Automatic Discovery of Covariant Read-Only Fields

Jens Palsberg
Purdue University*

Tian Zhao
Purdue University†

Trevor Jim
AT&T Labs Research‡

October 22, 2003

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 relies crucially on covariant subtyping of read-only fields to ensure that subclasses are translated to subtypes.

In this paper, we present a type inference algorithm for an Abadi-Cardelli object calculus in which fields are marked either as updateable or as read-only. The type inference problem is P-complete, and our algorithm runs in $O(n^3)$ time. The same complexity results hold for the calculus in which the fields are not explicitly annotated as updateable or read-only; perhaps surprisingly, the annotations do not make type inference easier. We show that type inference is equivalent to the problem of solving type constraints, and this forms the core of our algorithm and implementation.

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, in an assignment x=e, we can allow x and e to have different types, provided that the type of e is a subtype of the type of x. This relieves the programmer from having to insert coercions.

Subtyping comes in many flavors. For object types of the form $[\ell : B, \ldots]$, where $\ell : B$ denotes a field ℓ of type B, 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* in the subtype ordering, 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 are a useful addition to type systems. For example, Glew has given a nice translation of classes and objects into an intermediate calculus in which the method tables of classes are placed

*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.

in CROFs; covariant subtyping on the method table fields is needed to ensure that subclasses are translated to subtypes. We will discuss Glew's translation in more detail later in the paper.

First, however, we illustrate some of the central technical difficulties of CROFs with a much simpler, more whimsical example. Our program is written in a variant of an Abadi-Cardelli object calculus [1]. Each method `@(`x`)`b binds a name x that denotes the smallest enclosing object, much like Java's "this." The program declares four objects, then makes two successive method calls:

```
let
  Man      = [id = @(x)x]
  Batman   = [id = @(y)y  secretid = @(z)z]
  Phone    = [dial = @(d)Man]
  Batphone = update Phone.dial <=
     @(e) call (call Batman.id).secretid
in
  call (call Batphone.dial).id
```

Here is the intuition behind the example. A `Man` has an identity (himself). `Batman` has an identity too, and he also has a secret identity. By dialing the `Phone` you can reach `Man`, and by dialing the `Batphone` you can reach `Batman`: the definition of `Batphone` is equivalent to `[dial = @(e)Batman]` (the update is a functional update).

Some fairly powerful type systems cannot handle this example: Palsberg and Jim [21] 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 reason is that in this calculus, `Batman` is not a subtype of `Man`:

$$\begin{array}{rcll} \texttt{Man} & : & \mu(X)[\texttt{id}^0 : X] & \text{(from the definition of }\texttt{Man}\text{)} \\ \texttt{Batman} & : & \mu(Y)[\texttt{id}^0, \texttt{secretid}^0 : Y] & \text{(from the definition of }\texttt{Batman}\text{)} \\ \mu(Y)[\texttt{id}^0, \texttt{secretid}^0 : Y] & \not\leq & \mu(X)[\texttt{id}^0 : X] & \text{(from the definition of subtyping)} \end{array}$$

The `id` field of `Batman` has `Batman`'s type, and the `id` field of `Man` has `Man`'s type, so by invariant subtyping, the two types are not related. The only common type of `Man` and `Batman` is [ ], so we have

$$\begin{array}{rcl} \texttt{Phone} & : & [\texttt{dial}^0 : [\,]] \\ \texttt{Batphone} & : & [\texttt{dial}^0 : [\,]] \end{array}$$

In conclusion, `call (call Batphone.dial).id` is *not* typable.

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:

$$\begin{array}{lcl} \texttt{Man} & : & [\texttt{id}^0\texttt{: [ ]]} \\ \texttt{Batman} & : & [\texttt{id}^0\texttt{: [secretid}^+\texttt{: [id}^+\texttt{: [ ]]]; secretid}^0\texttt{: [id}^+\texttt{: [ ]]]} \\ \texttt{Phone} & : & [\texttt{dial}^0\texttt{: [id}^+\texttt{: [ ]]]} \\ \texttt{Batphone} & : & [\texttt{dial}^0\texttt{: [id}^+\texttt{: [ ]]]} \end{array}$$

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. It turns out that this information does not make the problem of type inference any easier, so our algorithm automatically *discovers* which fields should be CROFs.

The types produced by our algorithm are intended to be minimal-shape types, in the sense of [12, 24], although proving that is left to future work. The idea of minimal shape is that the inferred types exhibit exactly the amount of structure of objects that the program exploits. For example, we have `Man` : [$\texttt{id}^0$: [ ]] because there is no usage of `Man.id`, so the return type of `Man.id` need not have any features at all. Note that minimal-shape types need not be principal types. If principal types are desired, then the best known approach is to output a representation of the type constraints used by the type inference algorithm, possibly in a simplified form [27]. In the worst case, such a set of constraints has a size that is linear in the size of the program.

One can achieve a degree of modularity by letting the programmer specify types at, say, module boundaries. Those types then become part of the input to the type inference algorithm, so, in effect, the algorithm determines whether there is a typing which is consistent with the specified types. Without any such declared types, the inferred types can be quite different for two programs that use their objects in slightly different ways. This may not be desirable for large-scale software development. Similarly, slightly different programs may have quite different inferred annotations of the fields of the object types. It remains an open problem to find a convenient mechanism to explain to a programmer why a certain field ended up being a CROF.

The above program is also typable with the so-called simple self-types of Palsberg and Jim [21]. Bugliesi and Pericas-Geertsen [5] 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 [21], while our type inference algorithm for the more expressive type system $\mathbf{Ob}_{1<:\mu+}$ runs in polynomial time.

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

```
let
  Man      = [id = @(x)x]
  Batman   = [id = @(y)y  secretid+ = @(z)z]
  Phone    = [dial = @(d)Man]
  Batphone = update Phone.dial <=
     @(e) call (call Batman.id).secretid
in
  call (call Batphone.dial).id
```

For this program, our implementation produces the following types:

$$\begin{array}{lcl}
\texttt{Man} & : & [\texttt{id}^0\texttt{: [ ]]} \\
\texttt{Batman} & : & [\texttt{id}^0\texttt{: [secretid}^+\texttt{: [id}^+\texttt{: [ ]]]; secretid}^+\texttt{: [id}^+\texttt{: [ ]]]} \\
\texttt{Phone} & : & [\texttt{dial}^0\texttt{: [id}^+\texttt{: [ ]]]} \\
\texttt{Batphone} & : & [\texttt{dial}^0\texttt{: [id}^+\texttt{: [ ]]]}
\end{array}$$

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

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

As we have mentioned, adding annotations, such as the one for `secretid`, 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 (or, equivalently, gives the annotation 0), then our algorithm will *discover* whether it is advantageous to make it read-only. The "discovery" process takes place at the type level: a field which was annotated as read-write may be implicitly turned into a read-only field at any time via subsumption.

Glew's type system Glew's translation of objects and classes to a typed intermediate language [7] is an important motivation for our work on CROFs. 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:

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

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:

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

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? Our paper shows that type inference is possible for a fragment of Glew's type system. Glew's type system also features function types, bounded universal polymorphism, and self types, as well as a special variance annotation for record types which indicates whether subtyping can be applied at all. Our long-term goal is to extend the algorithm to cover a larger fragment of Glew's system. Such an algorithm would make it possible to omit bulky type annotations, and to automatically discover the CROFs.

Constraint solving A $O(n^3)$ time type inference algorithm for Abadi and Cardelli's type system $\mathbf{Ob}_{1<:\mu}$ (only invariant fields) was given by Palsberg [20]; later, an $O(n^2)$ time algorithm for the same problem was given by Henglein [8]. 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., $([\,], [\ell^0 : [\,]])$). 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 [20, 12, 24, 22, 23].

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 $(B, B') \in R$ (covariance closed)

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 $(B, B') \in R$ (agree on shared invariant 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:

$$(V, [\ell^+ : B]) \qquad (V, [\ell^+ : B'])$$

where V is a type variable, $B = [m^0 : [\,]]$, and $B' = [m^0 : [m^0 : [\,]]]$. The key property of B, B' is that they don't have a common lower bound. Applying the four closure rules above will only add reflexive constraints. Hence, 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:

$$(A, B) \qquad (A, B')$$

This is not possible: there is no subtype of both B and B' in this system. With only covariant fields, the set of types form a lattice. Once invariant fields are introduced, not all pairs of types have lower bounds. In our setting with both covariant and invariant fields, all pairs of types have a least upper bound. 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 $[\ell^+ : B]$ and $[\ell^+ : B']$ must have a lower bound, it must also be the case that B and B' 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 technique that employs a convenient characterization of the subtyping order (Lemma 2.9). 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 result that type inference is P-hard indicates there are no fast NC-class parallel algorithms for the type inference problem, unless NC = P.

Our prototype implementation, already showcased above, works with a version of an 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 [17, 32, 9, 6, 3].

