Breaking through the n^3 barrier: Faster object type inference

Fritz Henglein*

December 17th, 1996

Abstract

Abadi and Cardelli [AC96] have presented and investigated object calculi that model most object-oriented features found in actual object-oriented programming languages. The calculi are innate object calculi in that they are *not* based on λ-calculus. They present a series of type systems for their calculi, four of which are first-order. Palsberg [Pal95] has shown how typability in each one of these systems can be decided in time $O(n^3)$, where n is the size of an untyped object expression, using an algorithm based on dynamic transitive closure. He also shows that each of the type inference problems is hard for polynomial time under log-space reductions.

In this paper we show how we can break through the *(dynamic) transitive closure bottleneck* and improve each one of the four type inference problems from $O(n^3)$ to the following time complexities:

	no subtyping	subtyping
w/o rec. types	$O(n)$	$O(n^2)$
with rec. types	$O(n \log^2 n)$	$O(n^2)$

The key ingredient that lets us "beat" the worst-case time complexity induced by using general dynamic transitive closure or similar algorithmic methods is that object subtyping is *invariant*: an object type is a subtype of a "shorter" type with a subset of the field names if and only if the common fields have *equal* types.

*Affiliation: DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, Email: henglein@diku.dk. This research was partially supported by the Danish Research Council, project *DART*.

1 Introduction

This paper is self-contained in its technical contents. The algorithmic methods are all pre-1975 and can be found in any textbook on algorithms; e.g. [AHU74, CLR90].

We shall rely on Abadi and Cardelli's book [AC96] and Palsberg [Pal95] for a motivation of the relevance of the calculi, their typing systems and the practical importance of efficient and easily implementable type inference. Even though the presentation is for the pure calculi presented by Abadi and Cardelli we believe that the techniques and results presented here can be used in object type systems for real-scale programming languages, though it should be clear that complex interactions of object-oriented features in full-blown language designs may make application of our techniques difficult or impossible.

1.1 Object types

Object types are described by the grammar

$$A, B ::= X \mid [l_i : A_i]_{i=1\ldots n} \mid \mu X.A'$$

where X ranges over an infinite set of *type variables*, the l_i in object types are pairwise distinct *field names*, and the order of *fields* $l_i : A_i$ in an object type $[l_i : A_i]_{i=1\ldots n}$ is unimportant. *Recursive (object) types* of the form $\mu X.A$ are denotations for the (possibly) infinite trees obtained from their unfolding. We say A and B are equal and write $A = B$ if A and B are equal up to reordering of fields and infinite unfolding of recursive type occurrences. For example, the following three types are equal to each other.

$$\begin{array}{l} \mu X.[l_1 : [l_1 : X, l_2 : []], l_2 : []] \\ \mu Y.[l_1 : Y, l_2 : []] \\ [l_1 : \mu Y.[l_1 : Y, l_2 : []], l_2 : []] \end{array}$$

Henceforth equal object types will be interchangeable in any context. *Type environments* are lists of pairs consisting of a variable x and a type A.

1.2 Object type systems

The object calculi we shall investigate are:

- $\mathbf{Ob}_1$[AC96, Section 7, p. 83],
- $\mathbf{Ob}_{1\mu}$[AC96, Section 9, p. 114],

Rule	$\mathbf{Ob}_1$	$\mathbf{Ob}_{1\mu}$	$\mathbf{Ob}_{1<:}$	$\mathbf{Ob}_{1<:\mu}$
(VAL)	$\surd$	$\surd$	$\surd$	$\surd$
(OBJ)	$\surd$	$\surd$	$\surd$	$\surd$
(SEL)	$\surd$	$\surd$	$\surd$	$\surd$
(UPD)	$\surd$	$\surd$	$\surd$	$\surd$
(SUB)			$\surd$	$\surd$
(REFL)			$\surd$	$\surd$
(TRANS)			$\surd$	$\surd$
(OBSUB)			$\surd$	$\surd$
rec. type?		$\surd$		$\surd$

Table 1: Type system definitions

- $\mathbf{Ob}_{1<:}$[AC96, Section 8, p. 94], and
- $\mathbf{Ob}_{1<:\mu}$[AC96, Section 9, p. 117].

The inference rules for these systems are given in Figure 1.[1] Table 1 shows which rules make up which inference system.

