Information and Computation, 123(2):198–209, 1995.

Efficient Inference of Object Types

Jens Palsberg
palsberg@theory.lcs.mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology
NE43-340
545 Technology Square
Cambridge, MA 02139

Abstract

Abadi and Cardelli have recently investigated a calculus of objects [2]. The calculus supports a key feature of object-oriented languages: an object can be emulated by another object that has more refined methods. Abadi and Cardelli presented four first-order type systems for the calculus. The simplest one is based on finite types and no subtyping, and the most powerful one has both recursive types and subtyping. Open until now is the question of type inference, and in the presence of subtyping "the absence of minimum typings poses practical problems for type inference" [2].

In this paper we give an $O(n^3)$ algorithm for each of the four type inference problems and we prove that all the problems are P-complete. We also indicate how to modify the algorithms to handle functions and records.

1 Introduction

Abadi and Cardelli have recently investigated a calculus of objects [2]. The calculus supports a key feature of object-oriented languages: an object can be emulated by another object that has more refined methods. For example, if the method invocation $a.l$ is meaningful for an object a, then it will also be meaningful for objects with more methods than a, and for objects with more refined methods. This phenomenon is called *subsumption*.

The calculus contains four constructions: variables, objects, method invocation, and method override. It is similar to the calculus of Mitchell, Honsell, and Fisher [20] in allowing method override, but it differs significantly in also allowing subsumption but not allowing objects to be extended with more methods.

	Abadi, Cardelli [2]	Mitchell, Honsell, Fisher [20]
Objects	$\surd$	$\surd$
Method override	$\surd$	$\surd$
Subsumption	$\surd$	
Object extension		$\surd$

Abadi and Cardelli presented four first-order type systems for their calculus. The simplest one is based on finite types and no subtyping, and the most powerful one has both recursive types and subtyping. The latter can type many intriguing object-oriented programs, including objects whose methods return an updated self [2], see also [4, 3, 1].

Open until now is the question of type inference:

> Given an untyped program a, is a typable? If so, annotate it.

In the presence of subtyping "the absence of minimum typings poses practical problems for type inference" [2].

In this paper we give an $O(n^3)$ algorithm for each of the four type inference problems and we prove that all the problems are P-complete.

Choose: finite types or recursive types. Choose: subtyping or no subtyping. In any case: type inference is P-complete and computable in $O(n^3)$.

Our results have practical significance:

1. For object-oriented languages based on method override and subsumption, we provide the core of efficient type inference algorithms.

2. The P-completeness indicates that there are no fast NC-class parallel algorithms for the type inference problems, unless NC = P.

In Section 2 we present Abadi and Cardelli's calculus. For readability, Section 3–6 concentrate on the most powerful of the type systems, the one with recursive types and subtyping. The other type systems requires similar developments that will be summarized in Section 7. We first present the type system (Section 3), and we then prove that the type inference problem is log space equivalent to a constraint problem (Section 4) and a graph problem (Section 5), and we prove that a program is typable if and only if the corresponding graph problem involves a *well-formed* graph (Section 6). If the graph is well-formed, then a certain finite automaton represents a *canonical* typing of the program. In Section 7 we give algorithms for all four type inference problems and in Section 8 we prove that all the problems are P-complete under log space reductions. In Section 9 we give three examples of how the most powerful of our type inference algorithms works. Finally, in Section 10 we discuss related work and possible extensions. The reader is encouraged to refer to the examples while reading the other sections.

Our approach to type inference is related to that of Kozen, Schwartzbach, and the present author in [17]. Although the problems that we solve here are much different from that solved in [17], the two approaches have the same ingredients: constraints, graphs, and automata.

We have produced a prototype implementation in Scheme of the most powerful of our type inference algorithms. Experiments have been carried out on a SPARCserver 1000 (with four SuperSPARC processors) running Scm version 4e1. For example, the implementation used 24 seconds to process a 58 lines program. This is encouraging because we used a rather slow implementation language and because we did not fine-tune the implementation.

A potential obstacle for practical use of our algorithms is the property that the canonical typing of a program may have a representation of a size which is quadratic in the size of the program. Another potential obstacle may be the use of adjacency matrices to represent certain graphs. If those graphs are sparse in practice, then it may be worthwhile using less space-demanding data structures at the cost of slower worst-case performance. Further experiments are needed to evaluate the speed and space-usage of the algorithms on programs of realistic size.

2 Abadi and Cardelli's Calculus

Abadi and Cardelli first present an untyped object calculus, called the ς-calculus. The ς-terms are generated by the following grammar:

$a ::= x$			variable
	$[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$	(l_i distinct)	object
	$a.l$		field selection / method invocation
	$a.l \Leftarrow \varsigma(x)b$		field update / method override

We use a, b, c to range over ς-terms. An object $[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$ has method names l_i and methods $\varsigma(x_i)b_i$. The order of the components does not matter. In a method $\varsigma(x)b$, we have that x is the self variable and b is the body. Thus, in the body of a method we can refer to any enclosing object, like in the Beta language [18].

The reduction rules for ς-terms are as follows. If $o \equiv [l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, then, for $j \in 1..n$,

- $o.l_j \ \rightsquigarrow \ b_j[o/x_j]$
- $o.l_j \Leftarrow \varsigma(y)b \ \rightsquigarrow \ o[l_j \leftarrow \varsigma(y)b]$

Here, $a[o/x]$ denotes the ς-term a with o substituted for free occurrences of x (after renaming bound variables if necessary); and $o[l_j \leftarrow \varsigma(y)b]$ denotes the ς-term o with the l_j field replaced by $\varsigma(y)b$. An evaluation context is an expression with one hole. For an evaluation context $a[.]$, if $b \rightsquigarrow b'$, then $a[b] \rightsquigarrow a[b']$.

A ς-term is said to be an *error* if it is irreducible and it contains either $o.l_j$ or $o.l_j \Leftarrow \varsigma(y)b$, where $o \equiv [l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, and o does *not* contain an l_j field.

For examples of reductions, consider first the object $o \equiv [l = \varsigma(x)x.l]$. The expression $o.l$ yields the infinite computation:

$$o.l \rightsquigarrow x.l[o/x] \equiv o.l \rightsquigarrow \ldots$$

As another example, consider the object $o' \equiv [l = \varsigma(x)x]$. The method l returns self:

$$o'.l \rightsquigarrow x[o'/x] \equiv o'$$

As a final example, consider the object $o'' \equiv [l = \varsigma(y)(y.l \Leftarrow \varsigma(x)x)]$. The method l returns a modified self:

$$o''.l \leadsto (o''.l \Leftarrow \varsigma(x)x) \leadsto o'$$

These three examples are taken from Abadi and Cardelli's paper [2].

Abadi and Cardelli demonstrate how to encode the pure λ-calculus in the ς-calculus. Note the following difference between these two calculi. In pure λ-calculus *no* term yields an error; in the ς-calculus for example $[\ ].l$ yields an error. The reason is that objects are structured values. In a λ-calculus with pairs, some terms yield errors, like in the ς-calculus.

3 Type Rules

The following type system for the ς-calculus catches errors statically, that is, rejects all programs that may yield errors [2].

An object type is an edge-labeled regular tree. A tree is *regular* if it has finitely many distinct subtrees. Labels are drawn from some possibly infinite set $\mathcal{N}$ of method names. We represent a type as a non-empty, prefix-closed set of strings over $\mathcal{N}$. One such string represents a path from the root. We use $A, B, \ldots$ to denote types. The set of all types is denoted T. A type is *finite* if it is finite as a set of strings.

For $l_1, \ldots, l_n \in \mathcal{N}$, $A_1, \ldots, A_n \subseteq \mathcal{N}^*$ and $\alpha \in \mathcal{N}^*$, define

$$\begin{aligned} [l_i : A_i \ {}^{i \in 1..n}] &= \{\epsilon\} \cup \{l_1\alpha \mid \alpha \in A_1\} \cup \ldots \cup \{l_n\alpha \mid \alpha \in A_n\} \\ A \downarrow \alpha &= \{\beta \mid \alpha\beta \in A\} . \end{aligned}$$

Here, $[l_i : A_i \ {}^{i \in 1..n}]$ is an object type with components $l_i : A_i$, and $A \downarrow \alpha$ is the subtree of A at α if $\alpha \in A$, $\emptyset$ if not. The following properties are immediate from the definitions:

(i) $[l_i : A_i \ {}^{i \in 1..n}] \downarrow l_i = A_i$

(ii) $(A \downarrow \alpha) \downarrow \beta = A \downarrow \alpha\beta$

The set of object types is ordered by the subtyping relation $\leq$ as follows:

$$A \leq B \quad \text{if and only if} \quad \forall l \in \mathcal{N} : \quad l \in B \Rightarrow (l \in A \ \wedge \ A \downarrow l = B \downarrow l)$$

Clearly, $\leq$ is a partial order. Intuitively, if $A \leq B$, then A may contain more fields than B, and for common fields, A and B must have the same type. For example, $[l : A, m : B] \leq [l : A]$, but $[l : [m : A]] \not\leq [l : [\,]]$. Notice that if $A \leq B$, then $B \subseteq A$.

To state typing rules, Abadi and Cardelli use an explicitly typed version of the ς-calculus where each bound variable is annotated with a type.