1.3 Related Work

One of the first uses of annotations such as $+$, often called *variance annotations*, can be found in Pierce and Sangiorgi's paper [25] 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 "$-$". The main problem is that for a system with invariant, covariant, and contravariant fields, not all pairs of types have least upper bounds. For example, consider the types

$$\begin{array}{rcl} C & = & [l^0 : A] \\ C' & = & [l^0 : B] \\ A & = & [\,] \\ B & = & [m^0 : [\,]]. \end{array}$$

Notice that $B \leq A$. Here are two incomparable upper bounds of C, C':

$$\begin{array}{c} [l^+ : A] \\ [l^- : B]. \end{array}$$

However, there is no *least* upper bound of C, C'. So, with both covariant and contravariant fields, the types do *not* form a lattice.

Bugliesi and Pericas-Geertsen [5] studied a type system, called split types, in which *two* types are associated with every updateable record field. One of the field types behaves in a contravariant way, while the other behaves in a covariant way. In this approach, the subtyping relation is a lattice, type inference is straightforward and can be done in $O(n^3)$ time, and both read-only and write-only fields are supported. Our result suggests that type inference becomes more complicated when using the more traditional approach of associating a single type with each field.

Rémy [28] used covariant and contravariant fields in a calculus with object extension, and depth and width subtyping. His language is explicitly typed.

Igarashi and Viroli [11] used variance annotations to control subtyping between different instantiations of a generic class, and to specify the visibility of fields and methods. Their example language is explicitly typed.

Igarashi and Kobayashi [10] showed how to infer types with annotations about the *uses* of communication channels in concurrent programs. A use is either 0 (never used), 1 (used at most once), or ω (used arbitrarily). The set of uses forms an algebra with operations such as $0 + 1 = 1$. Depending on the use annotations of a record type, the fields may enjoy covariant or contravariant subtyping. Their approach differs from ours in the use of variance variables (called *use variables*), whose value determines whether a channel type is contravariant, covariant or invariant. Because these are variables, structural decomposition (closure) is sometimes be suspended until some of these variables receive values, which requires a form of conditional constraints. Our approach does not use conditional constraints and therefore appear simpler. Another notable difference between their type inference problem and ours is that their type inference problem uses finite types without width subtyping and with all of covariance, contravariance, and invariance, while ours uses recursive types and width subtyping, but only covariance and invariance. It remains to be seen whether it is possible to extend their techniques to handle width subtyping and recursive types.

Tang and Hofmann [30] studied type inference for a logic of Abadi and Leino, for the purpose of helping with automatic generation of verification conditions [31]. They use a subtyping relation for object types in which fields are invariant and methods are covariant. Thus, in their type inference problem it is explicitly specified what is read-only and what is updateable. The two most notable differences between their type inference problem and ours are that (1) they consider finite types while we study recursive types and (2) we enable automatic discovery of CROFs. Their work was carried out independently of ours; the technical approaches have some basic ideas in common. In particular, most of our nine rules for satisfaction-closure seems to have counterparts in Tang and Hofmann's approach.

Our type system does not contain a bottom type, that is, a least type. Adding a bottom type would make the set of types form a lattice, rather than a semilattice, and it would make more programs type check [33]. Type inference with bottom types have been investigated for various type systems [24, 22, 23, 5], and in each case type inference can be done in cubic time in the size of the program. Thus, we can expect type inference for our type system extended with a bottom type to be in cubic time. However, bottom types seem not to be popular. Java does have a top type, that is, a greatest type, called Object, but it does not have a bottom type. Generic Java [4] allows an emulation of a bottom type, as shown in this code snippet suggested by one of the anonymous reviewers:

```
class C<T> {
   public T m() {
      return null;
   }
}
```

where `T` is a type parameter which effectively works as a bottom type (the type of `null`). Notice though that this bottom type does not have its own name, so we cannot declare variables of that type.

Above, we discussed a range of previous work [1, 7, 25, 11, 10, 30, 31] in which notions of covariant fields were added to type systems in order to type check more programs, and yet *none* of those papers have a bottom type in their type systems. Pierce [26] discusses ways in which adding a bottom type to a type system complicates matters considerably, particularly in systems with bounded quantification. Based on these observations, we feel that even though adding a bottom type simplifies type inference, it is well justified to study a type system without a bottom type.

1.4 Examples

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

$$\begin{array}{rcl}
\mathrm{Man} & = & [\mathrm{id} = \varsigma(x)x] \\
\mathrm{Batman} & = & [\mathrm{id} = \varsigma(y)y,\ \mathrm{secretid} = \varsigma(z)z] \\
\mathrm{Phone} & = & [\mathrm{dial} = \varsigma(d)\mathrm{Man}] \\
\mathrm{Batphone} & = & \mathrm{Phone.dial} \Leftarrow \varsigma(e)\mathrm{Batman.id.secretid} \\
\mathrm{Main} & = & \mathrm{Batphone.dial.id}
\end{array}$$

Applying the rules of Section 4.1, we obtain the constraints listed in Figure 1. 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).

The constraint set in Figure 1 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 the types P, Q and the environments E, F:

$$\begin{array}{rcl}
P & = & [\mathrm{id}^{+} : [\,]] \\
Q & = & [\mathrm{id}^{0} : [\mathrm{secretid}^{+} : P],\ \mathrm{secretid}^{0} : P] \\
E & = & \emptyset[d : [\mathrm{dial}^{0} : P]] \\
F & = & \emptyset[e : [\mathrm{dial}^{0} : P]].
\end{array}$$

We can derive $\emptyset \vdash \mathrm{Batphone.dial.id} : [\,]$ as follows, using the type rules to be presented in Section 3. The number to the right of each horizontal line indicates which type rule was used.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E[x : [\mathrm{id}^0 : [\,]]] \vdash x : [\mathrm{id}^0 : [\,]]}{E[x : [\mathrm{id}^0 : [\,]]] \vdash x : [\,]}(5)}{E \vdash \mathrm{Man} : [\mathrm{id}^0 : [\,]]}(2)}{E \vdash \mathrm{Man} : P}(5)}{\emptyset \vdash \mathrm{Phone} : [\mathrm{dial}^0 : P]}(2) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{F[y : Q] \vdash y : Q}{F[y : Q] \vdash y : [\mathrm{secretid}^{+} : P]}(5) \quad \dfrac{F[z : Q] \vdash z : Q}{F[z : Q] \vdash z : P}(5)}{F \vdash \mathrm{Batman} : Q}(2)}{F \vdash \mathrm{Batman} : [\mathrm{id}^{+} : [\mathrm{secretid}^{+} : P]]}(5)}{F \vdash \mathrm{Batman.id} : [\mathrm{secretid}^{+} : P]}(3)}{F \vdash \mathrm{Batman.id.secretid} : P}(3)}{\emptyset \vdash \mathrm{Batphone} : [\mathrm{dial}^0 : P]}(4)}{\emptyset \vdash \mathrm{Batphone.dial} : P}(3)}{\emptyset \vdash \mathrm{Batphone.dial.id} : [\,]}(3)$$

Occurrence	Constraints
x	(U_x, V_x)
Man	$([\mathrm{id}^0 : V_x], V_{Man})$
	$U_x \equiv [\mathrm{id}^0 : V_x]$
y	(U_y, V_y)
z	(U_z, V_z)
Batman	$([\mathrm{id}^0 : V_y,\ \mathrm{secretid}^0 : V_z], V_{Batman})$
	$U_y \equiv [\mathrm{id}^0 : V_y,\ \mathrm{secretid}^0 : V_z]$
	$U_z \equiv [\mathrm{id}^0 : V_y,\ \mathrm{secretid}^0 : V_z]$
Phone	$([\mathrm{dial}^0 : V_{Man}], V_{Phone})$
	$U_d \equiv [\mathrm{dial}^0 : V_{Man}]$
Batphone	$(V_{Phone}, V_{Batphone})$
	$V_{Phone} \equiv U_e$
	$(V_{Phone}, [\mathrm{dial}^0 : V_{Batman.id.secretid}])$
Batman.id	$(V_{Batman}, [\mathrm{id}^+ : V_{Batman.id}])$
Batman.id.secretid	$(V_{Batman.id}, [\mathrm{secretid}^+ : V_{Batman.id.secretid}])$
Batphone.dial	$(V_{Batphone}, [\mathrm{dial}^+ : V_{Batphone.dial}])$
Batphone.dial.id	$(V_{Batphone.dial}, [\mathrm{id}^+ : V_{Batphone.dial.id}])$

Figure 1: Constraints for the example program

Notice the five uses of subsumption:

$$\begin{aligned}
[\mathrm{id}^0 : [\,]] &\leq [\,] \\
[\mathrm{id}^0 : [\,]] &\leq P \\
Q &\leq [\mathrm{secretid}^+ : P] \\
Q &\leq P \\
Q &\leq [\mathrm{id}^+ : [\mathrm{secretid}^+ : P]].
\end{aligned}$$

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

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

where U, W are type variables and ℓ, m are labels of fields. The satisfaction-closure of R and the accompanying lower-bound relation are shown in Figure 2. 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.

Let us now explain how the rules for satisfaction-closure (Definition 5.1) generate the constraints in the table above. 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-closure of R (excerpt)	Lower-bound relation (excerpt)
$(U\ ,\ [\ell^+ : [\ell^+ : [m^+ : [\,]]]])$	$(\,[\ell^+ : [\ell^0 : W]]\ ,\ [\ell^+ : [\ell^+ : [m^+ : [\,]]]]\,)$
$(U\ ,\ [\ell^+ : [\ell^0 : W]])$	$(\,[\ell^0 : W]\ ,\ [\ell^+ : [m^+ : [\,]]]\,)$
$(W\ ,\ [m^+ : [\,]])$.	