Note that *object expressions* are *untyped*; in particular, no type declarations are given for bound expression variables. We also elide the explicit *fold* and *unfold* constructs [AC96, Section 9] since recursive types are denotations for the (possibly) infinite trees obtained from their complete unfolding. Note also that regular infinite trees denoted by recursive types are only allowed in the type systems that explicitly permit them. See Table 1. Type inference can be thought of as inserting explicit type declarations and *fold*/*unfold* constructs in a given untyped object expression.

We say an object expression a is typable in type system O where $O \in \{\mathbf{Ob}_1, \mathbf{Ob}_{1\mu}, \mathbf{Ob}_{1<:}, \mathbf{Ob}_{1<:\mu}\}$ if there exists a derivation of *typing judgement* $E \vdash a : A$ in system O for some type environment E and type A. In this case we write $E \vdash_O a : A$.

[1]For simplicity's sake we use somewhat informal notation in that we do not axiomatize well-formedness of type environments.

(VAL) $$E, x : A, E' \vdash x : A \qquad (x \notin domain\, E')$$

(OBJ) $$\frac{E, x_i : [l_i : B_i]_{i \in \mathcal{I}} \vdash b_i : B_i \qquad (\forall i \in \mathcal{I})}{E \vdash [l_i = \varsigma(x_i) b_i]_{i \in \mathcal{I}} : [l_i : B_i]_{i \in \mathcal{I}}}$$

(SEL) $$\frac{E \vdash a : [l_i : B_i]_{i \in \mathcal{I}}}{E \vdash a.l_j : B_j} \qquad (j \in \mathcal{I})$$

(UPD) $$\frac{E \vdash a : [l_i : B_i]_{i \in \mathcal{I}} \qquad E, x : [l_i : B_i]_{i \in \mathcal{I}} \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x) b : [l_i : B_i]_{i \in \mathcal{I}}} \quad (j \in \mathcal{I})$$

(SUB) $$\frac{E \vdash a : A \qquad \vdash A \leq B}{E \vdash a : B}$$

(REFL) $$\vdash A \leq A$$

(TRANS) $$\frac{\vdash A \leq A' \qquad \vdash A' \leq A''}{\vdash A \leq A''}$$

(OBSUB) $$\vdash [l_i : B_i]_{i \in \mathcal{J}} \leq [l_i : B_i]_{i \in \mathcal{I}} \qquad (\mathcal{I} \subseteq \mathcal{J})$$

Figure 1: Object type inference rules

$$(\mathrm{VAL})^n \quad E, x : A, E'' \vdash x : A' \quad (x \notin domain\ E'') \quad A = A'$$

$$(\mathrm{OBJ})^n \quad \frac{E, x_i : A \vdash b_i : B_i' \qquad (\forall i \in \mathcal{I})}{E \vdash [l_i = \varsigma(x_i)b_i]_{i\in\mathcal{I}} : A'} \qquad \left\{ \begin{array}{l} B_i' \leq B_i, i \in \mathcal{I} \\ A = [l_i : B_i]_{i\in\mathcal{I}} \\ A' = A \end{array} \right\}$$

$$(\mathrm{SEL})^n \quad \frac{E \vdash a : A}{E \vdash a.l_j : B} \qquad A \leq [l_j : B]$$

$$(\mathrm{UPD})^n \quad \frac{E \vdash a : A \qquad E, x : A' \vdash b : B'}{E \vdash a.l_j \Leftarrow \varsigma(x)b : A''} \qquad \left\{ \begin{array}{l} A = A' \\ A = A'' \\ B' \leq B \\ A \leq [l_j : B] \end{array} \right\}$$

Figure 2: Normalized type inference rules

2 Type inference for $\mathbf{Ob}_{1<:}$ and $\mathbf{Ob}_{1<:\mu}$

2.1 Normalized derivations

Every typing derivation can be *normalized* such that applications of (REFL) and (TRANS) are completely eliminated, and applications of (SUB) occur only as the last steps in object formation and method update. Let us write $A \leq B$ if $A = B = X$ for some type variable X or $\vdash A \leq B$ is derivable using a single application of Rule (OBSUB).