If a is an explicitly typed ς-term, A is an object type, and E is a type environment, that is, a partial function assigning types to variables, then the judgment $E \vdash a : A$ means that a has the type A in the environment E. This holds when the judgment is derivable using the following five rules:

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

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

$$\forall j \in 1..n : \quad \frac{E \vdash a : [l_i : B_i \;^{i \in 1..n}]}{E \vdash a.l_j : B_j} \tag{3}$$

$$\forall j \in 1..n : \quad \frac{E \vdash a : A \quad E[x \leftarrow A] \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x : A)b : A} \quad \text{(where } A = [l_i : B_i \;^{i \in 1..n}]\text{)} \tag{4}$$

$$\frac{E \vdash a : A \quad A \leq B}{E \vdash a : B} \tag{5}$$

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.

Notice that rule (3) can be replaced by the equivalent rule

$$\frac{E \vdash a : [l_j : B_j]}{E \vdash a.l_j : B_j} \tag{6}$$

Notice also that rule (4) can be replaced by the equivalent rule

$$\frac{E \vdash a : A \quad E[x \leftarrow A] \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x : A)b : A} \quad \text{(where } A \leq [l_j : B_j]\text{)} \tag{7}$$

If $E \vdash a : A$ is derivable, we say that a is *well-typed* with type A. An untyped ς-term a is said to be *typable* if there exists an annotated version of a which is well-typed.

For comparison with the typing rules for simply typed λ-calculus, notice that:

- Rule (1) is identical to the rule for variables in λ-calculus;
- Rule (2) can be understood as the rule for object type introduction, just like the rule for λ-abstraction is the rule for function type introduction; and
- Rule (3) can be understood as the rule for object type elimination, just like the rule for application is the rule for function type elimination in λ-calculus.

Rule (4) has no obvious counterpart among the typing rules for simply typed λ-calculus.

For examples of type derivations, let us consider the three example terms from Section 2. First consider the object $o \equiv [l = \varsigma(x)x.l]$. The expression $o.l$ can be typed as follows, with x implicitly typed with $[l : [\,]]$:

$$\cfrac{\cfrac{\cfrac{\emptyset[x \leftarrow [l : [\,]]] \vdash x : [l : [\,]]}{\emptyset[x \leftarrow [l : [\,]]] \vdash x.l : [\,]}}{\emptyset \vdash o : [l : [\,]]}}{\emptyset \vdash o.l : [\,]\ .}$$

Consider then the object $o' \equiv [l = \varsigma(x)x]$. The expression $o'.l$ can be typed as follows, with x implicitly typed with $[l : [\,]]$:

$$\cfrac{\cfrac{\cfrac{\emptyset[x \leftarrow [l : [\,]]] \vdash x : [l : [\,]] \quad [l : [\,]] \leq [\,]}{\emptyset[x \leftarrow [l : [\,]]] \vdash x : [\,]}}{\emptyset \vdash o' : [l : [\,]]}}{\emptyset \vdash o'.l : [\,]\ .}$$

Consider then the object $o'' \equiv [l = \varsigma(y)b]$, where $b \equiv y.l \Leftarrow \varsigma(x)x$. The expression $o''.l$ can be typed as follows, with both x and y implicitly typed with $[l : [\,]]$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\emptyset[y \leftarrow [l : [\,]]] \vdash y : [l : [\,]] \quad \emptyset[y \leftarrow [l : [\,]], x \leftarrow [l : [\,]]] \vdash x : [\,]}{\emptyset[y \leftarrow [l : [\,]]] \vdash b : [l : [\,]] \quad [l : [\,]] \leq [\,]}}{\emptyset[y \leftarrow [l : [\,]]] \vdash b : [\,]}}{\emptyset \vdash o'' : [l : [\,]]}}{\emptyset \vdash o''.l : [\,]\ .}$$

Consider finally the object $[\,]$. Trying to type the expression $[\,].l$ will fail because rule (2) can only give $[\,]$ the type $[\,]$, so rule (3) cannot be applied afterwards.

4 From Rules to Constraints

In this section we prove that the type inference problem is log space equivalent to solving a finite system of type constraints. The constraints isolate the essential combinatorial structure of the type inference problem.

Definition 4.1 Given a denumerable set of variables, an *AC-system* (Abadi/Cardelli-system) is a finite set of inequalities $W \leq W'$, where W and W' are of the forms V or $[l_i : V_i \;^{i \in 1..n}]$, and where $V, V_1, \ldots, V_n$ are variables. If L maps variables to types, then define $\widetilde{L}$ as follows:

$$\widetilde{L}(W) = \begin{cases} [l_1 : L(V_1), \ldots, l_n : L(V_n)] & \text{if } W \text{ is of the form } [l_i : V_i \;^{i \in 1..n}] \\ L(V) & \text{if } W \text{ is a variable } V \end{cases}$$

A *solution* for an AC-system is a map L from variables to types such that for all $W \leq W'$ in the AC-system, $\widetilde{L}(W) \leq \widetilde{L}(W')$. $\square$

For examples of AC-systems, see Section 9.

We first prove that the type inference problem is log space reducible to solving AC-systems.

Given an untyped ς-term c, assume that it has been α-converted so that all bound variables are distinct. We will now generate an AC-system from c where the bound variables of c are a subset of the variables in the constraint system. This will be convenient in the statement and proof of Lemma 4.2 below. Let X be the set of bound variables in c; let Y be a set of variables disjoint from X consisting of one variable $[\![b]\!]$ for each occurrence of a subterm b of c; and let Z be a set of variables disjoint from X and Y consisting of one variable $\langle a.l_j \rangle$ for each occurrence of a subterm $a.l_j$ of c. (The notations $[\![b]\!]$ and $\langle a.l_j \rangle$ are ambiguous because there may be more than one occurrence of the term b or $a.l_j$ in c. However, it will always be clear from context which occurrence is meant.) Notice that there are two variables $\langle a.l_j \rangle$ and $[\![a.l_j]\!]$ for each occurrences of a subterm $a.l_j$ of c. Intuitively, $\langle a.l_j \rangle$ denotes the type of $a.l_j$ *before* subtyping, and $[\![a.l_j]\!]$ denotes the type of $a.l_j$ *after* subtyping. We generate the following AC-system of inequalities over $X \cup Y \cup Z$:

- for every occurrence in c of a bound variable x, the inequality

$$x \leq [\![x]\!] \tag{8}$$

- for every occurrence in c of a subterm of the form $[l_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]$, the inequality

$$[l_i : [\![b_i]\!]\ ^{i \in 1..n}] \leq [\![[l_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]]\!] \tag{9}$$

  and for every $i \in 1..n$, the equalities

$$x_i = [l_i : [\![b_i]\!]\ ^{i \in 1..n}] \tag{10}$$

- for every occurrence in c of a subterm of the form $a.l_j$, the inequalities

$$[\![a]\!] \leq [l_j : \langle a.l_j \rangle] \tag{11}$$

$$\langle a.l_j \rangle \leq [\![a.l_j]\!] \tag{12}$$

- for every occurrence in c of a subterm of the form $a.l_j \Leftarrow \varsigma(x)b$, the constraints

$$[\![a]\!] \leq [\![a.l_j \Leftarrow \varsigma(x)b]\!] \tag{13}$$

$$[\![a]\!] = x \tag{14}$$

$$[\![a]\!] \leq [l_j : [\![b]\!]] \tag{15}$$

In (8) to (15), each equality $A = B$ denotes the two inequalities $A \leq B$ and $B \leq A$.

Denote by $C(c)$ the AC-system of constraints generated from c in this fashion. For a ς-term c of size n, the AC-system $C(c)$ is of size $O(n)$, and it is generated using $O(\log n)$ space. We show below that the solutions of $C(c)$ over T correspond to the possible type annotations of c in a sense made precise by Lemma 4.2. For examples of AC-systems generated from ς-terms, see Section 9.

The constraints are motivated by the forms of the corresponding type rules. The reason for the use of $\leq$ in four of the constraint rules can be summarized as follows:

- Given a type derivation, we can find a particular type derivation where the subsumption rule (5) is used exactly once for each occurrence of a subterm.

This explains the use of $\leq$ in the constraints (8), (9), (12), (13). For example, consider an occurrence in c of a variable x. If the constraint $x \leq [\![x]\!]$ has solution L, then we can construct the type derivation

$$\frac{L \vdash x : L(x) \quad L(x) \leq L([\![x]\!])}{L \vdash x : L([\![x]\!])} \; .$$

The use of $\leq$ in the constraints (11) and (15) is motivated by the rules (6) and (7).

Notice that we *cannot* replace the constraints (11) and (12) by the single constraint

$$[\![a]\!] \leq [l_j : [\![a.l_j]\!]] \tag{16}$$

To see this, let $a.l_j$ occur in c and suppose $C(a.l_j)$ has solution L (so in particular, (11) and (12) has solution L). Consider the type derivation

$$\frac{\dfrac{L \vdash a : L([\![a]\!]) \quad L([\![a]\!]) \leq [l_j : L(\langle a.l_j \rangle)]}{L \vdash a : [l_j : L(\langle a.l_j \rangle)]}}{\dfrac{L \vdash a.l_j : L(\langle a.l_j \rangle) \quad L(\langle a.l_j \rangle) \leq L([\![a.l_j]\!])}{L \vdash a.l_j : L([\![a.l_j]\!])}}$$

Clearly, $L([\![a]\!]) \downarrow l_j$ need *not* be equal to $L([\![a.l_j]\!])$. With the constraint (16), however, they are forced to be equal.

Let E be a type environment assigning a type to each variable occurring freely in c. If L is a function assigning a type to each variable in $X \cup Y \cup Z$, we say that L *extends* E if E and L agree on the domain of E.

If b is an annotated ς-term, then we let $\overline{b}$ denote the corresponding untyped term. Moreover, we let $\widehat{b}$ be the partial function that maps each bound variable in b to its type annotation.

