivity of R (sat-closure rule (iii)) that $(U, [\ell^+ : [\ell^0 : W]])$ is in the sat-closure of R. Furthermore, since both $(U, [\ell^+ : [\ell^0 : W]])$ and $(U, [\ell^+ : [\ell^+ : [m^+ : [\,]]]])$ are in the sat-closure of R, we have from Lemma 5.3, Property **(C)**, that $([\ell^+ : [\ell^0 : W]], [\ell^+ : [\ell^+ : [m^+ : [\,]]]])$ is in the lower bound relation, and hence, from sat-closure rule (vii), we have that also $([\ell^0 : W], [\ell^+ : [m^+ : [\,]]])$ is in the lower-bound relation. Finally, since $([\ell^0 : W], [\ell^+ : [m^+ : [\,]]])$ is in the lower-bound relation, we have from sat-closure rule (viii) that $(W, [m^+ : [\,]])$ is in the sat-closure of R. Given the sat-closure of R, call it R', our algorithm checks for satisfaction-inconsistency, that is, subtyping constraints that obviously are unsatisfiable. In this case, the satisfaction-closure is satisfaction-consistent, and our algorithm then constructs the following solution $S_{R'}$:

$$\begin{aligned} S_{R'}(U) &= [\ell^+ : [\ell^0 : [m^+ : [\,]]]] \\ S_{R'}(V) &= [\ell^+ : [\ell^0 : [m^+ : [\,]]]] \\ S_{R'}(W) &= [m^+ : [\,]]. \end{aligned}$$

One might try to devise a constraint solving algorithm that would be an alternative to ours. The constraint set R is a good benchmark: it seems nontrivial to derive $(W, [m^+ : [\,]])$ without the help of a lower-bound relation.

Paper overview In Section 2, we define types and subtyping, and we give a decision procedure for subtyping. In Section 3 we present an extension of the Abadi-Cardelli object calculus, and in Section 4 we show that the type inference problem for that calculus is equivalent to a constraint problem. In Section 5 we give an $O(n^3)$-time algorithm for solving constraints, and in Section 6 we show that the constraint problem is P-hard.

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 $\mathsf{Variances} = \{0, +\}$ of variance annotations. Variance annotations are ordered by the smallest partial order $\sqsubseteq$ such that $0 \sqsubseteq +$.

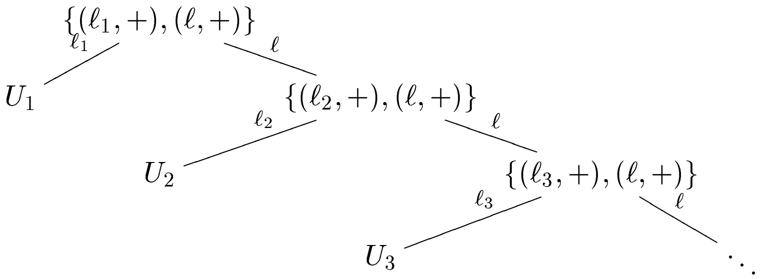
The alphabet Σ of our trees is defined

$$\Sigma = \mathcal{TV} \cup \{\sigma \subseteq \mathsf{Labels} \times \mathsf{Variances} \mid (\ell, v), (\ell, v') \in \sigma \Rightarrow v = v'\}.$$

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 $(\ell, v) \in A(\alpha)$ if and only if $A(\alpha\ell)$ 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. Of course, we will be particularly interested in two subsets of $\mathcal{T}(\Sigma)$, the finite trees $\mathcal{T}_{\mathsf{fin}}(\Sigma)$ and the finitely branching and regular trees $\mathcal{T}_{\mathsf{reg}}(\Sigma)$. Some definitions, results, and proofs are given in terms of $\mathcal{T}(\Sigma)$, in such a way that they immediately apply to $\mathcal{T}_{\mathsf{fin}}(\Sigma)$ and $\mathcal{T}_{\mathsf{reg}}(\Sigma)$. In particular, we will state conditions on whether one tree is a subtype of another that result in an *algorithm* in case both trees are in $\mathcal{T}_{\mathsf{fin}}(\Sigma)$ or $\mathcal{T}_{\mathsf{reg}}(\Sigma)$.