Normalized derivations are captured by the *normalized inference rules* in Figure 2. We capture the completeness of normalized derivations with respect to the general type systems by the following theorem:

Theorem 2.1 *Let $\boldsymbol{Ob}_{1<:}{}^n$ (without recursive types) and $\boldsymbol{Ob}_{1<:\mu}{}^n$ (with recursive types) be defined by the inference rules given in Figure 2. Then:*

1. *$E \vdash_{\boldsymbol{Ob}_{1<:}} e : A$ if and only if there exists A' such that $E \vdash_{\boldsymbol{Ob}_{1<:}{}^n} e : A'$ and $A' \leq A$.*

2. *$E \vdash_{\boldsymbol{Ob}_{1<:\mu}} e : A$ if and only if there exists A' such that $E \vdash_{\boldsymbol{Ob}_{1<:\mu}{}^n} e : A'$ and $A' \leq A$.*

The proofs of both cases are identical. In each case the "if" direction is immediate since the inference rules in Figure 2 are derivable from the standard object type inference rules in Figure 1. As for the "only if" direction we need to prove a somewhat stronger property, due to Rule $(\mathrm{UPD})^n$ where type A is moved into the assumptions of the second premise. Let us write $E \leq E'$ if $E = x_1 : A_1, \ldots, x_n : A_n$, $E' = x_1 : A'_1, \ldots x_n : A'_n$ and $A_i \leq A'_i$ for $1 \leq i \leq n$.

Lemma 2.2 *Let O be $\boldsymbol{Ob}_{1<:}$ or $\boldsymbol{Ob}_{1<:\mu}$. If $E \vdash_O a : A$ and $E' \leq E$ then there exists A' such that $E' \vdash_{O^n} e : A'$ and $A' \leq A$.*

PROOF This lemma is proved by straightforward rule induction on the rules in Figure 1:

Rule (VAL): Assume $E_1, x : A, E_2 \vdash_O x : A$ by Rule (VAL). Let $E' \leq E$; that is, $E' = E'_1, x : A', E'_2$ with $E'_1 \leq E_1, A \leq A'$ and $E'_2 \leq E_2$. By Rule $(\mathrm{VAL})^n$ we derive $E' \vdash_{O^n} x : A'$ and we are done since $A' \leq A$.

Rule (OBJ): Assume $E \vdash_O [l_i = \varsigma(x_i)b_i]_{i=1\ldots n} : A$ is derived by Rule (OBJ) from $E, x_i : A \vdash_O b_i : B_i \quad (\forall i \in 1\ldots n)$. Let $E' \leq E$. Then $E', x_i : A \leq E, x_i : A$, and by induction hypothesis we have $E', x_i : A \vdash_{O^n} B'_i$ with $B'_i \leq B_i$ for $1 \leq i \leq n$. Thus we can apply Rule $(\mathrm{OBJ})^n$ to derive $E' \vdash_{O^n} \varsigma(x_i)b_i]_{i=1\ldots n} : A$.

Rule (SEL): Assume $E \vdash_O a.l_j : B_j$ is derived from $E \vdash_O a : A$ by Rule (SEL). In this case $A \leq [l_j : B_j]$ since A must contain the field $l_j : B_j$. Let $E' \leq E$. By induction hypothesis we have $E' \vdash_{O^n} a : A'$ with $A' \leq A$. Since $\leq$ is transitively closed we have $A' \leq [l_j : B_j]$. Thus Rule $(\mathrm{SEL})^n$ is applicable and we obtain $E' \vdash_{O^n} a.l_j : B_j$.

Rule (UPD): Assume $E \vdash_O a.l_j \Leftarrow \varsigma(x)b : A$ is derived from $E \vdash_O a : A$ and $E, x : A \vdash_O b : B_j$ by Rule (UPD). Thus $A \leq [l_j : B_j]$ since A must contain the field $l_j : B_j$. Let $E' \leq E$. By induction hypothesis for the first premise we have $E' \vdash_{O^n} a : A'$ with $A' \leq A$. Consequently, $E', x : A' \leq E, x : A$ and by the induction hypothesis for the second premise we have $E', x : A' \vdash B'_j$ for some $B'_j \leq B_j$. Note that we have $A' \leq [l_j : B_j]$ since $A' \leq A \leq [l_j : B_j]$, and $B'_j \leq B_j$. Thus Rule $(\mathrm{UPD})^n$ is applicable and yields $E' \vdash_{O^n} a.l_j \Leftarrow \varsigma(x)b : A'$. Since $A' \leq A$ we are finished.

Rule (SUB): Assume $E \vdash_O a : B$ is derived from $E \vdash_O a : A$ and $\vdash A \leq B$ by Rule (SUB). Let $E' \leq E$. By induction hypothesis we have $E' \vdash_{O^n}$

$a : A'$ with $A' \leq A$. Since $\vdash A \leq B$ if and only if $A \leq B$ and $\leq$ is transitively closed it follows that $A' \leq B$, and we are done.