Lemma 4.2 *The judgment $E \vdash c : A$ is derivable if and only if there exists a solution L of $C(\overline{c})$ extending both E and $\widehat{c}$, such that $L([\![\overline{c}]\!]) = A$. In particular, if c is closed, then c is well-typed with type A if and only if there exists a solution L of $C(\overline{c})$ extending $\widehat{c}$ such that $L([\![\overline{c}]\!]) = A$.*

Proof. We first prove that if $C(\overline{c})$ has a solution L extending both E and $\widehat{c}$, then $L \vdash c : L([\![\overline{c}]\!])$ is derivable. We proceed by induction on the structure of c.

For the base case, $L \vdash x : L(\llbracket x \rrbracket)$ is derivable using rules (1) and (5), since $L(x) \leq L(\llbracket x \rrbracket)$.

For the induction step, consider first $[l_i = \varsigma(x_i)\overline{b_i}\ ^{i \in 1..n}]$. Let $A = [l_i : L(\llbracket \overline{b_i} \rrbracket)\ ^{i \in 1..n}]$. To derive $L \vdash [l_i = \varsigma(x_i : L(x_i))b_i\ ^{i \in 1..n}] : L(\llbracket [l_i = \varsigma(x_i)\overline{b_i}\ ^{i \in 1..n}] \rrbracket)$, by rule (5) and the fact that $A \leq L(\llbracket [l_i = \varsigma(x_i)\overline{b_i}\ ^{i \in 1..n}] \rrbracket)$, it suffices to derive $L \vdash [l_i = \varsigma(x_i : L(x_i))b_i\ ^{i \in 1..n}] : A$. From the fact that $L(x_i) = A$ for every $i \in 1..n$, it suffices to derive $L \vdash [l_i = \varsigma(x_i : A)b_i\ ^{i \in 1..n}] : A$. The side condition of rule (2) is clearly satisfied, so it suffices to derive, for each $i \in 1..n$, $L[x_i \leftarrow A] \vdash b_i : L(\llbracket \overline{b_i} \rrbracket)$, or in other words, $L \vdash b_i : L(\llbracket \overline{b_i} \rrbracket)$. But since L is a solution of $C([l_i = \varsigma(x_i)\overline{b_i}\ ^{i \in 1..n}])$, it is also a solution of $C(\overline{b_i})$ for each $i \in 1..n$, thus the desired derivations are provided by the induction hypothesis.

Now consider $a.l_j$. Since L is a solution of $C(\overline{a}.l_j)$, it is also a solution of $C(\overline{a})$. From the induction hypothesis, we obtain a derivation of $L \vdash a : L(\llbracket \overline{a} \rrbracket)$. By rule (3) and the fact that $L(\llbracket \overline{a} \rrbracket) \leq [l_j : L(\langle \overline{a}.l_j \rangle)]$, we obtain a derivation of $L \vdash a.l_j : L(\langle \overline{a}.l_j \rangle)$. Using rule (5) and the fact that $L(\langle \overline{a}.l_j \rangle) \leq L(\llbracket \overline{a}.l_j \rrbracket)$, we then obtain a derivation of $L \vdash a.l_j : L(\llbracket \overline{a}.l_j \rrbracket)$.

Finally consider $a.l_j \Leftarrow \varsigma(x : L(x))b$. Let $A = L(\llbracket \overline{a} \rrbracket)$. To derive $L \vdash a.l_j \Leftarrow \varsigma(x : L(x))b : L(\llbracket \overline{a}.l_j \Leftarrow \varsigma(x)\overline{b} \rrbracket)$, by rule (5) and the fact that $A \leq L(\llbracket \overline{a}.l_j \Leftarrow \varsigma(x)\overline{b} \rrbracket)$, it suffices to derive $L \vdash a.l_j \Leftarrow \varsigma(x : L(x))b : A$. From the fact that $A = L(x)$, it suffices to derive $L \vdash a.l_j \Leftarrow \varsigma(x : A)b : A$. The side condition of rule (7) is satisfied because $A \leq [l_j : L(\llbracket \overline{b} \rrbracket)]$, so it suffices to derive $L \vdash a : A$ and $L[x \leftarrow A] \vdash b : L(\llbracket \overline{b} \rrbracket)$, or in other words $L \vdash a : L(\llbracket \overline{a} \rrbracket)$ and $L \vdash b : L(\llbracket \overline{b} \rrbracket)$. But since L is a solution of $C(\overline{a}.l_j \Leftarrow \varsigma(x)\overline{b})$, it is also a solution of $C(\overline{a})$ and $C(\overline{b})$, thus the desired derivations are provided by the induction hypothesis.

We then prove that if $E \vdash c : A$ is derivable, then there exists a solution L of $C(\overline{c})$ extending both E and $\widehat{c}$.

Suppose $E \vdash c : A$ is derivable, and consider a derivation of minimal length. Since the derivation is minimal, there is exactly one application of the rule (1) involving a particular occurrence of a variable x, exactly one application of the rule (2) involving a particular occurrence of a subterm $[l_i = \varsigma(x_i : A)b_i\ ^{i \in 1..n}]$, exactly one application of the rule (3) involving a particular occurrence of a subterm $a.l_j$, and exactly one application of the rule (4) involving a particular occurrence of a subterm $a.l_j \Leftarrow \varsigma(x : A)b$. In the case of a variable x, there is a unique type B such that $F(x) = B$ for any F such that a judgment $F \vdash a : B'$ appears in the derivation for some occurrence of a subterm a of $\varsigma(x : A)b$; this can be proved by induction on the structure of the derivation of $F \vdash a : B'$. Finally, there can be at most one application of the rule (5) involving a particular occurrence of any subterm; if there were more than one, they could be combined using the transitivity of $\leq$ to give a shorter derivation.

Now construct L as follows. For every free variable x of c define $L(x) = E(x)$. For every bound variable x, let $\varsigma(x : A)b$ be the method in which it is bound, and define $L(x) = A$. For every occurrence of a subterm a of c, find the last judgment in the derivation of the form $F \vdash a : B$ involving that occurrence of a, and define $L([\![\overline{a}]\!]) = B$. Intuitively, the *last* judgment of the form $F \vdash a : B$ means the judgment *after* the use of subsumption. Finally, for every occurrence of a subterm $a.l_j$ of c, find the unique application of the rule (3) deriving $F \vdash a.l_j : B_j$, and define $L(\langle a.l_j \rangle) = B_j$.

Certainly L extends E and $\widehat{c}$ and $L([\![\overline{c}]\!]) = A$. We now show that L is a solution of $C(\overline{c})$.

For an occurrence of a bound variable x, there are two cases. Suppose first that the variable is bound in a method that occurs in an object declaration. Find the unique application of the rule (2) deriving the judgment $F \vdash [l_i = \varsigma(x_i : A)b_i \; ^{i \in 1..n}] : A$ from a family of premises where one of them is $F[x \leftarrow A] \vdash b : B_i$. Then $L(x) = A$. The rule (1) must have been applied to obtain a judgment of the form $G \vdash x : L(x)$ and only rule (5) applied to that occurrence of x thereafter, thus $L(x) \leq L([\![x]\!])$. Suppose then that the variable is bound in a method that occurs in a method override. Find the unique application of the rule (4) deriving the judgment $F \vdash a.l_j \Leftarrow \varsigma(x : A)b : A$ from two premises where one of them is $F[x \leftarrow A] \vdash b : B_j$. As before, we get that $L(x) \leq L([\![x]\!])$.

For an occurrence of a subterm of the form $[l_i = \varsigma(x_i : A)b_i \; ^{i \in 1..n}]$, find the unique application of the rule (2) deriving the judgment $F \vdash [l_i = \varsigma(x_i : A)b_i \; ^{i \in 1..n}] : A$ from the premises $F[x_i \leftarrow A] \vdash b_i : B_i$, where $A = [l_i : L([\![\overline{b_i}]\!]) \; ^{i \in 1..n}]$. Then $L(x_i) = A$, and $A \leq L([\![[l_i = \varsigma(x_i)\overline{b_i} \; ^{i \in 1..n}]]\!])$.

For an occurrence of a subterm of the form $a.l_j$, find the unique application of the rule (3) deriving the judgment $F \vdash a.l_j : B_j$ from the premise $F \vdash a : [l_i : B_i \; ^{i \in 1..n}]$. Then $L([\![\overline{a}]\!]) = [l_i : B_i \; ^{i \in 1..n}]$ and $B_j = L(\langle \overline{a}.l_j \rangle) \leq L([\![\overline{a}.l_j]\!])$. Thus, $L([\![\overline{a}]\!]) \leq [l_j : L(\langle \overline{a}.l_j \rangle)]$, by the definition of $\leq$.

Finally for an occurrence of a subterm of the form $a.l_j \Leftarrow \varsigma(x:A)b$, find the unique application of the rule (4) deriving the judgment $F \vdash a.l_j \Leftarrow \varsigma(x:A)b : A$ from the premises $F \vdash a : A$ and $F[x \leftarrow A] \vdash b : B_j$, where $A = [l_i : B_i \ ^{i \in 1..n}]$. Then $L([\![\overline{a}]\!]) = A \leq L([\![\overline{a}.l_j \Leftarrow \varsigma(x)\overline{b}]\!])$, and $A = L(x)$. Moreover, $L([\![\overline{b}]\!]) = B_j$, so $L([\![\overline{a}]\!]) \leq [l_j : L([\![\overline{b}]\!])]$, by the definition of $\leq$. □

We then prove that solving AC-systems is log space reducible to the type inference problem.