Figure 2: The satisfaction-closure (excerpt) of two constraints

satisfaction-consistent, and our algorithm then constructs the following solution $S_{R'}$:

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

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 an 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 partial order $\sqsubseteq$ such that $0 \sqsubseteq +$, $0 \sqsubseteq 0$, and $+ \sqsubseteq +$.

The alphabet Σ of our trees is defined

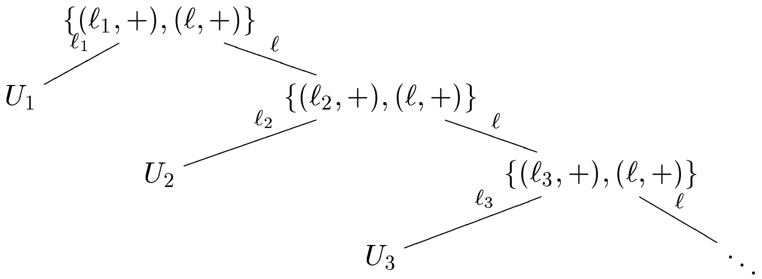
$$\Sigma = \mathcal{TV} \cup \{\sigma \subseteq \mathsf{Labels} \times \mathsf{Variances} \mid (\ell, v), (\ell, v') \in \sigma \Rightarrow v = v'\}.$$

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 $(\ell, v) \in A(\alpha)$ for some $v \in \mathsf{Variances}$ if and only if $A(\alpha\ell)$ is defined. We use A, B, C to range over the set $\mathcal{T}(\Sigma)$ of trees. Notice that types can contain type variables as leaves. We could simplify the development by restricting ourselves to ground types. For example, the notions of subtype-consistency and satisfaction-consistency would then coincide. However, we feel that it is important to show how to handle type variables and therefore we do that in detail.

A *regular* tree is a possibly infinite tree with a finite number of distinct subtrees. Note that trees need not be regular or even finitely branching. Of course, we will be particularly interested in two subsets of $\mathcal{T}(\Sigma)$, the finite trees $\mathcal{T}_{\text{fin}}(\Sigma)$ and the finitely branching and regular trees $\mathcal{T}_{\text{reg}}(\Sigma)$. Some definitions, results, and proofs are given in terms of $\mathcal{T}(\Sigma)$, in such a way that they immediately apply to $\mathcal{T}_{\text{fin}}(\Sigma)$ and $\mathcal{T}_{\text{reg}}(\Sigma)$. 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 $\mathcal{T}_{\text{fin}}(\Sigma)$ or $\mathcal{T}_{\text{reg}}(\Sigma)$.

An example tree is given below, representing the nonregular type

$$[\ell_1^+ : U_1, \ell^+ : [\ell_2^+ : U_2, \ell^+ : [\ell_3^+ : U_3, [\ell^+ : \ldots]]]].$$



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

We will define and reason about subtyping using five related kinds of relations over types:

$$\begin{tikzcd}
\text{simulaton (Definition 2.1)} \arrow[r] \arrow[dr] & \text{subtype-consistent (Definition 2.8)} \arrow[r] & \text{satisfaction-consistent (Definition 5.4)} \\
\text{satisfaction-closed (Definition 5.1)} \arrow[r] & \text{subtype-closed (Definition 2.7)} &
\end{tikzcd}$$

The arrows indicate implication; for example, every simulation is subtype-consistent.

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 $(B', B) \in R$.

$\square$

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, cf. [15].

Similarly, it is easy to show that $\leq$ is a preorder. First, $\leq$ is reflexive because the identity on $\mathcal{T}(\Sigma)$ is a simulation. Next, define the composition $(R \circ R')$ of binary relations R and R' over a set X in the usual way:

$$(x, x') \in (R \circ R') \Leftrightarrow \exists x'' \in X.\ (x, x'') \in R, (x'', x') \in R'.$$

Lemma 2.3 *If R is a 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 $A' = [\ell^{v'} : B', \ldots]$. Then since R is a simulation, we have
  - $A'' = [\ell^{v''} : B'', \ldots]$,
  - $A = [\ell^{v} : B, \ldots]$,
  - $v \sqsubseteq v'' \sqsubseteq v'$,
  - $(B'', B') \in R$,
  - $v' = 0 \Rightarrow (B', B'') \in R$,
  - $(B, B'') \in R$, and
  - $v'' = 0 \Rightarrow (B'', B) \in R$.

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

  We have $(B, B'') \in R$, $(B'', B') \in R$; that is, $(B, B') \in (R \circ R)$.

  If $v' = 0$, then from $v'' \sqsubseteq v'$ we have $v'' = 0$, and so $(B', B'') \in R$, $(B'', B) \in R$, that is, $(B', B) \in (R \circ R)$, as desired.

□

Corollary 2.4 $\leq$ *is transitive.*

Proof. From Lemma 2.3 we have that $(\leq \circ \leq)$ is a simulation, and we also have that $\leq$ is the largest simulation, so $\leq \supseteq (\leq \circ \leq)$. □

Our simulations differ from the simulations typically found in concurrency in that they are all anti-symmetric.

Lemma 2.5 *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, $A = [\ell_i^{v_i} : B_i \ ^{i\in 1..n}]$ and $A' = [\ell_i^{v_i} : B_i' \ ^{i\in 1..n}]$. Therefore $A(\alpha) = \{\ell_i^{v_i} : i \in 1..n\} = A'(\alpha)$ 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 $A = [\ell^v : B, \ldots]$, then by the definition of simulation and the antisymmetry of $\sqsubseteq$, we have $A' = [\ell^v : B', \ldots]$ 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.

□

Lemma 2.6 $\leq$ *is a partial order.*

Proof. This follows immediately from Corollary 2.4 and Lemma 2.5. □

We may apply the principle of *co-induction* [16] 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 2.7 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 $(B', B) \in R$.

$\square$

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 2.8 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. $\square$

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

Lemma 2.9 *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.*

Proof.

- (ii) $\Rightarrow$ (i): Suppose $(A, B) \in R$ and let R' be the subtype-closure of R. We have $(A, B) \in R \subseteq R'$ and that R' is a simulation, so by co-induction we have that $A \leq B$.
- (i) $\Rightarrow$ (iii): 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.

- (iii) $\Rightarrow$ (ii): 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 $A' = [\ell^{v'} : B', \ldots]$, by subtype-consistency A must be of the form $[\ell^{v} : B, \ldots]$, where $v \sqsubseteq v'$. From $v \sqsubseteq v'$ and R' being subtype-closed, we have $(B, B') \in R'$. If $v' = 0$, then from $v \sqsubseteq v'$ we have $v = 0$, so from R' being subtype-closed, we have $(B', B) \in R'$.

□

This immediately suggests an algorithm for testing whether $A \leq B$ for $A, B \in \mathcal{T}_{\text{reg}}(\Sigma)$: 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 the maximum number of pairs of subtrees from A and B, that is, n^2, and can be constructed in n^2 time. Consistency checking is linear, so the algorithm runs in time $O(n^2)$.

Theorem 2.10 *Subtyping on $\mathcal{T}_{\text{reg}}(\Sigma)$ is decidable in $O(n^2)$ time.*

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

3 An Abadi-Cardelli Object Calculus

We now present an extension of the Abadi-Cardelli object calculus called $\mathbf{Ob}_{1<:\mu}$ [1].

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 $\mathbf{Ob}_{1<:\mu}$ [1]: a term in $\mathbf{Ob}_{1<:\mu}$ 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 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 $\varsigma(y)b$, preserving the variance annotation. 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). The semantics is confluent [1].

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 $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

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

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

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

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

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

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 program a is *well-typed* if $\emptyset \vdash a : A$ is derivable for some A. The following result can be proved by a well-known technique [19, 34].

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

The type inference problem for our extension of an 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 over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ is solvable if and only if there is a substitution S such that for all $(A, B) \in R$, we have $S(A) \leq S(B)$. In the remainder of the paper, we use R to range over finite relations on $\mathcal{T}_{\mathrm{reg}}(\Sigma)$. We will refer to each $(A, B) \in R$ as a constraint, and to R as a set of constraints. For a finite relation R on $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, the size of R is the sum of the sizes of occurrences of types in R. The size of a type A is the number of distinct subtrees of A.

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 $E'(x) = A'$. Notice that the definition of $E' \leq E$ allows E' to have a larger domain that E. 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, $E \vdash a : A$, $A \leq [\ell^+ : C]$;*
- *$c = [\ell_i^{v_i} = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, 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. (If c' occurs more than once in c, then $V_{c'}$ is 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 constraint

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

  - 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

$$([\ell_i^{v_i} : V_{b_i} \ ^{i \in 1..n}], V_{[\ell_i^{v_i} = \varsigma(x_i)b_i \ ^{i \in 1..n}]}) \tag{8}$$

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

$$U_{x_j} \equiv [\ell_i^{v_i} : V_{b_i} \ ^{i \in 1..n}]. \tag{9}$$

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

$$(V_a, V_{(a.\ell \Leftarrow \varsigma(x)b)}) \tag{10}$$

$$V_a \equiv U_x \tag{11}$$

$$(V_a, [\ell^0 : V_b]). \tag{12}$$

$\square$

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

Our definition of constraint generation is "global." One can also specify constraint generation in a local way, using inference rules and using existential quantifiers to represent local ("fresh") type variables, see, e.g., [29]. However, for the type system in this paper, there is no serious disadvantage to using the global approach.