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

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

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 $\alpha \neq \epsilon$.

2.2 Defining subtyping via simulations

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

- For all U, $A = U$ if and only if $A' = U$.
- For all ℓ, v', B', if $A' = [\ell^{v'} : B', \ldots]$, then there exist v, B such that $A = [\ell^{v} : B, \ldots]$, $v \sqsubseteq v'$, and
  - $(B, B') \in R$, and
  - $v' = 0$ implies $(B', B) \in R$.

□

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 $\leq$:

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

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, v', B', if $A' = [\ell^{v'} : B', \ldots]$, then there exist v, B such that $A = [\ell^{v} : B, \ldots]$, $v \sqsubseteq v'$, and*
  - *$B \leq B'$, and*
  - *$v' = 0$ implies $B' \leq B$.*

All of these results are standard in concurrency theory, and have easy proofs, c.f. [9]. 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.*

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 An algorithm for subtyping

The co-induction principle results in an easy algorithm for subtyping on $\mathcal{T}_{\text{fin}}(\Sigma)$ and $\mathcal{T}_{\text{reg}}(\Sigma)$. 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 it satisfies the following two properties.

- If $([\ell^{v} : B, \ldots], [\ell^{v'} : B', \ldots]) \in R$ and $v \sqsubseteq v'$, then $(B, B') \in R$.
- If $([\ell^{0} : B, \ldots], [\ell^{0} : B', \ldots]) \in R$, then $(B', B) \in R$.

□

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-inconsistent* if any of the following cases hold.

- $(U, A) \in R$ or $(A, U) \in R$ for some distinct U and A.
- $([\ell^+ : B, \ldots], [\ell^0 : B', \ldots]) \in R$ for some ℓ, B, B'.
- $([\ell_i^{v_i} : B_i \ ^{i \in I}], [\ell^v : B, \ldots]) \in R$ for some ℓ, v, B, and ℓ_i, v_i, B_i for $i \in I$; and furthermore $\ell \neq \ell_i$ for all $i \in I$.

We say R is *subtype-consistent* iff R is not subtype-inconsistent. □

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.*

i) $A \leq B$ *for every* $(A, B) \in R$.

ii) *The subtype-closure of R is a simulation.*

iii) *The subtype-closure of R is subtype-consistent.*

This immediately suggests an algorithm for testing whether $A \leq B$ for $A, B \in \mathcal{T}_{\mathrm{reg}}(\Sigma)$: construct the subtype-closure of $\{(A, B)\}$ and test whether it is subtype-consistent. If n is the number of distinct subtrees of A and B, then the subtype-closure of $\{(A, B)\}$ is of size at most n^2, and can be constructed in n^2 time. Consistency checking is linear, so the algorithm runs in time $O(n^2)$.

Theorem 2.7 *Subtyping on $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ is decidable in $O(n^2)$ time.*

In the remainder of the paper, we only consider types in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

3 The Abadi-Cardelli Object Calculus

We now present an extension of the Abadi-Cardelli object calculus [1] and a static type system.

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

$$
\begin{array}{rcll}
a, b, c & ::= & x & \text{variable} \\
& | & [\ell_i^{v_i} = \varsigma(x_i) b_i \ ^{i \in 1..n}] & \text{object } (\ell_i \text{ distinct}) \\
& | & a.\ell & \text{field selection / method invocation} \\
& | & (a.\ell \Leftarrow \varsigma(x) b) & \text{field update / method update}
\end{array}
$$

An object $[\ell_i^{v_i} = \varsigma(x_i) b_i \ ^{i \in 1..n}]$ has method names ℓ_i and methods $\varsigma(x_i) b_i$. The order of the methods does not matter. Each method $\varsigma(x) b$ 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. Each method name ℓ_i is annotated with a variance annotation

$v_i \in \{0, +\}$ which in the case of 0 indicates that the method is both readable/invocable and writable/updateable, while it in the case of + indicates that the method is only readable/invocable. As syntactic sugar, we will allow variance annotations to be omitted, and in such cases the default is 0. With this default, our calculus is an extension of the Abadi-Cardelli calculus [1]: a term in the Abadi-Cardelli calculus is also a term in our calculus, namely one where all variance annotations implicitly are 0. A *value* is of the form $[\ell_i^{v_i} = \varsigma(x_i)b_i{}^{i\in 1..n}]$. A *program* is a closed expression.