Lemma 4.3 *Solvability of AC-systems is logspace-reducible to the type inference problem.*

Proof. Let C be an AC-system. Recall that for an inequality $W \leq W'$ in C, both W and W' are of the forms V or $[l_i : V_i \ ^{i \in 1..n}]$, where $V, V_1, \ldots, V_n$ are variables. Define

$$
\begin{array}{lllll}
a^C = & [& l_V & = & \varsigma(x)(x.l_V) \\
& & & & \quad \text{for each variable } V \text{ in } C \\
& & l_R & = & \varsigma(x)[l_i = \varsigma(y)(x.l_{V_i}) \ ^{i \in 1..n}] \\
& & & & \quad \text{for each } R \text{ in } C \text{ of the form } [l_i : V_i \ ^{i \in 1..n}] \\
& & m_{R,l_j} & = & \varsigma(x)((x.l_{V_j} \Leftarrow \varsigma(y)(x.l_R.l_j)).l_R) \\
& & & & \quad \text{for each } R \text{ in } C \text{ of the form } [l_i : V_i \ ^{i \in 1..n}] \\
& & & & \quad \text{and for each } j \in 1..n \\
& & l_{W \leq W'} & = & \varsigma(x)((x.l_{W'} \Leftarrow \varsigma(y)(x.l_W)).l_W) \\
& & & & \quad \text{for each inequality } W \leq W' \text{ in } C \\
&] & & &
\end{array}
$$

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

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

$$
\begin{array}{lllll}
A = & [& l_V & : L(V) & \text{for each variable } V \text{ in } C \\
& & l_R & : \tilde{L}(R) & \text{for each } R \text{ in } C \text{ of the form } [l_i : V_i \ ^{i \in 1..n}] \\
& & m_{R,l_j} & : \tilde{L}(R) & \text{for each } R \text{ in } C \text{ of the form } [l_i : V_i \ ^{i \in 1..n}] \\
& & & & \text{and for each } j \in 1..n \\
& & l_{W \leq W'} & : \tilde{L}(W) & \text{for each inequality } W \leq W' \text{ in } C \\
&] & & &
\end{array}
$$

Define also

$$
\begin{array}{llll}
d = & [& l_V & = \varsigma(x:A)(x.l_V) \\
& & & \quad\text{for each variable } V \text{ in } C \\
& & l_R & = \varsigma(x:A)[l_i = \varsigma(y:\widetilde{L}(R))(x.l_{V_i})\ ^{i\in 1..n}] \\
& & & \quad\text{for each } R \text{ in } C \text{ of the form } [l_i : V_i\ ^{i\in 1..n}] \\
& & m_{R,l_j} & = \varsigma(x:A)((x.l_{V_j} \Leftarrow \varsigma(y:A)(x.l_R.l_j)).l_R) \\
& & & \quad\text{for each } R \text{ in } C \text{ of the form } [l_i : V_i\ ^{i\in 1..n}] \\
& & & \quad\text{and for each } j \in 1..n \\
& & l_{W\le W'} & = \varsigma(x:A)((x.l_{W'} \Leftarrow \varsigma(y:A)(x.l_W)).l_W) \\
& & & \quad\text{for each inequality } W \le W' \text{ in } C \\
&] & &
\end{array}
$$

Clearly, d is an annotated version of a^C and $\emptyset \vdash d : A$ is derivable.

We then prove that if a^C is typable, then C is solvable. Suppose a^C is typable. From Lemma 4.2 we get a solution M of $C(a^C)$. Notice that each method in a^C binds a variable x. Each of these variables corresponds to a distinct type variable in $C(a^C)$. Since M is a solution of $C(a^C)$, and $C(a^C)$ contains constraints of the form $x = [\ldots]$ for each method in a^C (from rule (10)), all those type variables are mapped by M to the same type. Thus, we can think of all the bound variables in a^C as being related to the same type variable, which we will write as x.

Define

$$
L(V) \quad = \quad M(x)\downarrow l_V \qquad \text{for each variable } V \text{ in } C.
$$

The definition is justified by Property 1 below.

- **Property 1** If V is a variable in C, then $M(x)\downarrow l_V$ is defined.
- **Property 2** For each R in C of the form $[l_i : V_i\ ^{i\in 1..n}]$, we have $M(x)\downarrow l_R = [l_i : (M(x)\downarrow l_{V_i})\ ^{i\in 1..n}]$.

We will proceed by first showing the two properties and then showing that C has solution L.

To see Property 1, notice that in the body of the method l_V we have the expression $x.l_V$. Since M is a solution of $C(a^C)$, we have from the rules (8) and (11) that M satisfies

$$
x \le [\![x]\!] \le [l_V : \langle x.l_V \rangle] \ .
$$

We conclude that $M(x)\downarrow l_V = M(\langle x.l_V \rangle)$ is defined.

To see Property 2, let R be an occurrence in C of the form $[l_i : V_i \;\; ^{i\in 1..n}]$. In the body of the method l_R have the expression $[l_i = \varsigma(y)(x.l_{V_i}) \;\; ^{i\in 1..n}]$. Since M is a solution of $C(a^C)$, we have from the rules (10), (9), (8), (11), and (12) that M satisfies

$$\begin{array}{rcll} x & = & [\ldots l_R : [\![[l_i = \varsigma(y)(x.l_{V_i}) \;\; ^{i\in 1..n}]]\!] \ldots] & (17) \\ [l_i : [\![x.l_{V_i}]\!] \;\; ^{i\in 1..n}] & \leq & [\![[l_i = \varsigma(y)(x.l_{V_i}) \;\; ^{i\in 1..n}]]\!] & (18) \\ x \leq [\![x]\!] & \leq & [l_{V_i} : \langle x.l_{V_i}\rangle \;\; ^{i\in 1..n}] \quad \text{for each } i \in 1..n & (19) \\ \langle x.l_{V_i}\rangle & \leq & [\![x.l_{V_i}]\!] \quad \text{for each } i \in 1..n & (20) \end{array}$$

Thus, for each $i \in 1..n$,

$$\begin{array}{rcll} M(x)\downarrow l_{V_i} & = & M(\langle x.l_{V_i}\rangle) & \text{from (19)} \\ & \leq & M([\![x.l_{V_i}]\!]) & \text{from (20)} \\ & = & M([\![[l_i = \varsigma(y)(x.l_{V_i}) \;\; ^{i\in 1..n}]]\!])\downarrow l_i & \text{from (18)} \\ & = & M(x)\downarrow l_R \downarrow l_i & \text{from (17)} \end{array}$$

For each $j \in 1..n$, in the body of the method m_{R,l_j}, we have the expression $x'.l_{V_j} \Leftarrow \varsigma(y)(x.l_R.l_j)$ where we, for clarity, have written the first occurrence of x as x'. Since M is a solution of $C(a^C)$, we have from the rules (8), (15), (8), (11), (12), (11), and (12) that M satisfies

$$\begin{array}{rcll} x \leq [\![x']\!] & \leq & [l_{V_j} : [\![x.l_R.l_j]\!]] & (21) \\ x \leq [\![x]\!] & \leq & [l_R : \langle x.l_R\rangle] & (22) \\ \langle x.l_R\rangle & \leq & [\![x.l_R]\!] & (23) \\ [\![x.l_R]\!] & \leq & [l_j : \langle x.l_R.l_j\rangle] & (24) \\ \langle x.l_R.l_j\rangle & \leq & [\![x.l_R.l_j]\!] & (25) \end{array}$$

Thus,

$$\begin{array}{rcll} M(x)\downarrow l_R \downarrow l_j & = & M(\langle x.l_R\rangle)\downarrow l_j & \text{from (22)} \\ & = & M([\![x.l_R]\!])\downarrow l_j & \text{from (23)} \\ & = & M(\langle x.l_R.l_j\rangle) & \text{from (24)} \\ & \leq & M([\![x.l_R.l_j]\!]) & \text{from (25)} \\ & = & M(x)\downarrow l_{V_j} & \text{from (21)} \end{array}$$

We conclude that for each $i \in 1..n$, $M(x)\downarrow l_R \downarrow l_i = M(x)\downarrow l_{V_i}$. From this, (17), and (18) we get Property 2.

We can summarize Property 1 and 2 as follows.

- **Property 3** If W is lefthand-side or a righthand-side of an inequality in C, then $M(x)\downarrow l_W$ is defined and $\widetilde{L}(W) = M(x)\downarrow l_W$.

We will now show that C has solution L. Consider an inequality $W \leq W'$ in C. The body of the method $l_{W\leq W'}$ contains the expression $x'.l_{W'} \Leftarrow \varsigma(y)(x.l_W)$ where we, for clarity, have written the first occurrence of x as x'. Since M is a solution of $C(a^C)$, we have from the rules (8), (15), (8), (11), and (12) that M satisfies

$$\begin{aligned} x \leq [\![x']\!] &\leq [l_{W'} : [\![x.l_W]\!]] && (26) \\ x \leq [\![x]\!] &\leq [l_W : \langle x.l_W \rangle] && (27) \\ \langle x.l_W \rangle &\leq [\![x.l_W]\!] && (28) \end{aligned}$$

We conclude

$$\begin{aligned} \widetilde{L}(W) &= M(x)\downarrow l_W && \text{from Property 3} \\ &= M(\langle x.l_W \rangle) && \text{from (27)} \\ &\leq M([\![x.l_W]\!]) && \text{from (28)} \\ &= M(x)\downarrow l_{W'} && \text{from (26)} \\ &= \widetilde{L}(W') && \text{from Property 3} \end{aligned}$$

□

For example, let C be the AC-system consisting of the single constraint

$$V \leq [l : W]$$

We then get:

$$\begin{aligned} a^C = [\ & l_V = \varsigma(x)(x.l_V) \\ & l_W = \varsigma(x)(x.l_W) \\ & l_{[l:W]} = \varsigma(x)[l = \varsigma(y)(x.l_W)] \\ & m_{[l:W],l} = \varsigma(x)((x.l \Leftarrow \varsigma(y)(x.l_{[l:W]}.l)).l_{[l:W]}) \\ & l_{V\leq[l:W]} = \varsigma(x)((x.l_{[l:W]} \Leftarrow \varsigma(y)(x.l_V)).l_V) \\ ] & \end{aligned}$$

By combining Lemmas 4.2 and 4.3 we get the following result.