Since these are all the rules that derive judgements of the form $E \vdash a : A$ we are finished. $\square$

2.2 Constraint formulation

Every inference rule in Figure 2 is *linearized* in the sense that every metavariable A, A', B_i, B_i', B' occurs at most once in each rule. The required relation on types denoted by the metavariables is expressed by side conditions for each rule. Since the inference rules are strongly syntax-directed (there is exactly one applicable rule for each expression construct) every object expression determines a unique *derivation skeleton*[2], where a *unique type variable* annotates every subexpression occurrence, plus associated *constraints* on the type variables reflecting the side conditions. The type variables in the derivation skeleton correspond to the meta variables in Figure 2,

An *equational constraint* is a formal equation of the form $A = B$. A *subtyping constraint* is a formal inequational formula of the form $A \leq B$. A *valuation* ρ is a mapping from type variables to object types, which is canonically extended to object types. We say ρ is *without recursive types* if it maps every type variable to a finite type; that is, object types describable without μ. We write $\rho(A)$ for the application of ρ to A. An equational constraint $A = B$ is *solved* by valuation ρ if $\rho(A) = \rho(B)$; that is, $\rho(A)$ and $\rho(B)$ are equal. Similarly, a subtyping constraint $A \leq B$ is solved by ρ if $\rho(A) \leq \rho(B)$; that is, $\rho(A) = \rho(B) = X$ for some type variable X or $\rho(A)$ is an object type that contains all the fields of $\rho(B)$. A valuation solves a set of constraints if it solves each individual constraint in it. Finally, a set of constraints is *solvable with recursive types* if there is a valuation that solves it; if there is solving valuation without recursive types then the constraints are *solvable without recursive types*.

Note that an object expression is typable in $\mathbf{Ob}_{1<:}$, respectively $\mathbf{Ob}_{1<:\mu}$, if and only if the constraints its derivation skeleton generates are *solvable*; that is, if and only if there is a valuation (without, respectively with, recursive types) such that all equational and subtyping constraints are satisfied. This is summarized in the following lemma:

[2] Uniqueness is up to renaming of type variables.

Lemma 2.3 *Given object expression a there is a set of equational and subtyping constraints C_a such that C_a is solvable with/without recursive types if and only if a is typable in $\boldsymbol{Ob}_{1<:\mu}/\boldsymbol{Ob}_{1<:}$.*

Furthermore, C_a can be computed from a in linear time.

For the time complexity in the lemma we may assume that a is given as a syntax tree since parsing including identification of field names and variables can be done in linear time [CP95, CP91]. In particular, given an object expression of length n (measured in bits) we may assume that variables and field names are represented by natural numbers in the interval $[1 \ldots m]$ where $m = O(n)$ and a syntax tree of size $O(n)$. The following algorithm, which can be implemented in time $O(n)$, generates C_a: Annotate every node in the syntax tree by a unique type variable. Then generate the constraints corresponding to the side conditions in Figure 2 for each node in the syntax tree.

2.3 Constraint closure

The basic strategy of constraint-based techniques is as follows:

1. Close the set of constraints under (some of) its *logical entailments*.
2. If the entailments include $\perp$ (absurdity, inconsistency) then the constraints are unsolvable; otherwise extract a canonical solution from the closed set of constraints.

Alternatively, the closure conditions can be thought of as rewritings on constraint systems which preserve the set of all solutions. The rewritings must be strong enough to construct direct evidence for inconsistency (that is, "materialize" an inconsistency) whenever a set of constraints is unsolvable.

Figure 3 provides a formal system for deriving equational and subtyping constraints that are entailed by a given set of constraints C. We have added the *absurd constraint* $\perp$, which is, by definition, unsolvable. To summarize, constraints P are generated by the following grammar:

$$P ::= \perp \mid A = B \mid A \leq B.$$

The rules in Figure 3 for system $\mathbf{Ob}_{1<:\mu}$. For $\mathbf{Ob}_{1<:}$ we add another rule to those of Figure 3:

$$\text{(CYCLE)} \qquad \frac{C \vdash B_0 \leq [l_{i_1} : B_1], \ldots, B_{k_1} \leq [l_k : B_k], B_k \leq [l_0 : B_0]}{C \vdash \perp}.$$