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

- If $a = [\ell_i^{v_i} = \varsigma(x_i)b_i{}^{i\in 1..n}]$, then, for $j \in 1..n$,
  - $a.\ell_j \rightsquigarrow b_j[x_j := a]$, and
  - if $v_j = 0$, then $(a.\ell_j \Leftarrow \varsigma(y)b) \rightsquigarrow a[\ell_j \leftarrow \varsigma(y)b]$.
- If $b \rightsquigarrow b'$, then $a[b] \rightsquigarrow a[b']$.

Here, $b_j[x_j := a]$ denotes the expression 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 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 $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

$$E \vdash x : A \quad \text{(provided } E(x) = A\text{)} \tag{1}$$

$$\frac{E[x_i : A] \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [\ell_i = \varsigma(x_i)b_i{}^{i\in 1..n}] : A} \quad \text{(where } A = [\ell_i^{v_i} : B_i{}^{i\in 1..n}]\text{)} \tag{2}$$

$$\frac{E \vdash a : A}{E \vdash a.\ell : B} \quad \text{(where } A \leq [\ell^+ : B]\text{)} \tag{3}$$

$$\frac{E \vdash a : A \quad E[x : A] \vdash b : B}{E \vdash a.\ell \Leftarrow \varsigma(x)b : A} \quad \text{(where } A \leq [\ell^0 : B]\text{)} \tag{4}$$

$$\frac{E \vdash a : A}{E \vdash a : B} \quad \text{(where } A \leq B\text{)} \tag{5}$$

The first four 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, 19].

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 Type Inference is equivalent to Constraint Solving

A *substitution* S is a finite partial function from type variables to types in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, 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.

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

We now prove that the type inference problem is logspace-equivalent to solving constraints.

4.1 From Type Inference to Constraint Solving

We first prove that the type inference problem is logspace-reducible to solving constraints.

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 4.1 (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 $\leq$.

Lemma 4.2 (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, $E \vdash a : A$, $A \leq [\ell^+ : B]$, and $B \leq C$;*
- *$c = [\ell_i^{v_i} = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, and for some A, and some B_i, for $i \in 1..n$, $E[x_i : A] \vdash b_i : B_i$, and $A = [\ell_i^{v_i} : B_i \ ^{i \in 1..n}] \leq C$; or*
- *$c = (a.\ell \Leftarrow \varsigma(x)b)$, and for some A and B, $E \vdash a : A$, $E[x : A] \vdash b : B$, $A \leq [\ell^0 : B]$, and $A \leq C$.*

Definition 4.3 Let c be an expression 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 contains a type variable U_x for every term variable x appearing in c.
- Y_c is a set of fresh type variables. It contains 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

$$(U_x, V_x). \tag{6}$$

- For each occurrence in c of a subterm of the form $a.\ell$, the two constraints

$$(V_a, [\ell^+ : U_{a.\ell}]) \tag{7}$$

$$(U_{a.\ell}, V_{a.\ell}). \tag{8}$$

- For each occurrence in c of a subterm of the form $[\ell_i^{v_i} = \varsigma(x_i)b_i \ {}^{i\in 1..n}]$, the constraint

$$([\ell_i^{v_i} : V_{b_i} \ {}^{i\in 1..n}], V_{[\ell_i^{v_i} = \varsigma(x_i)b_i \ {}^{i\in 1..n}]}) \tag{9}$$

  and for each $j \in 1..n$, the constraints

$$U_{x_j} \equiv [\ell_i^{v_i} : V_{b_i} \ {}^{i\in 1..n}]. \tag{10}$$

- For each occurrence in c of a subterm of the form $(a.\ell \Leftarrow \varsigma(x)b)$, the constraints