Theorem 4.4 *The type inference problem is log space equivalent to solving AC-systems.*

5 From Constraints to Graphs

In this section we prove that solving AC-systems is log space equivalent to solving a certain kind of constraint graphs. The graphs yield a convenient setting for applying algorithms like transitive closure.

Definition 5.1 A *AC-graph* is a directed graph $G = (N, S, L, \leq)$ consisting of two disjoint sets of nodes N and S, and two disjoint sets of directed edges L and $\leq$. Each edge in L is labeled with some $l \in \mathcal{N}$, and each edge in $\leq$ is labeled with $\leq$. An AC-graph satisfies the properties:

- any N node has finitely many outgoing L edges, all to S nodes, and those edges have distinct labels;
- any S node has *no* outgoing L edges;

For each $h : S \to T$, define $\widetilde{h} : (N \cup S) \to T$ as

$$\widetilde{h}(u) = \begin{cases} [l_1 : h(v_1), \ldots, l_n : h(v_n)] & \text{if } u \xrightarrow{l_i} v_i \text{ are the } L \text{ edges from } u \in N \\ h(u) & \text{if } u \in S \end{cases}$$

A *solution* for G is any map $h : S \to T$ such that if $u \xrightarrow{\leq} v$, then $\widetilde{h}(u) \leq \widetilde{h}(v)$. The solution h is *finite* if $h(s)$ is a finite set for all s. □

For examples of AC-graphs, see Section 9.

Recall that we represent a type as a set of strings. Note that a solvable AC-graph has a pointwise $\subseteq$-least solution. To see this, observe that the intersection of any non-empty family of solutions is itself a solution. Thus, provided that solutions exist, the intersection of all solutions is the pointwise $\subseteq$-least solution.

Theorem 5.2 *Solving AC-systems is log space equivalent to solving AC-graphs.*

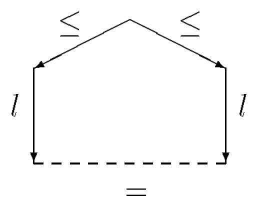
Proof. Given an AC-system, we construct an AC-graph as follows. Associate a unique N node with every subexpression of the form $[l_i : V_i \ ^{i \in 1..n}]$, and associate a unique S node with every variable. From the node for $[l_i : V_i \ ^{i \in 1..n}]$, define for each $i \in 1..n$ an L edge labeled l_i to the node for V_i. Finally, define $\leq$ edges for the inequalities. Notice that the AC-graph can be generated in log space. Clearly, the resulting AC-graph is solvable if and only if the AC-system is solvable.

Conversely, given an AC-graph, we construct an AC-system as follows. For every $\leq$ edge, generate the obvious inequality. Notice that the AC-system can be generated in log space. Clearly, the resulting AC-system is solvable if and only if the AC-graph is solvable. □

For examples of AC-graphs generated from AC-systems, see Section 9.

The following two definitions provide the basis for the algorithms for solving AC-graphs that will be presented in Section 7. First we define the *closure* of an AC-graph, which makes the solutions (or lack thereof) explicit. Then we define a condition for solvability: well-formedness of the graph. The concepts of closure and well-formedness are brought together in Theorem 6.5 which says that a closed graph is solvable if and only if it is well-formed.

Definition 5.3 An AC-graph is *closed* if the edge relation $\leq$ is reflexive, transitive, and closed under the following rule which say that the dashed edge exists whenever the solid ones do:



The $=$ edge denotes two $\leq$ edges, one in each direction.

The *closure* of an AC-graph G is the smallest closed AC-graph containing G as a subgraph. □

Note that an AC-graph and its closure have the same set of solutions. To see this, observe that any solution of the closure of G is also a solution of G, since G has fewer constraints. Conversely, observe that the closure of G can be constructed from G by iterating the closure rules, and it follows inductively by the definition of $\leq$ that any solution of G satisfies the additional constraints added by this process.

Definition 5.4 An AC-graph is *well-formed* if for any nodes $u, v \in N$ with $u \xrightarrow{\leq} v$, if v has an outgoing edge labeled l, then so does u. □

6 From Graphs to Automata

In this section we define an automaton $\mathcal{M}$ and prove our main result. This automaton will be used to characterize the canonical assignment of types to nodes of a given AC-graph.

Definition 6.1 Let an AC-graph $G = (N, S, L, \leq)$ be given. The automaton $\mathcal{M}$ is defined as follows. The input alphabet of $\mathcal{M}$ is $\mathcal{N}$ (the set of method names). The states of $\mathcal{M}$ are $N \cup S$. We use p, q, s, u, v, w to range over states. The transitions are defined as follows.

$$\begin{array}{ll} v \xrightarrow{\epsilon} v' & \text{if } v \xrightarrow{\leq} v' \text{ in } G \\ v \xrightarrow{l} v' & \text{if } v \xrightarrow{l} v' \text{ in } G \end{array}$$

If p and q are states of $\mathcal{M}$ and $\alpha \in \mathcal{N}^*$, we write $p \xrightarrow{\alpha} q$ if the automaton can move from state p to state q under input α, including possible ϵ-transitions.

The automaton $\mathcal{M}_s$ is the automaton $\mathcal{M}$ with start state s. All states are accept states; thus the language accepted by $\mathcal{M}_s$ is the set of strings α for which there exists a state p such that $s \xrightarrow{\alpha} p$. We denote this language by $\mathcal{L}(s)$.

The set $\mathcal{L}(s)$ is clearly nonempty, since $s \xrightarrow{\epsilon} s$ is in $\mathcal{M}$, and it is prefix-closed since all states in $\mathcal{M}$ are accept states. Moreover, $\mathcal{L}(s)$ is regular: each of the finitely many states in $\mathcal{M}$ corresponds to a subtree in $\mathcal{L}(s)$. Hence, $\mathcal{L}(s) \in T$. $\Box$

The intuition motivating the definition of $\mathcal{M}$ is that we want to identify the conditions that require a path to exist in any solution. Thus $\mathcal{L}(s)$ is the set of α that *must* be there; this intuition is made manifest in the following lemma.

Recall from Section 3 that $A \downarrow \alpha$ is the subtree of A at α if $\alpha \in A$, $\emptyset$ if not.

Lemma 6.2 *Let G be an AC-graph. If $h : S \to T$ is any solution and $s \xrightarrow{\alpha} p$ in $\mathcal{M}$, then $\alpha \in \widetilde{h}(s)$ and $\widetilde{h}(s) \downarrow \alpha \leq \widetilde{h}(p)$.*

Proof. We proceed by induction on the number of transitions. If this is zero, then $p = s$ and $\alpha = \epsilon$, and the result is immediate. Otherwise, assume that $s \xrightarrow{\alpha} p$ in $\mathcal{M}$ and the lemma holds for this sequence of transitions. We argue by cases, depending on the form of the next transition out of p.

Consider first a transition $p \xrightarrow{\epsilon} q$. Then $p \overset{\leq}{\Rightarrow} q$ in G, so $\widetilde{h}(p) \leq \widetilde{h}(q)$. By the induction hypothesis, we get $\alpha\epsilon = \alpha \in \widetilde{h}(s)$ and

$$\widetilde{h}(s)\downarrow\alpha\epsilon = \widetilde{h}(s)\downarrow\alpha \leq \widetilde{h}(p) \leq \widetilde{h}(q).$$

Consider then a transition $p \xrightarrow{l} q$. Then $p \xrightarrow{l} q$ in G, so $\widetilde{h}(p)\downarrow l = \widetilde{h}(q)$. Thus, $l \in \widetilde{h}(p)$. By the induction hypothesis we get $\widetilde{h}(p) \subseteq \widetilde{h}(s)\downarrow\alpha$, so $l \in \widetilde{h}(s)\downarrow\alpha$, hence $\alpha l \in \widetilde{h}(s)$. We also get

$$\widetilde{h}(s)\downarrow\alpha l = (\widetilde{h}(s)\downarrow\alpha)\downarrow l = \widetilde{h}(p)\downarrow l = \widetilde{h}(q)$$

using the definition of $\leq$. □

Lemma 6.3 *Let G be a closed AC-graph. Suppose that G contains the edges $u \overset{\leq}{\Rightarrow} v \xrightarrow{l} w$. Then $\widetilde{\mathcal{L}}(u)\downarrow l = \widetilde{\mathcal{L}}(w)$.*

Proof. Follows from G being closed. □

Lemma 6.4 *Let G be a closed AC-graph. Suppose that G contains the edge $u_1 \overset{\leq}{\Rightarrow} u_2$. If $l \in \widetilde{\mathcal{L}}(u_1)$ and $l \in \widetilde{\mathcal{L}}(u_2)$, then $\widetilde{\mathcal{L}}(u_1)\downarrow l = \widetilde{\mathcal{L}}(u_2)\downarrow l$.*

Proof. For each $i \in 1..2$ we observe that for any node w_i and edges $u_i \overset{\leq}{\Rightarrow} v_i \xrightarrow{l} w_i$, we get from Lemma 6.3 that $\widetilde{\mathcal{L}}(u_i)\downarrow l = \widetilde{\mathcal{L}}(w_i)$. Thus, it is sufficient to choose two such nodes w_1 and w_2, and prove that $\widetilde{\mathcal{L}}(w_1) = \widetilde{\mathcal{L}}(w_2)$. This choice is possible since $l \in \widetilde{\mathcal{L}}(u_1)$ and $l \in \widetilde{\mathcal{L}}(u_2)$. From G being closed we get that there are edges $w_1 \overset{\leq}{\Rightarrow} w_2$ and $w_2 \overset{\leq}{\Rightarrow} w_1$. Thus, $\widetilde{\mathcal{L}}(w_1) = \widetilde{\mathcal{L}}(w_2)$ indeed holds. □

Theorem 6.5 *A closed AC-graph is solvable if and only if it is well-formed. If it is solvable, then $\mathcal{L}$ is the pointwise $\subseteq$-least solution.*

Proof. Let G be a closed AC-graph. Clearly, if G is solvable, then it is well-formed.