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 $S(E_c) \vdash c : S(V_c)$.*

Proof. The proof is by induction on the structure of c. In (ii), 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 $\mathcal{C}(c) = \{(U_x, V_x)\}$.
  - i) 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 4.2, $E(x) \leq C$, so S_c is a solution to $\mathcal{C}(c)$.
  - ii) By (1), $S(E_c) \vdash c : S(U_x)$.
    And since $S(U_x) \leq S(V_x) = S(V_c)$, we have $S(E_c) \vdash c : S(V_c)$ by (5).
- If $c = a.\ell$, then $E_c = E_a$ and $\mathcal{C}(c) = \mathcal{C}(a) \cup \{(V_a, [\ell^+ : V_{a.\ell}])\}$.
  - i) By Lemma 4.2, for some A, $E \vdash a : A$, $A \leq [\ell^+ : C]$.
    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 $S_c = S_a \cup \{V_{a.\ell} := C\}$. Then S_c solves $\mathcal{C}(c)$, $S_c(V_c) = S_c(V_{a.\ell}) = C$, and $S_c(E_c) = S_a(E_a) \subseteq E$.
  - ii) By induction, $S(E_a) \vdash a : S(V_a)$.
    Since $S(V_a) \leq S([\ell^+ : V_{a.\ell}])$, by (5) we have $S(E_a) \vdash a : S([\ell^+ : V_{a.\ell}])$.
    Then by (3), $S(E_a) \vdash a.\ell : S(V_{a.\ell})$.
    Since $S(V_{a.\ell}) = S(V_c)$, 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 $c = [\ell_i^{v_i} = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, then $E_c = \bigcup_{i\in 1..n}(E_{b_i} \backslash x_i)$, and

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

  i) By Lemma 4.2, for some A, and some B_i for $i \in 1..n$, we have $E[x_i : A] \vdash b_i : B_i$ and $A = [\ell_i^{v_i} : B_i \ ^{i\in 1..n}] \leq C$.
  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 $S_{b_i}(E_{b_i}) \subseteq E[x_i : A]$.
  Let $S_c = (\bigcup_{i\in 1..n} S_{b_i}) \cup \{V_c := C\} \cup \{U_{x_i} := A \mid i \in 1..n\}$.
  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, 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).
  Then 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.

  ii) By induction, $S(E_{b_j}) \vdash b_j : S(V_{b_j})$ for all $j \in 1..n$.
  By weakening, $S(E_c \cup \{x_j : U_{x_j}\}) \vdash b_j : S(V_{b_j})$ for all $j \in 1..n$.
  Since S solves $\mathcal{C}(c)$, $S(U_{x_j}) = S([\ell_i^{v_i} : V_{b_i} \ ^{i\in 1..n}])$ for all $j \in 1..n$.
  Then by (2), $S(E_c) \vdash c : S([\ell_i^{v_i} : V_{b_i} \ ^{i\in 1..n}])$.
  Finally $S([\ell_i^{v_i} : V_{b_i} \ ^{i\in 1..n}]) \leq S(V_c)$, so we have $S(E_c) \vdash c : S(V_c)$ by (5).

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

$$\mathcal{C}(c) = \mathcal{C}(a) \cup \mathcal{C}(b) \cup \{(V_a, V_c), V_a \equiv U_x, (V_a, [\ell^0 : V_b])\}.$$

  i) By Lemma 4.2, for some A and B, $E \vdash a : A$, $E[x : A] \vdash b : B$, $A \leq [\ell^0 : B]$, 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 $S_c(E_c) \subseteq E$.

  ii) 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 (4), $S(E_c) \vdash c : S(V_a)$, and by (5), $S(E_c) \vdash c : S(V_c)$.

□

4.2 From Constraint Solving to Type Inference

The following result is proved by a method similar to the one used by Palsberg [20] and Palsberg and Jim [21].

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

Proof. It is straightforward to show that any constraint set over $\mathcal{T}_{\rm reg}(\Sigma)$ can be simplified, in a solution-preserving way, such that each constraint is of the form (W, W') where W and W' are of the forms V or $[\ell_i^{v_i} : V_i \ {}^{i\in 1..n}]$, where $v_i \in \{0, +\}$, and where $V, V_1, \ldots, V_n$ are variables. The advantage of this simplification is that object types are not nested syntactically. Let R be such a simplified constraint set. Define

$$
\begin{array}{lll}
a^R = [& \ell_V^0 & = \varsigma(x)(x.\ell_V) \\
& & \quad \text{for each variable } V \text{ in } R \\
& \ell_Q^0 & = \varsigma(x)[\ell_i^{v_i} = \varsigma(y)(x.\ell_{V_i}) \ {}^{i\in 1..n}] \\
& & \quad \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ {}^{i\in 1..n}] \\
& m^0_{Q,\ell_j} & = \varsigma(x)((x.\ell_{V_j} \Leftarrow \varsigma(y)(x.\ell_Q.\ell_j)).\ell_Q) \\
& & \quad \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ {}^{i\in 1..n}] \\
& & \quad \text{and for each } j \in 1..n \\
& k^0_{Q,\ell_j} & = \varsigma(x)((x.\ell_Q).\ell_j \Leftarrow \varsigma(y)(x.\ell_{V_j})) \\
& & \quad \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ {}^{i\in 1..n}] \\
& & \quad \text{and for each } j \in 1..n \text{ with } v_j = 0 \\
& \ell^0_{(W,W')} & = \varsigma(x)((x.\ell_{W'} \Leftarrow \varsigma(y)(x.\ell_W)).\ell_W) \\
& & \quad \text{for each constraint } (W, W') \in R \\
]
\end{array}
$$

Notice that a^R can be generated in log space.

We first prove that if R is solvable then a^R is typable. Suppose R has solution S. Define

$$
\begin{array}{llll}
A = [& \ell_V^0 & : S(V) & \text{for each variable } V \text{ in } R \\
& \ell_Q^0 & : S(Q) & \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ {}^{i\in 1..n}] \\
& m^0_{Q,\ell_j} & : S(Q) & \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ {}^{i\in 1..n}] \\
& & & \text{and for each } j \in 1..n \\
& k^0_{Q,\ell_j} & : S(Q) & \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ {}^{i\in 1..n}] \\
& & & \text{and for each } j \in 1..n \text{ with } v_j = 0 \\
& \ell^0_{(W,W')} & : S(W) & \text{for each constraint } (W, W') \in R \\
]
\end{array}
$$

It is straightforward to show that $\emptyset \vdash a^R : A$ is derivable.

We now prove that if a^R is typable, then R is solvable. Suppose a^R is typable. From Theorem 4.4 we get a solution S of $\mathcal{C}(a^R)$.

Notice that each method in a^R binds a variable x. Each of these variables corresponds to a distinct type variable in $\mathcal{C}(a^R)$. Since S is a solution of $\mathcal{C}(a^R)$, and $\mathcal{C}(a^R)$ contains constraints of the form $U_x = [\ldots]$ for each method in a^R (from rule (9)), 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^R as being related to the same type variable, which we will write as U_x.

Notice also that most methods in a^R bind a variable y. None of these variables are used in a^R, and each of them corresponds to a distinct type variable in $\mathcal{C}(a^R)$. They will not play any role in the rest of the proof. For $A = [\ell^v : B, \ldots]$, we define $A{\downarrow}\ell = B$.

Define

$$S_R(V) \quad = \quad S(U_x){\downarrow}\ell_V \qquad \text{for each variable } V \text{ in } R.$$

The definition is justified by Property 1 below.

- **Property 1** If V is a variable in R, then $S(U_x){\downarrow}\ell_V$ is defined.

- **Property 2** For each Q in R of the form $[\ell_i^{v_i} : V_i \quad {}^{i\in 1..n}]$, we have $S(U_x){\downarrow}\ell_Q = [\ell_i^{v_i} : (S(U_x){\downarrow}\ell_{V_i}) \quad {}^{i\in 1..n}]$.

We will proceed by first showing the two properties and then showing that R has solution S_R.

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 $\mathcal{C}(a^R)$, we have from the rules (6) and (7) that S satisfies

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

so

$$S(U_x){\downarrow}\ell_V \leq S(V_{x.\ell_V}) \tag{13}$$

We conclude that since $S(V_{x.\ell_V})$ is defined, also $S(U_x){\downarrow}\ell_V$ is defined.