$$(V_a, V_{(a.\ell \Leftarrow \varsigma(x)b)}) \tag{11}$$

$$V_a \equiv U_x \tag{12}$$

$$(V_a, [\ell^0 : V_b]). \tag{13}$$

$\square$

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

Theorem 4.4 *$E \vdash c : C$ 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 4.5 *Let c_0 be an expression. For every subterm c of c_0,*

i) *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*

ii) *if S is a solution of $\mathcal{C}(c_0)$, then $S(E_c) \vdash c : S(V_c)$.*

4.2 From Constraint Solving to Type Inference

The following result is proved by a method similar to the one used by Palsberg [12] and Palsberg and Jim [13].

Lemma 4.6 *Solvability of constraints is logspace-reducible to the type inference problem.*

5 Solving Constraints

In this section we present an algorithm for deciding whether a relation R is solvable. We first define the notions of satisfaction-closure (Section 5.1) and satisfaction-consistency (Section 5.2), and then we prove that a relation R is solvable if and only its satisfaction-closure is satisfaction-consistent (Theorem 5.11).

5.1 Satisfaction-closure

Definition 5.1 If R is a relation on types, we say R is *satisfaction-closed* (abbreviated sat-closed) if there exists relation L on types such that

i) if $(A, B) \in R$, then $(A, A), (B, B) \in R$.

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

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

iv) if $(A, B) \in L$, then $(B, A) \in L$;

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

vi) if $([\ell^+ : B, \ldots], [\ell^+ : B', \ldots]) \in R$, then $(B, B') \in R$;

vii) if $([\ell^+ : B, \ldots], [\ell^+ : B', \ldots]) \in L$, then $(B, B') \in L$;

viii) if $([\ell^0 : B, \ldots], [\ell^+ : B', \ldots]) \in L$, then $(B, B') \in R$;

ix) if $([\ell^0 : B, \ldots], [\ell^0 : B', \ldots]) \in L$, then $(B, B') \in R$.

□

Notice that the intersection of a family of sat-closed relations is itself sat-closed. From that we have that for a given relation R, there is a smallest sat-closed relation that includes R; we call that sat-closed relation the *sat-closure* of R.

Notice also that for a given sat-closed relation R, there is a smallest relation L such that the sat-closure rules for R and L are satisfied; we call L *the lower-bound relation for* R.

For a relation R, notice that the sat-closure of R and the lower bound relation for the sat-closure of R are the pairwise-$\subseteq$-smallest pair (R', L) that contains $(R, \emptyset)$. We can compute R' and L by a straightforward fixed-point computation that uses the sat-closure rules to monotonically add elements to the two relations. We can apply McAllester's theorem [8] to get that the fixed-point computation takes $O(n^3)$ time where n is the size of R.

Lemma 5.2 *A relation and its sat-closure have the same set of solutions.*

A sat-closed relation has the five properties that are expressed in the following lemma.

Lemma 5.3 *Suppose R is sat-closed, and let L be the lower-bound relation for R.*

(A) *If $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in L$, then $(B, B') \in L$.*

(B) *If $(A, A') \in L$ and $(A, A_1), (A', A_2) \in R$, then then $(A_1, A_2) \in L$.*

(C) *If $(A, A_1), (A, A_2) \in R$, then $(A_1, A_2) \in L$.*

(D) *If* $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in R$ *and* $v \sqsubseteq v'$*, then* $(B, B') \in R$*.*

(E) *If* $([\ell^0 : B, \ldots], [\ell^0 : B', \ldots]) \in R$*, then* $(B', B) \in R$*.*

From Lemma 5.3, Properties **(D)** and **(E)**, we have that if a relation is sat-closed, then it is also subtype-closed.

5.2 Satisfaction-consistency

Definition 5.4 We say R is *satisfaction-inconsistent* (abbreviated sat-inconsistent) if any of the following two cases hold.

- $([\ell^+ : B, \ldots], [\ell^0 : B', \ldots]) \in R$ for some ℓ, B, B'.
- $([\ell_i^{v_i} : B_i \;^{i \in I}], [\ell^v : B, \ldots]) \in R$ for some ℓ, v, B, and ℓ_i, v_i, B_i for $i \in I$; and furthermore $\ell \neq \ell_i$ for all $i \in I$.