(HYP$^=$) $$C \cup \{A = B\} \vdash A = B$$

(SYM) $$\frac{C \vdash A = B}{C \vdash B = A}$$

(TRANS$^=$) $$\frac{C \vdash A = A' \qquad C \vdash A' = A''}{C \vdash A = A''}$$

(SUB$^=$) $$\frac{C \vdash A = B}{C \vdash A \leq B} \qquad \frac{C \vdash A = B}{C \vdash B \leq A}$$

(HYP$^\leq$) $$C \cup \{A \leq B\} \vdash A \leq B$$

(TRANS$^\leq$) $$\frac{C \vdash A \leq A' \qquad C \vdash A' \leq A''}{C \vdash A \leq A''}$$

(DECOMP$^\leq$) $$\frac{C \vdash [l_i : B_i]_{i \in \mathcal{I}} \leq [l_j : B'_j]_{j \in \mathcal{J}}}{C \vdash B_j = B'_j} \qquad (j \in \mathcal{J} \subseteq \mathcal{I})$$

(ABSURD$^\leq$) $$\frac{C \vdash [l_i : B_i]_{i \in \mathcal{I}} \leq [l_j : B'_j]_{j \in \mathcal{J}}}{C \vdash \bot} \qquad (\mathcal{J} \not\subseteq \mathcal{I})$$

(UNIQLAB) $$\frac{C \vdash A \leq [l_i : B_i]_{i \in \mathcal{I}} \quad C \vdash A \leq [l_j : B'_j]_{j \in \mathcal{J}}}{C \vdash B_k = B'_k} \qquad (k \in \mathcal{I} \cap \mathcal{J})$$

Figure 3: Entailment rules for equational and subtyping constraints

Lemma 2.4 *1. Constraints C are unsolvable with recursive types if and only if $C \vdash \bot$ (without rule (CYCLE)).*

2. Constraints C are unsolvable without recursive types if and only if $C \vdash \bot$ (with rule (CYCLE)).

PROOF It is easy to check that the inference rules are sound in the sense that, if $C \vdash P$, then any solution of C is a solution of P, where $\bot$ has no solutions at all. Thus, if $C \vdash \bot$ then C is unsolvable.

Conversely, assume that we cannot derive $C \vdash \bot$. Then we can define a solution ρ of C as follows. Let Subseq be a function that, given a set S of pairs of field names and object types with possibly repeated field names, selects a subset of these pairs such that every field name occurring as a first component in S is present in the subsequence, but only once. In other words, Subseq turns a set of field name/object type pairs into a valid object type. Given a set of constraints C, consider the following set of equations for variables X:

$$X = \mathrm{Subseq}(M_C(X))$$

where $M_C(X) = \{(l : \hat{\rho}(B)) \mid C \vdash X \leq [l : B]\}$ and $\hat{\rho}(X) = X, \hat{\rho}([l_i : B_i]_{i=1\ldots n}) = [l_i : \hat{\rho}(B_i)]_{i=1\ldots n}$. This is a *contractive* set of equations. Thus it defines a unique valuation ρ. The closure properties expressed in C guarantee that this is a well-defined solution. (Note that rule (CYCLE) guarantees that the type variables can be topologically ordered, resulting in a solution without recursive types.) $\square$

2.4 Closure algorithm

We now show how to compute efficiently the logical implications of a constraint set generated according to Lemma 2.3. Note that the inference system in Figure 3 has the *subformula property*: If $C \vdash A = B$ or $C \vdash A \leq B$ then both A and B occur in C. This alone shows that the set of entailments C^* of C is finite and can be computed in polynomial time, using generic methods such as DATALOG bottom-up evaluation. Such methods, however, take time $\Omega(n^3)$ or worse. In this section we present an asymptotically improved algorithm that executes in time $O(n^2)$.

The algorithm operates on *flow graphs*. Flow graphs are term graphs extended with directed *flow edges* and undirected *equivalence edges*. Nodes are labeled with either a type variable or with the object type constructor [.]. Type variables have no children whereas object constructor labeled nodes may have children that can be reached along *tree edges* that are labeled by field names $l_1, \ldots$. We represent a constraint set C by a term graph where every type variable is represented by a unique node and every nonvariable type occurrence in C corresponds to a unique node in the graph. Furthermore, subtyping constraints in C are modeled by *flow edges* and equational constraints by *equivalence edges*.[3] For simplicity we shall denote nodes by object types, even though there may be several nodes for any given nonvariable type.