Assume now that G is well-formed. To show that $\mathcal{L}$ is a solution for G, we need to prove that if $u \overset{\leq}{\Rightarrow} v$ in G, then $\widetilde{\mathcal{L}}(u) \leq \widetilde{\mathcal{L}}(v)$. Suppose $l \in \widetilde{\mathcal{L}}(v)$. We must prove that $l \in \widetilde{\mathcal{L}}(u)$ and $\widetilde{\mathcal{L}}(u) \downarrow l = \widetilde{\mathcal{L}}(v) \downarrow l$. If we can prove $l \in \widetilde{\mathcal{L}}(u)$, then $\widetilde{\mathcal{L}}(u) \downarrow l = \widetilde{\mathcal{L}}(v) \downarrow l$ follows from Lemma 6.4.

First, if $u \in N$, then from G being well-formed, we get $u \xrightarrow{l} w$ in G for some w. Thus, $l \in \widetilde{\mathcal{L}}(u)$. Second, if $u \in S$, then from $u \xrightarrow{\leq} v$, we get from Lemma 6.3 that $l \in \widetilde{\mathcal{L}}(u)$.

To show that $\mathcal{L}$ is the pointwise $\subseteq$-least solution, we need to show that for any solution $h : S \to T$, $\mathcal{L}(s) \subseteq h(s)$ for all s. This follows directly from Lemma 6.2. $\square$

Intuitively, the pointwise $\subseteq$-least solution is the one where the types contain only those fields that are absolutely required by the type rules.

7 Algorithms

We first demonstrate how to close an AC-graph and how to check if an AC-graph is well-formed. Then we proceed to presenting the type inference algorithms.

7.1 Closure

To compute the closure of an AC-graph $(N, S, L, \leq)$, we use four data structures:

1. **ITC**: a data structure for incremental transitive closure. It maintains the transitive closure of a graph of $\leq$ edges during edge insertions. Specifically, it maintains an adjacency matrix TC such that after each edge insertion, $TC[x, y] = 1$ if and only if there is a path from x to y in the graph, and $TC[x, y] = 0$ otherwise. An edge insertion may result in the addition of more edges, and so on, recursively. The insert operation returns a list of all edges that have been added, represented as node pairs. Initialization and maintenance of **ITC** can be done in $O(n^3)$ time, where n is the number of nodes [15, 21].

2. **PE**: a data structure with potential $\leq$ edges. This data structure is computed in a preprocessing phase. It is a matrix of the same form as the adjacency matrix, but with each entry being a list of pairs of

nodes. It is computed by, for each pair of L edges $x_1 \xrightarrow{l_1} y_1$ and $x_2 \xrightarrow{l_2} y_2$, checking if $l_1 = l_2$, and if so, appending $[(y_1, y_2), (y_2, y_1)]$ to $\mathbf{PE}[x_1, x_2]$. This takes $O(|L|^2)$ time. The idea is that, for $l_1 = l_2$, if we later find edges $x \xrightarrow{\leq} x_1$ and $x \xrightarrow{\leq} x_2$ for some x, then we must insert $x_1 \xrightarrow{l_1} y_1$ and $x_2 \xrightarrow{l_2} y_2$ into **ITC**.

3. **M**: a matrix of the same form as the adjacency matrix. It is used to keep track of the accesses to **PE**. Specifically, $\mathbf{M}[x, y] = 1$ if and only if we have accessed $\mathbf{PE}[x, y]$, and $\mathbf{M}[x, y] = 0$ otherwise.

4. **Q**: the worklist. Specifically, **Q** is a list of pairs of nodes. Each pair (x, y) indicates that $x \xrightarrow{\leq} y$ must be inserted into **ITC**.

The algorithm works as follows. First compute **PE**, initialize **ITC** to be the empty graph, initialize **M** to be the 0-matrix, and initialize **Q** to be the list of $\leq$ edges of the input graph. Then repeat the following step until **Q** is empty:

- Let (x, y) be a pair in **Q**. Remove it and insert it into **ITC**. Let R be the list which is returned by the insert operation. For each (x, y) in R, find the immediate successors z of x in **ITC**, and if $\mathbf{M}[y, z] = 0$, then set $\mathbf{M}[y, z] = 1$ and set $\mathbf{M}[z, y] = 1$, and append $\mathbf{PE}[y, z]$ to **Q**.

Suppose $(N, S, L, \leq)$ is the input graph, and let n be the number of nodes in the graph, that is, $n = |N| + |S|$. There can be $O(n^2)$ new edges in **ITC**, and for each one, we consider $O(n)$ immediate successors of a given node. For the graphs of our application, $|L| \in O(n)$, so for those the total running time of the algorithm is $O(n^3)$.

7.2 Well-formedness

To check that an AC-graph is well-formed, do a depth-first search of the graph, maintaining for each node in N both a set of labels of outgoing L edges and a list of outgoing $\leq$ edges. Each time a node $u \in N$ is to be exited, check for each edge $u \xrightarrow{\leq} v$ that the label set for v is a subset of the label set for u. The time spent is:

- The search itself takes time proportional to the number of edges, which is $O(|L| + |\leq|)$ time.

- The maintenance of label sets takes time proportional to inserting $|L|$ elements, which is $O(|L| \log |L|)$ time.

- The maintenance of lists of $\leq$ edges takes $O(|\leq|)$ time.

- The checks take $O(|L|^2)$ time. (There is either zero or one comparison of labels for each pair of L edges.)

Suppose $(N, S, L, \leq)$ is the input graph, and let n be the number of nodes in the graph, that is, $n = |N| + |S|$. For the graphs of our application, $|L| \in O(n)$, so for those the total running time of the algorithm is $O(n^2)$.

7.3 Type inference

We have shown that the type inference problem with recursive types and subtyping is log space equivalent to solving AC-graphs. Using the characterization of Theorem 6.5, we get a straightforward type inference algorithm:

Input: A ς-term of size n.
1: Construct the corresponding AC-graph (in log space).
2: Close the graph (in $O(n^3)$ time).
3: Check if the resulting graph is well-formed (in $O(n^2)$ time).
4: If the graph is well-formed,
then output "typable" together with the automaton $\mathcal{M}$
else output "not typable".

The entire algorithm requires $O(n^3)$ time. Every subterm of the input corresponds to a state (s) in the automaton $\mathcal{M}$, and a suggestion for its type is represented by the language $\mathcal{L}(s)$.

Consider then the type inference problem with *finite* types and subtyping. Clearly, this problem is log space equivalent to finding finite solutions to AC-graphs. By Theorem 6.5, there exists a finite solution if and only if $\mathcal{L}$ is a finite solution. Thus, we obtain a type inference algorithm by modifying step 4 of the above algorithm so that it checks that $\mathcal{L}$ is finite. To do that, we check for a cycle in $\mathcal{M}$ with at least one non-ϵ transition reachable from some (s). This can be done in linear time in the size of the closed AC-graph using depth-first search. Thus, the entire algorithm requires $O(n^3)$ time.

Consider then the type inference problems *without* subsumption. This causes us to change the inequalities (8), (9), (12), and (13) to the corresponding equalities. Note that we leave the inequalities (11) and (15) unchanged. We can then repeat the development of Sections 4–6, but this time with a subclass of AC-systems, which we call SAC-systems (simple AC-systems). In SAC-systems, the allowed constraints are of the forms $W = W'$ or $V \leq [l_i : V_i \;^{i \in 1..n}]$, where $V, V_1, \ldots, V_n$ are variables, and W, W' are of the forms V or $[l_i : V_i \;^{i \in 1..n}]$. Similarly, there is a subclass of AC-graphs, which we call SAC-graphs. In SAC-graphs, most $\leq$ edges come in pairs, yielding "equality" edges. Specifically, consider an edge $u \stackrel{\leq}{\rightarrow} v$. Unless, u is an S node and v is an N node, then there is also an edge $v \stackrel{\leq}{\rightarrow} u$. An SAC-graph is well-formed if it is well-formed as an AC-graph. Moreover, the closure of an SAC-graph G is the smallest closed SAC-graph containing G as a subgraph. Clearly, the "AC-closure" and the "SAC-closure" of an SAC-graph are the same. SAC-graphs can be solved using the above algorithms, in time $O(n^3)$, both with recursive and finite types.

In summary:

Theorem 7.1 *All four type inference problems are solvable in $O(n^3)$ time.*

8 Completeness

We now prove that all four type inference problems are P-hard under log space reductions. Clearly, it is sufficient to show that the two problems of solving SAC-systems with finite or recursive types are P-hard. These two results are obtained by reductions of simple type inference for λ-calculus, and of the monotone circuit value problem, respectively, as explained in the following.