We say R is *sat-consistent* iff R is not sat-inconsistent. $\square$

Notice that if a relation is subtype-consistent, then it is also sat-consistent.

Lemma 5.5 *If R is solvable, then R is sat-consistent.*

5.3 Main Result

We first list the terminology used in the later definitions.

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

To define the solution S_R, we will need the following notation. We use $\mathcal{T}$ to range over sets of types. Then we make the following definitions.

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

In the last definition, $\sqcap$ is the greatest lower bound of a nonempty set of variances; $\sqcap\emptyset$ is undefined.

The types of the above definitions are

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

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

$$\mathrm{LV}(\mathcal{T}) = \{(\ell, v) \mid v = \mathrm{Var}(\mathcal{T}, \ell)\}.$$

For a 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.

$$\begin{aligned} \delta_R^*(g)(\epsilon) &= g, \\ \delta_R^*(g)(\ell\alpha) &= \delta_R^*(\delta_R(g)(\ell))(\alpha). \end{aligned}$$

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

$$\begin{aligned} \text{Type}_R(g)(\alpha) &= \text{LV}(\delta_R^*(g)(\alpha)), \\ \text{TYPE}_R(\mathcal{R}) &= \{(\text{Type}_R(g), \text{Type}_R(h)) \mid (g,h) \in \mathcal{R}\}. \end{aligned}$$

We have that

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

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

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

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

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

- $(A, B) \in R$.
- For some ℓ, v, v', such that $v \sqsubseteq v'$, we have $([\ell^v : A, \ldots], [\ell^{v'} : B, \ldots]) \in R$.
- For some ℓ, we have $([\ell^0 : B, \ldots], [\ell^0 : A, \ldots]) \in R$.

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

Define the function $\mathcal{B}_R : \mathsf{RelStates} \to \mathsf{RelStates}$ by $(g, h) \in \mathcal{B}_R(\mathcal{R})$ iff one of the following conditions holds:

- $(g, h) \in \mathcal{R}$.
- For some ℓ and $(g', h') \in \mathcal{R}$, such that $\text{Var}(g', \ell) \sqsubseteq \text{Var}(h', \ell)$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$.
- For some ℓ and $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$, $\text{Var}(g', \ell) = 0$, and $\text{Var}(h', \ell) = 0$.

The next two lemmas (Lemma 5.6 and Lemma 5.7) are key ingredients in the proof of Lemma 5.8. Lemma 5.6 states fundamental relationship between TYPE_R, $\mathcal{A}$, and $\mathcal{B}_R$.

Lemma 5.6 *The following diagram commutes:*

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

Lemma 5.7 *Suppose R is sat-closed. For all n, if $(g, h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, then $g \supseteq h$.*

Lemma 5.8 *If R is sat-closed, then the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is subtype-consistent.*

Proof.

$$\begin{aligned} &\text{The subtype-closure of } \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R) \\ = &\bigcup_{0 \le n < \infty} \mathcal{A}^n \circ \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R) \qquad \text{(Definition of subtype-closure)} \\ = &\bigcup_{0 \le n < \infty} \mathrm{TYPE}_R \circ \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R) \qquad \text{(Lemma 5.6)} \\ = &\bigcup_{0 \le n < \infty} \ \bigcup_{(g,h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)} \{(\mathrm{Type}_R(g), \mathrm{Type}_R(h))\} \qquad \text{(Definition of } \mathrm{TYPE}_R\text{).} \end{aligned}$$

From Lemma 5.7 we have that if $(g, h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, then $g \supseteq h$. If $g \supseteq h$, then 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 relations. Since the union of a family of subtype-consistent 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 5.10. The two lemmas 5.9 and 5.10 are the two places where it is needed that a relation is satisfaction-consistent.