To see Property 2, let Q be an occurrence in R of the form $[\ell_i^{v_i} : V_i \quad {}^{i\in 1..n}]$. 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 $\mathcal{C}(a^R)$, we have from the rules (6), (7), and (12), that S satisfies

$$\begin{array}{rcll} (U_x & , & V_{x'}) \quad \text{and} \quad (V_{x'} & , \; [\ell_{V_j}^0 : V_{x.\ell_Q.\ell_j}]) & (14)\\ (U_x & , & V_x) \quad \text{and} \quad (V_x & , \; [\ell_Q^+ : V_{x.\ell_Q}]) & (15)\\ (V_{x.\ell_Q} & , & [\ell_j^+ : V_{x.\ell_Q.\ell_j}]) & & (16) \end{array}$$

Thus,

$$\begin{array}{rcll} S(U_x){\downarrow}\ell_Q & \leq & S(V_{x.\ell_Q}) & \text{from (15)}\\ & \leq & [\ell_j^+ : S(V_{x.\ell_Q.\ell_j})] & \text{from (16)}\\ & = & [\ell_j^+ : (S(U_x){\downarrow}\ell_{V_j})] & \text{from (14)} \end{array}$$

Therefore,

$$S(U_x){\downarrow}\ell_Q{\downarrow}\ell_j \quad \leq \quad S(U_x){\downarrow}\ell_{V_j} \tag{17}$$

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

$$([\ell_i^{v_i} : V_{x.\ell_{V_i}} \quad {}^{i\in 1..n}] \quad , \quad V_{[\ell_i^{v_i}=\varsigma(y)(x.\ell_{V_i}) \quad {}^{i\in 1..n}]}) \tag{18}$$

$$U_x \quad \equiv \quad [\ldots \ell_Q^0 : V_{[\ell_i^{v_i}=\varsigma(y)(x.\ell_{V_i}) \quad {}^{i\in 1..n}]} \ldots] \tag{19}$$

Thus, from (18) and (19) we have

$$[\ell_i^{v_i} : S(V_{x.\ell_{V_i}}) \quad {}^{i\in 1..n}] \leq S(V_{[\ell_i^{v_i}=\varsigma(y)(x.\ell_{V_i}) \quad {}^{i\in 1..n}]}) = S(U_x){\downarrow}\ell_Q, \tag{20}$$

so

$$\begin{array}{rcll} S(U_x){\downarrow}\ell_{V_j} & \leq & S(V_{x.\ell_{V_j}}) & \text{from (13)}\\ & \leq & S(U_x){\downarrow}\ell_Q{\downarrow}\ell_j & \text{from (20).} \end{array}$$

From that and (17) we conclude:

$$S(U_x){\downarrow}\ell_Q{\downarrow}\ell_j \quad = \quad S(U_x){\downarrow}\ell_{V_j}. \tag{21}$$

For each $j \in 1..n$ with $v_j = 0$, in the body of the method k_{Q,ℓ_j}, we have the expression $((x'.\ell_Q).\ell_j \Leftarrow \varsigma(y)(x.\ell_{V_j}))$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^R)$, we have from the rules (6), (7), and (12) that S satisfies

$$(U_x \ , \ V_{x'}) \quad \text{and} \quad (V_{x'} \ , \ [\ell_Q^+ : V_{x'.\ell_Q}]) \tag{22}$$

$$(V_{x'.\ell_Q} \ , \ [\ell_j^0 : V_{x.\ell_{V_j}}]) \tag{23}$$

Thus,

$$\begin{array}{rcll} S(U_x)\downarrow\ell_Q & \leq & S(V_{x'.\ell_Q}) & \text{from (22)} \\ & \leq & [\ell_j^0 : S(V_{x.\ell_{V_j}})] & \text{from (23)} \end{array}$$

From that and (20) we have that

- for each $j \in 1..n$ with $v_j = 0$, the variance annotation of the ℓ_j field of $S(U_x)\downarrow\ell_Q$ is 0, and
- for each $j \in 1..n$ with $v_j = +$, the variance annotation of the ℓ_j field of $S(U_x)\downarrow\ell_Q$ is $+$.

Therefore, by (21), we have that $S(U_x)\downarrow\ell_Q = [\ell_i^{v_i} : (S(U_x)\downarrow\ell_{V_i}) \ ^{i \in 1..n}]$, that is, Property 2.

We can summarize Property 1 and 2 as follows.

- **Property 3** If W is a left-hand side or a right-hand side of a constraint in R, then $S(U_x)\downarrow\ell_W$ is defined and $S_R(W) = S(U_x)\downarrow\ell_W$.

We will now show that R has solution S_R.

Consider a constraint (W, W') in R. The body of the method $\ell_{(W,W')}$ contains the expression $x'.\ell_{W'} \Leftarrow \varsigma(y)(x.\ell_W)$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^R)$, we have from the rules (6), (12), (6) and (7) that S satisfies

$$(U_x \ , \ V_{x'}) \quad \text{and} \quad (V_{x'} \ , \ [\ell_{W'}^0 : V_{x.\ell_W}]) \tag{24}$$

$$(U_x \ , \ V_x) \quad \text{and} \quad (V_x \ , \ [\ell_W^+ : V_{x.\ell_W}]) \tag{25}$$

We conclude

$$\begin{array}{rcll} S_R(W) & = & S(U_x)\downarrow\ell_W & \text{from Property 3} \\ & \leq & S(V_{x.\ell_W}) & \text{from (25)} \\ & = & S(U_x)\downarrow\ell_{W'} & \text{from (24)} \\ & = & S_R(W') & \text{from Property 3} \end{array}$$

□

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 if its satisfaction-closure is satisfaction-consistent (Theorem 5.13).

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 a relation L on types such that

i)	if $(A, B) \in R$, then $(A, A), (B, B) \in R$	(R is reflexive)
ii)	if $(A, B), (B, C) \in R$, then $(A, C) \in R$	(R is transitive)
iii)	if $(A, B) \in R$, then $(A, B) \in L$	(subtypes have a lower bound)
iv)	if $(A, B) \in L$, then $(B, A) \in L$	(L is symmetric)
v)	if $(A, B) \in L$, and $(B, C) \in R$, then $(A, C) \in L$	(a lower bound extends to supertypes)
vi)	if $([\ell^+ : B, \ldots], [\ell^+ : B', \ldots]) \in R$, then $(B, B') \in R$	(R is covariance-closed)
vii)	if $([\ell^+ : B, \ldots], [\ell^+ : B', \ldots]) \in L$, then $(B, B') \in L$	(L is covariance-closed)
viii)	if $([\ell^0 : B, \ldots], [\ell^+ : B', \ldots]) \in L$, then $(B, B') \in R$	(agree on shared invariant fields)
ix)	if $([\ell^0 : B, \ldots], [\ell^0 : B', \ldots]) \in L$, then $(B, B') \in R$.	(agree on shared invariant fields)

□

Notice that for a given relation R, we can construct a sat-closed relation R' that includes R, by letting R' consist of all pairs of subtrees of types that occur in R.

Notice also 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 smallest 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 that 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 $(R, \emptyset)$ such that R' is sat-closed. 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 analyze the complexity of the fixed-point computation using a technique and a theorem of McAllester [14], as follows. McAllester [14, Section 10] presents a technique for encoding record types, such as the ones we use here, without using elipses. The encoding entails a slight reformulation of the rules for sat-closure; however, the transitivity rule for R (sat-closure rule (ii)) remains the key source of complexity. We can then apply a theorem of McAllester [14, Theorem 1] to get that the fixed-point computation takes $O(n^3)$ time where n is the size of R.

Define that (R, L) is solvable iff there exists a mapping S such that

- if $(A, B) \in R$, then $S(A) \leq S(B)$, and
- if $(A, B) \in L$, then there exists a type C, such that $C \leq S(A)$ and $C \leq S(B)$.

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

Proof. Since the sat-closure of a relation R contains R, it follows that any solution of the sat-closure of R is also a solution of R.

To prove the converse, we will prove the following more general property:

> Any solution of R is also a solution of the pairwise-$\subseteq$-smallest pair (R', L) that contains $(R, \emptyset)$ such that R' is sat-closed.

We proceed by induction on the fixed-point computation of (R', L) from $(R, \emptyset)$. Each iteration begins with a pair (R_1, L_1), and at the end of an iteration step, some pairs may have been added to R_1 and L_1. We need to show that after each iteration, any solution of R is also a solution of the resulting pair of relations.

In the base case, we have that any solution of R is also a solution of $(R, \emptyset)$.

In the induction step, suppose (R_1, L_1) has a solution S. For each of the nine sat-closure rules, we need to show that the rules will only add pairs to R_1 and L_1 that have solution S:

- Assume rule (i) has been used. The relation $\leq$ is reflexive, so $S(A) \leq S(A)$ and $S(B) \leq S(B)$, so S is still a solution.

- Assume rule (ii) has been used. From the induction hypothesis, we have $S(A) \leq S(B)$ and $S(B) \leq S(C)$. The relation $\leq$ is transitive, so $S(A) \leq S(C)$, so S is still a solution.

- Assume rule (iii) has been used. From the induction hypothesis, we have $S(A) \leq S(B)$, so a common $\leq$-lower bound for $S(A)$ and $S(B)$ is $S(A)$, so S is still a solution.

- Assume rule (iv) has been used. From the induction hypothesis, we have that $S(A)$ and $S(B)$ have a common $\leq$-lower bound, so $S(B)$ and $S(A)$ have a common $\leq$-lower bound, so S is still a solution.

- Assume rule (v) has been used. From the induction hypothesis, we have that $S(A)$ and $S(B)$ have a common $\leq$-lower bound D, and that $S(B) \leq S(C)$. The relation $\leq$ is transitive, so $D \leq C$, and hence D is a common $\leq$-lower bound for $S(A)$ and $S(C)$; so S is still a solution.