We first consider finding finite solutions to SAC-systems. Simple type inference for λ-calculus is P-complete under log space reductions [12]. It is log space equivalent to finding finite solutions to a finite set of equations of the forms $V = V' \rightarrow V''$ or $V = C$, where V, V', V'' are variables, and C is a constant. Variables range over the set of finite binary trees over the binary constructor $\rightarrow$ and some set of nullary constants. A solution is a map L from variables to such trees such that all equations are satisfied. We can in log space transform such a set of equations into an equivalent SAC-system by translating $V = V' \rightarrow V''$ into $V = [l : V' \;\; r : V'']$, and translating $V = C$ into $V = [\,]$. Thus, finding finite solutions to a SAC-system is P-hard.

We then consider finding arbitrary solutions to SAC-systems. A monotone circuit [7] is a directed acyclic graph G whose nodes, called *gates*, are of five different kinds:

1. *input gates* with no in-edge and one out-edge;
2. *and-gates* with two in-edges and one out-edge;
3. *or-gates* with two in-edges and one out-edge;
4. *fan-out gates* with one in-edge and two out-edges; and
5. a single *output gate* with one in-edge and no out-edges.

Furthermore, all gates are reachable from the input gate, and the output gate is reachable from all gates.

Every assignment a of truth values to the input gates of G can be extended uniquely to a truth value assignment $\overline{a}$ to all gates of G by defining:

1. if n is an input gate, then $\overline{a}(n) = a(n)$;
2. if n is an and-gate with predecessors n' and n'', then $\overline{a}(n) = \overline{a}(n') \wedge \overline{a}(n'')$;
3. if n is an or-gate with predecessors n' and n'', then $\overline{a}(n) = \overline{a}(n') \vee \overline{a}(n'')$;
4. if n is a fan-out gate with predecessor n' and out-edges n_1, n_2, then $\overline{a}(n_1) = \overline{a}(n')$ and $\overline{a}(n_2) = \overline{a}(n')$; and
5. if n is the output gate with predecessor n', then $\overline{a}(n) = \overline{a}(n')$.

The monotone circuit value problem [16] is the problem of deciding, given a monotone circuit G and an assignment a to the input gates of G, whether $\overline{a}(n) = \mathbf{true}$ for the output gate n. It is P-complete under log-space reductions [16].

We will now give a log space reduction of the monotone circuit value problem to the problem of finding arbitrary solutions to SAC-graphs. The construction is adapted from Henglein's proof of P-hardness of left-linear semi-unification [13]. That proof, in turn, is adapted from Dwork, Kanellakis, and Mitchell's proof of P-hardness of unification [8].

Given a monotone circuit and an assignment to input gates, we construct an SAC-graph as follows. Each gate in the circuit yields a small "gadget" consisting of a few nodes and edges. Each gadget has a pair of designated nodes for every in- and outedge of the encoded gate:

1. Each input gate is represented by two S nodes n, n'. The output node pair is (n, n').

2. Each and-gate is represented by three S nodes n, n', n''. The two input node pairs are (n, n') and (n', n''). The output node pair is (n, n'').

3. Each or-gate is represented by two S nodes n, n'. The two input node pairs are (n, n') and (n, n'). The output node pair is also (n, n').

4. Each fan-out gate is represented by two N nodes n, n' and four S nodes n_1, n_2, n'_1, n'_2, and four edges $n \xrightarrow{0} n_1$, $n \xrightarrow{1} n_2$, $n' \xrightarrow{0} n'_1$, and $n' \xrightarrow{1} n'_2$, where 0 indicates "left" and 1 indicates "right". The input node pair is (n, n'). The output node pairs are (n_1, n'_1) and (n_2, n'_2).

5. The output gate is represented by two N nodes n_1, n_2 and one S node n'_2, and one edge $n_2 \xrightarrow{l} n'_2$. The input node pair is (n_1, n_2).

Each edge in the circuit yields one or more edges in the SAC-graph, as follows. Suppose there is an edge from gate g to gate g' in the circuit. We connect the corresponding output pair (n, n') in the representation of g to the corresponding input pair (m, m') in the representation of g' by adding the edges $m \xrightarrow{\leq} n$ and $n' \xrightarrow{\leq} m'$.

Finally, consider an input gate which is assigned **true** by a. Suppose the gate is represented by n, n'. We add the edge $n \xrightarrow{\leq} n'$.

Having now constructed the SAC-graph G, consider the two N nodes n_1, n_2 in the representation of the output gate g. Clearly, the closure of G contains an edge $n_1 \xrightarrow{\leq} n_2$ if and only if $\overline{a}(g) = \mathbf{true}$. By Theorem 6.5 it follows, that $\overline{a}(g) = \mathbf{true}$ if and only if G is not solvable. Thus, finding arbitrary solutions to a SAC-system is P-hard.

In summary:

Theorem 8.1 *All four type inference problems are P-complete.*

9 Examples

We now give three examples of how the type inference algorithms work. We use the algorithm for the system with recursive types and subtyping.

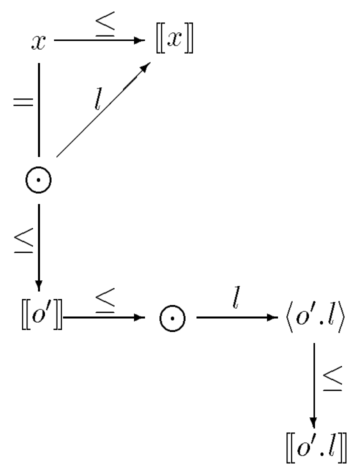
The first example is taken from Section 2. Consider the object $o \equiv [l = \varsigma(x)x.l]$. The expression $o.l$ yields the following constraints and in turn the following graph:

$$
\begin{array}{cl}
o.l & \left[\begin{array}{l} [\![o]\!] \leq [l : \langle o.l \rangle] \\ \langle o.l \rangle \leq [\![o.l]\!] \end{array} \right. \\
o & \left[\begin{array}{l} [l : [\![x.l]\!]] \leq [\![o]\!] \\ x = [l : [\![x.l]\!]] \end{array} \right. \\
x.l & \left[\begin{array}{l} [\![x]\!] \leq [l : \langle x.l \rangle] \\ \langle x.l \rangle \leq [\![x.l]\!] \end{array} \right. \\
x & x \leq [\![x]\!]
\end{array}
\qquad
\begin{tikzcd}
x \arrow[r, "\leq"] \arrow[d, dash, "="'] & {[\![x]\!]} \arrow[r, "\leq"] & \odot \arrow[d, "l"] \\
\odot \arrow[r, "l"] \arrow[d, "\leq"'] & {[\![x.l]\!]} & \langle x.l \rangle \arrow[l, "\leq"'] \\
{[\![o]\!]} \arrow[r, "\leq"] & \odot \arrow[r, "l"] & \langle o.l \rangle \arrow[d, "\leq"] \\
 & & {[\![o.l]\!]}
\end{tikzcd}
$$

To the left of the constraints we have written from which expressions they are generated. In the graph we indicate N nodes with the symbol $\odot$. To close the graph we only need to add three equality edges between the nodes $[\![x.l]\!]$, $\langle x.l \rangle$, and $\langle o.l \rangle$, and of course edges to make $\leq$ transitive. Clearly, the resulting graph is well-formed, hence solvable. By transforming the graph into an automaton, we get that the variable x has the $\subseteq$-least annotation $[l : [\,]]$.

The second example is also taken from Section 2. Consider the object $o' \equiv [l = \varsigma(x)x]$. The expression $o'.l$ yields the following constraints and in turn the following graph:

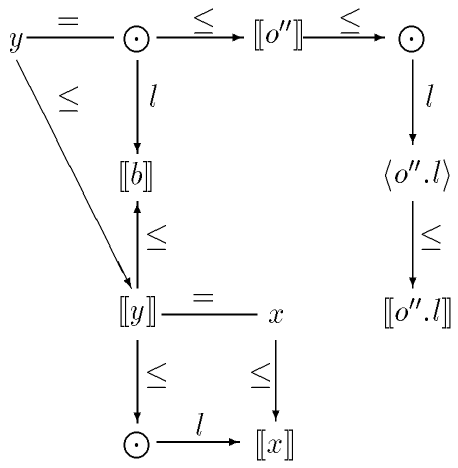
$$
\begin{array}{ll}
o'.l & \left[\begin{array}{l} [\![o']\!] \leq [l : \langle o'.l \rangle] \\ \langle o'.l \rangle \leq [\![o'.l]\!] \end{array}\right. \\
o' & \left[\begin{array}{l} [l : [\![x]\!]] \leq [\![o']\!] \\ x = [l : [\![x]\!]] \end{array}\right. \\
x & \;\; x \leq [\![x]\!]
\end{array}
$$



To close the graph we only need to add an equality edge between the nodes $[\![x]\!]$ and $\langle o'.l \rangle$, and of course edges to make $\leq$ transitive. Clearly, the resulting graph is well-formed, hence solvable. By transforming the graph into an automaton, we get that the variable x has the $\subseteq$-least annotation $[l : [\,]]$.

The third example is also taken from Section 2. Consider the object $o'' \equiv [l = \varsigma(y)b]$, where $b \equiv y.l \Leftarrow \varsigma(x)x$. The expression $o''.l$ yields the following constraints and in turn the following graph:

$$
\begin{array}{ll}
o''.l & \left[\begin{array}{l} [\![o'']\!] \leq [l : \langle o''.l \rangle] \\ \langle o''.l \rangle \leq [\![o''.l]\!] \end{array}\right. \\
o'' & \left[\begin{array}{l} [l : [\![b]\!]] \leq [\![o'']\!] \\ y = [l : [\![b]\!]] \end{array}\right. \\
b & \left[\begin{array}{l} [\![y]\!] \leq [\![b]\!] \\ [\![y]\!] = x \\ [\![y]\!] \leq [l : [\![x]\!]] \end{array}\right. \\
x & \;\; x \leq [\![x]\!] \\
y & \;\; y \leq [\![y]\!]
\end{array}
$$



To close the graph we only need to add an equality edge between the nodes $[\![b]\!]$ and $\langle o''.l\rangle$, and of course edges to make $\leq$ transitive. Clearly, the resulting graph is well-formed, hence solvable. By transforming the graph into an automaton, we get that the variables x, y both have the $\subseteq$-least annotation $[l : [\,]]$.