Lemma 5.9 *If $A = [\ell^v : B, \ldots]$ appears in R and R is sat-closed and sat-consistent, then*

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

Lemma 5.10 *If R is sat-closed and sat-consistent, then*

i) *for any type A appearing in R, $S_R(A) = \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})$; and*

ii) *$S_R(R) = \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$.*

We are now ready to prove the main result of this section.

Theorem 5.11 *R is solvable if and only if its sat-closure is sat-consistent.*

Proof. If R is solvable, then we have from Lemma 5.2 that the sat-closure of R is solvable, so from Lemma 5.5 we have the sat-closure of R is sat-consistent.

Conversely, let R' be the sat-closure of R, and assume that R' is sat-consistent. From Lemma 5.8 and Lemma 5.10, we have that the subtype-closure of $S_{R'}(R')$ is subtype-consistent, so from Lemma 2.6, we have that R' has solution $S_{R'}$, and so from Lemma 5.2 we have that R has solution $S_{R'}$. □

In summary, to solve a constraint set R, we proceed as follows. First, we compute the sat-closure R' of R (this takes $O(n^3)$ time where n is the size of R). Next, the R' is checked for sat-consistency (this takes $O(n)$ time). If R' is sat-inconsistent, then R is not solvable, otherwise $S_{R'}$ is an example of a solution of R. Thus, we have shown the following result.

Corollary 5.12 *Satisfiability of a constraint set is decidable in $O(n^3)$ time.*

6 P-hardness

Theorem 6.1 *Solvability of constraints is P-hard.*

Proof. An SC-system (simple constraint set) is a finite set of constraints of the forms

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

Notice that any SC-system is a constraint set. Thus it is sufficient to prove that solvability of SC-systems over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ is P-hard. We will do that by reducing a closely related P-hard problem to this problem.

Let $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ be the subset of $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ where all variance annotations are 0. In [12] it is proved that solvability of SC-systems over $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ is P-hard.

Let R be an SC-system. It is sufficient to prove that R is solvable over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ if and only if it is solvable over $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$.

It is immediate that if R is solvable over $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ then it is solvable over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

Conversely, suppose that R is solvable over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, and let R' be the sat-closure of R. We have that $S_{R'}$ is a solution of R', and therefore, by Lemma 5.2, $S_{R'}$ is also a solution of R. By Lemma 6.2, $S_{R'}$ maps all variables in R to elements of $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$, □

Lemma 6.2 *Let R to be an SC-system and R' is the sat-closure of R. If R is solvable over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, then $S_{R'}(V) \in \mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ for all V in R.*

7 Conclusion

We can do type inference in polynomial time for objects with both covariant and invariant fields. Covariant read-only fields can be either explicitly specified or discovered by our algorithm. Perhaps surprisingly, specifying read-only fields explicitly seems not to make type inference easier.

Acknowledgment: Palsberg was supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265.

References

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

[2] Marcin Benke. Efficient type reconstruction in the presence of inheritance. Manuscript, 1994.

[3] Michele Bugliesi and Santiago Pericas-Geertsen. Type inference for variant object types. *Information and Computation.* to appear.

[4] Alexandre Frey. Satisfying systems of subtype inequalities in polynomial space. In *Proceedings of SAS'97, International Static Analysis Symposium.* Springer-Verlag (*LNCS*), 1997.

[5] Neal Glew. An efficient class and object encoding. In *Proceedings of OOPSLA'00, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 311–324, Minneapolis, Minnesota, October 2000.

[6] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 176–185, 1995.

[7] 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 FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[8] David McAllester. On the complexity analysis of static analyses. In *Proceedings of SAS'99, 6th International Static Analysis Symposium*, pages 312–329. Springer-Verlag (*LNCS* 1694), 1999.

[9] 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, N.Y., 1990.

[10] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[11] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE*, 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 LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

[13] Jens Palsberg and Trevor Jim. Type inference with simple selftypes is NP-complete. *Nordic Journal of Computing*, 4(3):259–286, Fall 1997.

[14] 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 POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.

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

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

[17] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *LICS'93, Eighth Annual Symposium on Logic in Computer Science*, pages 376–385, 1993.

[18] Jerzy Tiuryn. Subtype inequalities. In *LICS'92, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 308–315, 1992.

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