- Assume rule (vi) has been used. From the induction hypothesis, we have that $[\ell^+ : S(B), \ldots] \leq [\ell^+ : S(B')]$. From Lemma 2.2 we have $S(B) \leq S(B')$, so S is still a solution.

- Assume rule (vii) has been used. From the induction hypothesis, we have that $[\ell^+ : S(B), \ldots]$ and $[\ell^+ : S(B')]$ have a common $\leq$-lower bound D. From Lemma 2.2 we have that $D = [\ell^v : A]$ where $A \leq S(B)$ and $A \leq S(B')$, so A is a common $\leq$-lower bound for $S(B)$ and $S(C)$; so S is still a solution.

- Assume rule (viii) has been used. From the induction hypothesis, we have that $[\ell^0 : S(B), \ldots]$ and $[\ell^+ : S(B')]$ have a common $\leq$-lower bound D. From Lemma 2.2 we have that $D = [\ell^0 : A]$ where $A = S(B)$ and $A \leq S(B')$, so $S(B) = A \leq S(B')$; so S is still a solution.

- Assume rule (ix) has been used. From the induction hypothesis, we have that $[\ell^0 : S(B), \ldots]$ and $[\ell^0 : S(B')]$ have a common $\leq$-lower bound D. From Lemma 2.2 we have that $D = [\ell^0 : A]$ where $A = S(B)$ and $A = S(B')$, so $S(B) = A = S(B')$, and therefore $S(B) \leq S(B')$; so S is still a solution.

$\square$

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* $(B', B) \in R$.

Proof. For Property **(A)** there are four cases. If $v = +, v' = +$, then from sat-closure rule (vii) we have $(B, B') \in L$. If $v = 0, v' = +$, or $v = 0, v' = 0$, then from sat-closure rules (viii) or (ix), we have $(B, B') \in R$, and then from sat-closure rule (iii) we have $(B, B') \in L$. Finally, if $v = +, v' = 0$, then from sat-closure rule (iv) we have $([\ell^{v'} : B', \ldots], [\ell^v : B, \ldots]) \in L$, so from sat-closure rule (viii) we have $(B', B) \in R$, so from sat-closure rule (iii) we have $(B', B) \in L$, and hence from sat-closure rule (iv) we have $(B, B') \in L$. So, in all cases $(B, B') \in L$.

For Property **(B)**, notice that from sat-closure rule (v), $(A, A') \in L$, and $(A', A_2) \in R$, we have $(A, A_2) \in L$. From sat-closure rule (iv) and $(A, A_2) \in L$, we have $(A_2, A) \in L$. From sat-closure rule (v), $(A_2, A) \in L$, and $(A, A_1) \in R$, we have $(A_2, A_1) \in L$, so from sat-closure rule (iv) we have $(A_1, A_2) \in L$.

For Property **(C)**, notice that from sat-closure rule (i) and $(A, A_1) \in R$ we have $(A, A) \in R$, so from sat-closure rule (iii) we have $(A, A) \in L$. From Property **(B)**, $(A, A) \in L$, and $(A, A_1), (A, A_2) \in R$ we conclude that $(A_1, A_2) \in L$.

For Property **(D)** there are three cases. If $v = +, v' = +$, then from sat-closure rule (vi) we have $(B, B') \in R$. If $v = 0, v' = +$, or $v = 0, v' = 0$, then from sat-closure rule (iv) and $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in R$, we have $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in L$, so from sat-closure rules (viii) or (ix), we have $(B, B') \in R$.

For Property **(E)**, we have from sat-closure rule (iii) that $([\ell^0 : B, \ldots], [\ell^0 : B', \ldots]) \in L$, so from sat-closure rule (iv), we have $([\ell^0 : B', \ldots], [\ell^0 : B, \ldots]) \in L$, so from sat-closure rule (ix), we have $(B', B) \in R$. □

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. □

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.*

Proof. Immediate. □

5.3 Main Result

We first list the terminology used in the later definitions. Recall that $\mathcal{T}_{\text{reg}}(\Sigma)$ is the set of recursive types considered in this paper.

$$\begin{array}{rcl} \mathsf{Types} & = & \mathcal{T}_{\text{reg}}(\Sigma) \\ \mathsf{States} & = & \mathrm{P}(\mathsf{Types}) \\ \mathsf{RelTypes} & = & \mathrm{P}(\mathsf{Types} \times \mathsf{Types}) \\ \mathsf{RelStates} & = & \mathrm{P}(\mathsf{States} \times \mathsf{States}) \end{array}$$

To define the solution S_R, we will need the following notation. We use g, h to range over sets of types. Then we make the following definitions.

$$\begin{array}{rcl} g.\ell & = & \{B \mid \exists A \in g.\ A = [\ell^v : B, \ldots]\}. \\ \text{above}_R(g) & = & \{B \mid \exists A \in g.\ (A, B) \in R\}. \\ \text{ABOVE}_R(R') & = & \{(\text{above}_R(\{A\}), \text{above}_R(\{B\})) \mid (A, B) \in R'\}. \\ \text{Var}(g, \ell) & = & \sqcap\{v \mid \exists A \in g.\ (\ell, v) \in A(\epsilon)\}. \end{array}$$

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

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

We define $\text{LV} : \mathsf{States} \to \mathcal{P}(\mathsf{Labels} \times \mathsf{Variances})$ such that for any set g of types, $\text{LV}(g)$ is the set of labels and variances implied by g, namely

$$\text{LV}(g) = \{(\ell, v) \mid v = \text{Var}(g, \ell)\}.$$

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:

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

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 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:

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

We have that

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

The following two lemmas express a fundamental property of Type_R.

Lemma 5.6 $\mathrm{Type}_R(g)(\ell\alpha) = \mathrm{Type}_R(\delta_R(g)(\ell))(\alpha)$.

Proof.

$$\begin{array}{rcll} \mathrm{Type}_R(g)(\ell\alpha) & = & \mathrm{LV}(\delta^*_R(g)(\ell\alpha)) & \text{(Definition of } \mathrm{Type}_R) \\ & = & \mathrm{LV}(\delta^*_R(\delta_R(g)(\ell))(\alpha)) & \text{(Definition of } \delta^*_R) \\ & = & \mathrm{Type}_R(\delta_R(g)(\ell))(\alpha) & \text{(Definition of } \mathrm{Type}_R) \end{array}$$

□

Lemma 5.7 *If $g.\ell \neq \emptyset$, then* $\mathrm{Type}_R(\delta_R(g)(\ell)) = \mathrm{Type}_R(\mathrm{above}_R(\delta_R(g)(\ell)))$.

Proof.

$$\begin{array}{rcll} \mathrm{Type}_R(\delta_R(g)(\ell)) & = & \mathrm{Type}_R(\mathrm{above}_R(g.\ell)) & \text{(Definition of } \delta_R \text{ and } g.\ell \neq \emptyset) \\ & = & \mathrm{Type}_R(\mathrm{above}_R(\mathrm{above}_R(g.\ell))) & \text{(Definition of } \mathrm{above}_R) \\ & = & \mathrm{Type}_R(\mathrm{above}(\delta_R(g)(\ell))) & \text{(Definition of } \delta_R \text{ and } g.\ell \neq \emptyset) \end{array}$$

□

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

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

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

Example To illustrate the definition of S_R, let us go into the details of the construction of S_R for the sat-closed and sat-consistent constraint set R of Figure 2. From Lemma 5.6 and Lemma 5.7, we have that if $\mathrm{LV}(g) = \{(\ell, v)\}$, then

$$\mathrm{Type}_R(g) = [\ell^v : \mathrm{Type}_R(\delta_R(g)(\ell))] = [\ell^v : \mathrm{Type}_R(\mathrm{above}_R(\delta_R(g)(\ell)))]. \quad (26)$$

Let us now consider the relation R in Figure 2 and first note that we have:

$$\begin{array}{rcl} \mathrm{above}_R(\{U\}) & = & \{\, U, [\ell^+ : [\ell^+ : [m^+ : [\,]]]], [\ell^+ : [\ell^0 : W]] \,\} \\ \mathrm{above}_R(\{W\}) & = & \{\, W, [m^+ : [\,]] \,\}. \end{array}$$

Next, let us calculate $S_R(W)$:

$$\begin{array}{rcll} S_R(W) & = & \mathrm{Type}_R(\mathrm{above}_R(\{W\})) & \text{(Definition of } S_R) \\ & = & \mathrm{Type}_R(\{\, W, [m^+ : [\,]] \,\}) & \text{(Definition of } \mathrm{above}_R) \\ & = & [m^+ : [\,]] & \text{(From 26).} \end{array}$$

Finally, let us calculate $S_R(U)$:

$$\begin{array}{rcll} S_R(U) & = & \mathrm{Type}_R(\mathrm{above}_R(\{U\})) & \text{(Definition of } S_R) \\ & = & \mathrm{Type}_R(\{\, U, [\ell^+ : [\ell^+ : [m^+ : [\,]]]], [\ell^+ : [\ell^0 : W]] \,\}) & \text{(Definition of } \mathrm{above}_R) \\ & = & [\ell^+ : \mathrm{Type}_R(\mathrm{above}_R(\{\, [\ell^+ : [m^+ : [\,]]], [\ell^0 : W] \,\}))] & \text{(From 26)} \\ & = & [\ell^+ : \mathrm{Type}_R(\{\, [\ell^+ : [m^+ : [\,]]]], [\ell^0 : W] \,\})] & \text{(Definition of } \mathrm{above}_R) \\ & = & [\ell^+ : [\ell^0 : \mathrm{Type}_R(\mathrm{above}_R(\{\, [m^+ : [\,]], W \,\}))]] & \text{(From 26)} \\ & = & [\ell^+ : [\ell^0 : \mathrm{Type}_R(\{\, [m^+ : [\,]], W \,\})]] & \text{(Definition of } \mathrm{above}_R) \\ & = & [\ell^+ : [\ell^0 : [m^+ : [\,]]]] & \text{(From 26)} \end{array}$$

The first step in proving that S_R is a solution to R 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 $\mathrm{Var}(g', \ell) \sqsubseteq \mathrm{Var}(h', \ell)$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$.
- For some ℓ and $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$, $\mathrm{Var}(g', \ell) = 0$, and $\mathrm{Var}(h', \ell) = 0$.