The fourth example illustrates what happens if the program is *not* typable. Consider the object $a \equiv [\,]$. The expression $a.l$ yields the following constraints and in turn the following graph:

$$
\begin{array}{ll}
a & [\,] \leq [\![a]\!] \\
a.l & \left[\begin{array}{l} [\![a]\!] \leq [l : \langle a.l\rangle] \\ \langle a.l\rangle \leq [\![a.l]\!] \end{array}\right.
\end{array}
\qquad
\begin{tikzcd}
\odot \arrow[r, "\leq"] & {[\![a]\!]} \arrow[r, "\leq"] & \odot \arrow[d, "l"] \\
{[\![a.l]\!]} & & \langle a.l\rangle \arrow[ll, "\leq"']
\end{tikzcd}
$$

To close the graph we only need to add a single edge to make $\leq$ transitive. The resulting graph is *not* well-formed, however, because the leftmost node does not have an outgoing l edge. Thus, the graph is not solvable.

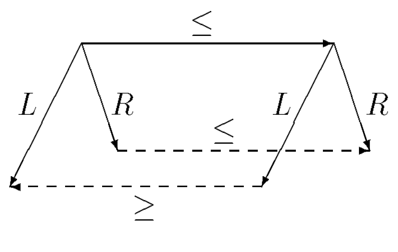
10 Related work

Type inference with subtyping has been studied for λ-calculi. In that setting, the problem of type inference with *atomic* subtyping can be presented as follows. Suppose types are finite trees over the binary function type constructor $\rightarrow$ and a partially ordered set $(\Sigma, \leq)$ of constants, that is, base types. The ordering on types is the smallest extension of $\leq$ such that $s \rightarrow t \leq s' \rightarrow t'$ if and only if $s' \leq s$ and $t \leq t'$. The type rules are those of simply typed λ-calculus with typed constants together with the subsumption rule. The type inference problem is to decide if a given λ-term is typable. This problem was introduced by Mitchell in 1984 [19] who showed that type inference is computable in *NEXPTIME* in the size of the program. If $(\Sigma, \leq)$ is a disjoint union of lattices, then type inference is computable in polynomial time [23]. Moreover, if $(\Sigma, \leq)$ is "tree-like", then type inference is computable in polynomial time [6]. In general, the type inference problem is *PSPACE*-hard [14, 23]. Type inference with atomic subtyping has also been studied in combination with ML polymorphism [10, 11].

Objects do not have base types, so type inference with atomic subtyping does not apply to Abadi and Cardelli's calculus. The object types considered in this paper are rather like *record* types. Type inference for λ-calculi with records but no subtyping has been studied by Wand [24] and Remy [22]. In the presence of subtyping, unification-based approaches to type inference seem not to apply.

Type inference for λ-calculi with records *and* subtyping has been studied by Eifrig, Smith, and Trifonov [9], using the approach to type inference of Aiken and Wimmers [5]. Their algorithm does not immediately apply to Abadi and Cardelli's calculus because of the following difference between the subtyping relations. For records, the conventional subtyping relations makes every record type constructor covariant in all arguments. For example, $[l : A, m : C]$ is a subtype of $[l : B]$ provided that A is a subtype of B. In Abadi and Cardelli's type system, $[l : A, m : C]$ can only be a subtype of $[l : B]$ if $A = B$. The conventional subtyping relation for records is unsound in the case of Abadi and Cardelli's calculus [2]. Even though the subtyping relation in Abadi and Cardelli's type system is smaller than the conventional one for record types, it yields a more complicated definition of closed AC-graphs, as illustrated in the following.

Our algorithms can easily be extended to handle also λ-terms and a contravariant function space constructor, in outline as follows. The constraints for λ-terms are as usual (see for example [17]). The definition of AC-graph needs to be extended with a set $N_{\rightarrow}$ of nodes and a set $E_{\rightarrow}$ of edges. Each edge in $E_{\rightarrow}$ is labeled by either L or R, where $L \notin \mathcal{N}$ and $R \notin \mathcal{N}$, where $\mathcal{N}$ is the set of method names. Each node in $N_{\rightarrow}$ corresponds to a function type constructor, and it has exactly two outgoing $E_{\rightarrow}$ edges, both to S nodes: one edge labeled L (indicating the left argument of $\rightarrow$) and one edge labeled R (indicating the right argument of $\rightarrow$). The definition of closed AC-graph needs to be extended so that it is closed under the rule which says that the dashed edges exists whenever the solid ones do:



Clearly, closing such graphs can be done in $O(n^3)$ time. Finally, the notion of well-formed AC-graph needs to be restricted so that:

1. No well-formed AC-graph can contain edges of the forms $u \xrightarrow{\leq} v$ or $v \xrightarrow{\leq} u$ where $u \in N$ and $v \in N_{\rightarrow}$,

2. No well-formed AC-graph can contain a pair of edges of the forms $u \xrightarrow{\leq} v$ and $u \xrightarrow{\leq} w$ where $u \in S$, $v \in N$, and $w \in N_{\rightarrow}$, and

3. No well-formed AC-graph can contain a pair of edges of the forms $v \xrightarrow{\leq} u$ and $w \xrightarrow{\leq} u$ where $u \in S$, $v \in N$, and $w \in N_{\rightarrow}$.

The extra checks can be done in $O(n^2)$ time where n is the number of nodes. Given proper definitions of the new set of types and the new $\leq$ relation (along the lines of [17]), it should be possible to prove that Theorem 6.5 holds. For the program $\lambda(x)((x.a)x)$, the most powerful of the type inference algorithms infers that x has the $\subseteq$-least annotation $[a : [\,] \rightarrow [\,]]$.

For a λ-calculus with records, our algorithm can be modified to handle the conventional subtyping relation for record types, simply by changing the definition of closed AC-graph such that if $u, v \in N$, $s, t \in S$, and the edges $u \xrightarrow{\leq} v$, $u \xrightarrow{l} s$, $v \xrightarrow{l} t$ exist, then also $s \xrightarrow{\leq} t$ exists. Clearly, closing a graph can be done in $O(n^3)$ time. The proof that Theorem 6.5 holds is left as an exercise for the reader.

In remains to be seen how to extend our algorithm to deal with ML polymorphism.

11 Conclusion

The type inference problems we have addressed are related to those treated by Eifrig, Smith, and Trifonov [9]. Specifically, our algorithm can be modified to handle functions and records, and it seems possible to modify their algorithm to handle the object calculus. The results of the two approaches appear to be complimentary. We have completeness results and efficient algorithms, while they have incremental algorithms that can handle ML polymorphism. Future work may attempt to combine the two approaches.

Acknowledgements. The author thanks Paris Kanellakis for comments that prompted me to find the reduction from AC-systems to the type inference problem. The author also thanks Fritz Henglein and Harry Mairson for discussions on proving P-hardness. Moreover, the author thanks Devdatt Dubhashi, Gudmund Frandsen, and Sven Skyum for discussions on algorithms for closing AC-graphs. Finally, the author thanks Ole Agesen and the anonymous referees for many helpful comments on a draft of the paper. A preliminary version of this paper appeared in Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science. The results of this paper were obtained while the author was at Northeastern University, Boston.

References

[1] Martín Abadi and Luca Cardelli. A semantics of object types. In *Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 332–341, 1994.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects. Manuscript, February 1994.

[3] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *Proc. ESOP'94, European Symposium on Programming*, pages 1–25. Springer-Verlag (*LNCS* 788), 1994.

[4] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Proc. TACS'94, Theoretical Aspects of Computing Sofware*, pages 296–320. Springer-Verlag (*LNCS* 789), 1994.

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

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

[7] Ravi B. Boppana and Michael Sipser. The complexity of finite functions. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume A, chapter 14, pages 757–804. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[8] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[9] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and it application to OOP. In *Proc. Mathematical Foundations of Programming Semantics*, 1995. To appear.

[10] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *Proc. ESOP'88, European Symposium on Programming*, pages 94–114. Springer-Verlag (*LNCS* 300), 1988.

[11] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. TAPSOFT'89*, pages 167–183. Springer-Verlag (*LNCS* 352), 1989.

[12] Fritz Henglein. Simple type inference and unification. Technical Report (SETL Newsletter) 232, Courant Institute of Mathematical Sciences, New York University, October 1988.

[13] Fritz Henglein. Fast left-linear semi-unification. In *Proc. ICCI'90, International Conference on Computing and Information*, pages 82–91. Springer-Verlag (*LNCS* 468), 1990.

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

[15] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[16] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume A, chapter 17, pages 869–941. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[17] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. Also in Proc. FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[18] Bent B. Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter

Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[19] John Mitchell. Coercion and type inference. In *Eleventh Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[20] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *LICS'93, Eighth Annual Symposium on Logic in Computer Science*, pages 26–38, 1993.

[21] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Springer-Verlag (*LNCS* 314), 1988.

[22] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Sixteenth Symposium on Principles of Programming Languages*, pages 77–88, 1989.

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

[24] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *LICS'89, Fourth Annual Symposium on Logic in Computer Science*, pages 92–97, 1989.