The next two lemmas (Lemma 5.8 and Lemma 5.9) are key ingredients in the proof of Lemma 5.10. Lemma 5.8 states fundamental relationship between TYPE_R, $\mathcal{A}$, and $\mathcal{B}_R$. The intuition behind Lemma 5.8 is that it doesn't matter whether we propagate information about subtype relationships (using $\mathcal{B}_R$ and $\mathcal{A}$) before or after we collapse the sets of types in each state to single types (using TYPE_R).

Lemma 5.8 *The following diagram commutes:*

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

Proof. To prove $\mathrm{TYPE}_R \circ \mathcal{B}_R \subseteq \mathcal{A} \circ \mathrm{TYPE}_R$, suppose $\mathcal{R} \in \mathsf{RelStates}$ and $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$. 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.

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

Second, suppose for some ℓ and $(g', h') \in \mathcal{R}$, such that $\mathrm{Var}(g', \ell) \sqsubseteq \mathrm{Var}(h', \ell)$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. From $(g', h') \in \mathcal{R}$, we have $(\mathrm{Type}_R(g'), \mathrm{Type}_R(h')) \in \mathrm{TYPE}_R(\mathcal{R})$. From Lemma 5.6 and $g = \delta_R(g')(\ell)$, we have

$$\begin{array}{rcll} \mathrm{Type}_R(g')(\ell\alpha) & = & \mathrm{Type}_R(\delta_R(g')(\ell))(\alpha) & \text{(From Lemma 5.6)} \\ & = & \mathrm{Type}_R(g)(\alpha) & \text{(From the definition of } g\text{)} \\ & = & A(\alpha) & \text{(From the definition of } A\text{)} \end{array}$$

so there must exist $v_A = \mathrm{Var}(g', \ell)$ such that

$$\mathrm{Type}(g') = [\ell^{v_A} : A, \ldots].$$

Similarly, there must exist $v_B = \mathrm{Var}(h', \ell)$ such that

$$\mathrm{Type}(h') = [\ell^{v_B} : B, \ldots].$$

From the characterizations of $\text{Type}_R(g')$ and $\text{Type}_R(h')$, and from the definition of $\mathcal{A}$, we have $(A, B) \in \mathcal{A} \circ \text{TYPE}_R(\mathcal{R})$.

Third, suppose 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 $\text{Var}(h', \ell) = 0$. From $(h', g') \in \mathcal{R}$, we have $(\text{Type}_R(h'), \text{Type}_R(g')) \in \text{TYPE}_R(\mathcal{R})$. From the definition of Type_R we have $\text{Type}_R(g') = [\ell^0 : A, \ldots]$ and $\text{Type}_R(h') = [\ell^0 : B, \ldots]$, so, by the definition of $\mathcal{A}$, we have $(A, B) \in \mathcal{A} \circ \text{TYPE}_R(\mathcal{R})$.

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

First, suppose $(A, B) \in \text{TYPE}_R(\mathcal{R})$. There must exist g and h such that $A = \text{Type}_R(g)$, $B = \text{Type}_R(h)$, 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 $(A, B) \in \text{TYPE}_R \circ \mathcal{B}_R$.

Second, suppose for some ℓ, v, and v', such that $v \sqsubseteq v'$, we have $([\ell^v : A, \ldots], [\ell^{v'} : B, \ldots]) \in \text{TYPE}_R(\mathcal{R})$. There must exist g' and h' such that $\text{Type}_R(g') = [\ell^v : A, \ldots]$, $\text{Type}_R(h') = [\ell^{v'} : B, \ldots]$, $(g', h') \in \mathcal{R}$, $\text{Var}(g', \ell) = v$, and $\text{Var}(h', \ell) = v'$. 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 $\mathcal{B}_R$. And by the definition of Type_R, $A = \text{Type}_R(g)$ and $B = \text{Type}_R(h)$, so $(A, B) \in \text{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$ as desired.

Third, suppose for some ℓ, we have $([\ell^0 : B, \ldots], [\ell^0 : A, \ldots]) \in \text{TYPE}_R(\mathcal{R})$. There must exist g' and h' such that $\text{Type}_R(g') = [\ell^0 : A, \ldots]$, $\text{Type}_R(h') = [\ell^0 : B, \ldots]$, and $(h', g') \in \mathcal{R}$. By the definition of Type_R, $\text{Var}(g', \ell) = 0$ and $\text{Var}(h', \ell) = 0$. 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 $\mathcal{B}_R$. And by the definition of Type_R, $A = \text{Type}_R(g)$ and $B = \text{Type}_R(h)$, so $(A, B) \in \text{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$ as desired. $\square$

Lemma 5.9 *Suppose R is sat-closed. For all n, we have that if $(g, h) \in \mathcal{B}_R^n \circ \text{ABOVE}_R(R)$, then $g \supseteq h$.*

Proof. Let L be the lower-bound relation for R. We will prove the following more general property:

For all n,
if $(g, h) \in \mathcal{B}_R^n \circ \text{ABOVE}_R(R)$, then

- *$g \supseteq h$, and*
- *if $A_1, A_2 \in g$, then $(A_1, A_2) \in L$.*

We proceed by induction on n. In the base case of $n = 0$, 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 $(B, C) \in R$, and together with $(A, B) \in R$ and transitivity of R (sat-closure rule (ii)), we have $(A, C) \in R$, so $C \in g$, and hence $g \supseteq h$. Suppose $A_1, A_2 \in g$. We have $(A, A_1), (A, A_2) \in R$, so from Lemma 5.3, Property **(C)**, we have that $(A_1, A_2) \in L$.

In the induction step, suppose $(g, h) \in \mathcal{B}_R^{n+1} \circ \text{ABOVE}_R(R)$. From the definition of $\mathcal{B}_R$ we have that there are three cases. First, suppose $(g, h) \in \mathcal{B}_R^n \circ \text{ABOVE}_R(R)$. From the induction hypothesis we have that the two desired properties are satisfied.

Second, suppose for some ℓ and $(g', h') \in \mathcal{B}_R^n \circ \text{ABOVE}_R(R)$, such that $\text{Var}(g', \ell) \sqsubseteq \text{Var}(h', \ell)$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. From the induction hypothesis we have $g' \supseteq h'$. From $g' \supseteq h'$ and the definition on δ_R it is immediate that $g \supseteq h$. Suppose $A_1, A_2 \in g$. From $g = \delta_R(g')(\ell)$ and the definition of δ_R, we have that there exists $[\ell^v : A_1', \ldots], [\ell^{v'} : A_2', \ldots] \in g'$, and $(A_1', A_1), (A_2', A_2) \in R$. From the induction hypothesis we have $([\ell^v : A_1', \ldots], [\ell^{v'} : A_2', \ldots]) \in L$, so from Lemma 5.3, property **(A)**, we have that $(A'_1, A'_2) \in L$. From Lemma 5.3, Property **(B)**, $(A'_1, A'_2) \in L$, $(A'_1, A_1), (A'_2, A_2) \in R$ we have $(A_1, A_2) \in L$.

Third, suppose for some ℓ and $(h', g') \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$, $\mathrm{Var}(g', \ell) = 0$, and $\mathrm{Var}(h', \ell) = 0$. From the definition of δ_R and $\mathrm{Var}(g', \ell) = 0$, we have that there exists at least one type $[\ell^0 : A, \ldots] \in g'$. From the induction hypothesis we have $h' \supseteq g'$. Thus, $[\ell^0 : A, \ldots] \in h'$. To prove $g \supseteq h$, suppose $B \in h$. By the definition of δ_R and $\mathrm{Var}(h', \ell) = 0$, there must exist a type $[\ell^0 : B', \ldots] \in h'$ such that $(B', B) \in R$. From the induction hypothesis and $[\ell^0 : A, \ldots], [\ell^0 : B', \ldots] \in h'$, we have $([\ell^0 : A, \ldots], [\ell^0 : B', \ldots]) \in L$. From sat-closure rule (ix), we have $(A, B') \in R$. Therefore, from the transitivity of R (sat-closure rule (ii)) and $(A, B'), (B', B) \in R$, we have $(A, B) \in R$, so $B \in g$, and hence $g \supseteq h$. The property that "if $A_1, A_2 \in g$, then $(A_1, A_2) \in L$" can be proved in the same way as in the previous case.

□

Lemma 5.10 *If R is sat-closed, then the subtype-closure of* $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ *is subtype-consistent.*

Proof.

$$\begin{aligned}
&\text{The subtype-closure of } \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R) \\
&= \bigcup_{0 \le n < \infty} \mathcal{A}^n \circ \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R) \qquad \text{(Definition of subtype-closure)} \\
&= \bigcup_{0 \le n < \infty} \mathrm{TYPE}_R \circ \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R) \qquad \text{(Lemma 5.8)} \\
&= \bigcup_{0 \le n < \infty} \quad \bigcup_{(g,h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)} \{(\mathrm{Type}_R(g), \mathrm{Type}_R(h))\} \qquad \text{(Definition of } \mathrm{TYPE}_R\text{)}.
\end{aligned}$$

From Lemma 5.9 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 possibly infinite 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 5.12. The two lemmas 5.11 and 5.12 are the two places where it is needed that a relation is satisfaction-consistent.

Lemma 5.11 *If $A = [\ell^v : B, \ldots]$ appears in R and R is sat-closed and sat-consistent, then*

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

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

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

To prove the direction $\subseteq$, suppose $C \in \mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell)$. From that we have there exists $C' \in (\mathrm{above}_R(\{A\})).\ell$ such that $(C', C) \in R$. From $C' \in (\mathrm{above}_R(\{A\})).\ell$ we have that there exists $[\ell^{v'} : C', \ldots]$ such that $(A, [\ell^{v'} : C', \ldots]) \in R$. From sat-consistency and $(A, [\ell^{v'} : C', \ldots]) \in R$, we have that $v \sqsubseteq v'$. From Lemma 5.3, Property **(D)**, $(A, [\ell^{v'} : C', \ldots]) \in R$, and $v \sqsubseteq v'$, we have that $(B, C') \in R$. From transitivity of R (sat-closure rule (ii)) and $(B, C'), (C', C) \in R$, we have $(B, C) \in R$, so $C \in \mathrm{above}_R(\{B\})$. □

Recall that for any relation R on types, we have defined S_R to be the least substitution such that for every U appearing in R, we have $S_R(U) = \mathrm{Type}_R(\mathrm{above}_R(\{U\}))$.

Lemma 5.12 *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) $S_R(R) = \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$.

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, $S_R(A)(\alpha) = \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha)$.

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

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

$$
\begin{aligned}
& S_R(A)(\alpha) \\
&= S_R(B)(\alpha') \qquad \text{(Definition of } S_R) \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{B\})(\alpha') \qquad \text{(Induction hypothesis)} \\
&= \mathrm{LV}(\delta_R^*(\mathrm{above}_R(\{B\}))(\alpha')) \qquad \text{(Definition of } \mathrm{Type}_R) \\
&= \mathrm{LV}(\delta_R^*(\mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell))(\alpha')) \qquad \text{(Lemma 5.11)} \\
&= \mathrm{LV}(\delta_R^*(\delta_R(\mathrm{above}_R(\{A\}))(\ell))(\alpha')) \qquad \text{(Definition of } \delta_R) \\
&= \mathrm{LV}(\delta_R^*(\mathrm{above}_R(\{A\}))(\ell\alpha')) \qquad \text{(Definition of } \delta_R^*) \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha) \qquad \text{(Definition of } \mathrm{Type}_R \text{ and } \alpha = \ell\alpha').
\end{aligned}
$$

If $\alpha = \ell\alpha'$ and A is a record without an ℓ field, then $S_R(A)(\alpha)$ is undefined. By sat-consistency, 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. □

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

Theorem 5.13 *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 5.10 and Lemma 5.12, we have that the subtype-closure of $S_{R'}(R')$ is subtype-consistent, so from Lemma 2.9, 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 5.14 *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

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

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

Let $\mathcal{T}^0_{\text{reg}}(\Sigma)$ be the subset of $\mathcal{T}_{\text{reg}}(\Sigma)$ where all variance annotations are 0. In [20] it is proved that solvability of SC-systems over $\mathcal{T}^0_{\text{reg}}(\Sigma)$ is P-hard.

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

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

Conversely, suppose that R is solvable over $\mathcal{T}_{\text{reg}}(\Sigma)$, 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 $\mathcal{T}^0_{\text{reg}}(\Sigma)$, □

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

Proof. From the sat-closure rules, it follows that R' is an SC-system as well. It is clear from the definition that $\text{above}_{R'}$ produces only types in $\mathcal{T}^0_{\text{reg}}(\Sigma)$. Also by definition, $\text{Type}_{R'}(g)(\alpha) = \text{LV}(\delta^*_{R'}(g)(\alpha))$, and, by the definition of $\delta_{R'}$, we have that $\delta^*_{R'}(g)(\alpha)$ is a set of types in $\mathcal{T}^0_{\text{reg}}(\Sigma)$ for any type set g and path α. Thus, the function $\text{Type}_{R'}$ produces only types in $\mathcal{T}^0_{\text{reg}}(\Sigma)$. Since $S_{R'}(V) = \text{Type}_{R'}(\text{above}_{R'}(\{V\}))$, we have that $S_{R'}(V) \in \mathcal{T}^0_{\text{reg}}(\Sigma)$. □

7 Conclusion and Future Work

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.

It may be possible to represent types as feature trees and thereby build a connection to solving feature constraints [18]; this is left to future work.

An interesting idea for an alternative approach to type inference that may be able to handle all variances $\{0, +, -\}$ was suggested by one of the anonymous reviewers. The idea is to define a metric $d(A, B)$ on the types, and show that it is a complete metric space and all the constructions are contractive (or non-expansive). Then with the Banach Fixpoint Theorem, we may be able show that there is a solution if the closure is consistent. This approach is similar to [13, 2] and is an interesting idea for future work.

Acknowledgment: We thank the anonymous reviewers who gave us a wealth of helpful suggestions that have lead to a greatly improved paper. Palsberg and Zhao were supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265.

A Notation

α	path (element of Labels^*)
ϵ	the empty path
Σ	the alphabet of trees
σ	element of Σ
A, B, C	types
$A(\alpha)$	symbol of Σ at path α in type A
$A \downarrow \alpha$	subtree of type A at path α
$A[U := B]$	type A with U replaced by B
a, b, c	terms
$a[\cdot]$	context with one hole
$a[b]$	context $a[\cdot]$ with hole filled by term b
$a[\ell \leftarrow \varsigma(x)b]$	update field ℓ of a with $\varsigma(x)b$
$a[x := b]$	term a with x replaced by b
a^R	term whose typability is equivalent to R
$a \rightsquigarrow b$	term a rewrites to term b
R	relation on types also known as set of constraints
$\mathcal{C}(a)$, E_a, X_a, Y_a	system equivalent to typability of term a
E	type environment
$E[x : A]$	type environment with updated binding for x
$E \backslash x$	type environment with binding for x removed
I	index set
i, j	indices
k, ℓ, m	labels
Labels	the set of labels
n	index bound
Q, W	elements of constraints
S	substitution
U, V	type variables
$\mathsf{Variances}$	the set of variances
v	variance (element of $\mathsf{Variances}$)
X	generic set
x, y	term variables
$\leq$	our subtyping relation
$\sqsubseteq$, $\sqcup$	ordering and lub on $\mathsf{Variances}$

References

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

[2] Alexander Aiken and Edward Wimmers. Type inclusion constraints and type inference. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

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

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA'98, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 183–200, 1998.

[5] Michele Bugliesi and Santiago Pericas-Geertsen. Type inference for variant object types. *Information and Computation*, 177(1):2–27, 2002.

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

[7] 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.

[8] Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997. http://www.cs.williams.edu/ kim/FOOL/index.html.

[9] 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.

[10] Atsushi Igarashi and Naoki Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Information and Computation*, 161(1):1–44, 2000.

[11] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *Proceedings of ECOOP'02, 16th European Conference on Object-Oriented Programming*, 2002.

[12] 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.

[13] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Computation*, 71(1/2):95–130, October/November 1986.

[14] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512 – 537, 2002.

[15] 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.

[16] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

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

[18] Martin Müller, Joachim Niehren, and Andreas Podelski. Ordering constraints over feature trees. *Constraints, an International Journal*, 5(1–2):7–42, January 2000.

[19] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE'89*, pages 357–373, 1989.

[20] 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.

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

[22] 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.

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

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

[25] 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.

[26] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[27] Francois Pottier. Simplifying subtyping constraints. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 122–133, 1996.

[28] Didier Rémy. From classes to objects via subtyping. In *European Symposium On Programming*. Springer-Verlag (*LNCS* 1381), March 1998.

[29] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Technical Report ACRC-99-009, University of South Australia, 1999.

[30] Francis Tang and Martin Hofmann. Type inference for objects with base types. Draft, 2001.

[31] Francis Tang and Martin Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects. In *Proceedings of FOOL'02, Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, January 2002.

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

[33] Mitchell Wand, Patrick M. O'Keefe, and Jens Palsberg. Strong normalization with non-structural subtyping. *Mathematical Structures in Computer Science*, 5(3):419–430, 1995.

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