Given a flow graph G, we say A and B are equivalent in G and write $A \sim_G B$ or simply $A \sim B$, if A reaches B in G along equivalence edges only.

We write $src(A)$ for the set of object constructor labeled nodes that reach A along flow edges and equivalence edges; and $snk(A)$ for the set of object constructor labeled nodes that are reachable along flow edges and equivalence edges from A. Here flow edges can be traversed in forward direction only, but equivalence edges can be traversed in either direction.

Our algorithm is presented in Figure 4. It checks whether a given set of constraints C is solvable with recursive types or not. It assumes that all constraints in C are of one of the following forms:

1. $X \leq [l : Y]$
2. $X \leq Y$
3. $X = [l_i : Y_i]_{i \in \mathcal{I}}$
4. $X = Y$

Note that after Phase I the algorithm adds only equivalence edges. We shall see that the algorithm computes all derivable logical implications of an initial constraint system if it terminates without failure. If the algorithm terminates with failure then the original constraint system has no solution.

2.5 Correctness of algorithm

Let us write $C^{\leq}$ for all constraints P such that $C \vdash P$ is derivable *without* use of any of the rules (DECOMP$^{\leq}$), (ABSURD$^{\leq}$), (UNIQLAB). We define $\mathcal{F}(C)$ to be the set of constraints P with a derivation of $C \vdash P$ that uses any of the rules (DECOMP$^{\leq}$), (ABSURD$^{\leq}$), (UNIQLAB) only as the very last step, if at all. Recall that $C^* = \{P \mid C \vdash P\}$.

[3]Note that there are three kinds of edges in flow graphs: tree edges, flow edges and equivalence edges.

Initialization. 1. Initialize the *workset of constraints* W to C.

2. Build the term graph representing all the types in C.

Phase I (Process all subtyping constraints)

3. For all constraints in W of the form $X \leq [l_j : Y]$, remove $X \leq [l_j : Y]$ from W and add a flow edge from X to the node representing $[l_j : Y]$.
4. For all constraints in W of the form $X \leq Y$, remove $X \leq Y$ from W and add a flow edge from X to Y.
5. For all X occuring in C do:
   - For all $[l_i : Y], [l_j : Y'] \in snk(X)$, if $i = j$ then add $Y = Y'$ to W.

Phase II (Process all equational constraints)

While W is nonempty do:

6. Extract a constraint $A = B$ from W and delete it from W.
7. If $A \sim B$ then jump to the previous step. Otherwise continue with the following step.
8. For each $[l_i : A_i]_{i \in \mathcal{I}} \in src(A)$ and $[l_j : B_j]_{j \in \mathcal{J}} \in snk(B)$ do:
   - if $\mathcal{J} \subseteq \mathcal{I}$ then add $A_j = B_j$ to W for all $j \in \mathcal{J}$, else abort with failure.
9. For each $[l_j : B_j]_{j \in \mathcal{J}} \in src(B)$ and $[l_i : A_i]_{i \in \mathcal{I}} \in snk(A)$ do:
   - if $\mathcal{I} \subseteq \mathcal{J}$ then add $B_i = A_i$ to W for all $i \in \mathcal{I}$, else abort with failure.
10. For each $[l_i : A_i]_{i \in \mathcal{I}} \in snk(A)$ and $[l_j : B_j]_{j \in \mathcal{J}} \in snk(B)$ do:
    - For all $k \in \mathcal{I} \cap \mathcal{J}$ add $A_k = B_k$ to W.
11. Add an equivalence edge between A and B and continue with the while-loop.

Figure 4: Closure algorithm

Proposition 2.5 *C^* is the least set of constraints C' such that $\mathcal{F}(C') \cup C \subseteq C'$.*

PROOF Let C' be any set of constraints such that $\mathcal{F}(C') \cup C \subseteq C'$. It follows by rule induction on Figure 3 that $C^* \subseteq C'$.

In the other direction, note that $\mathcal{F}$ is monotonic *w.r.t.* the subset ordering. Thus for every C there exists a least C' such that $\mathcal{F}(C') \cup C \subseteq C'$. Furthermore, $C' = \bigcup_{i=0,1\ldots} \mathcal{F}^i(C)$. Now we can prove by induction on i that for every $P \in \mathcal{F}^i(C)$ there is a derivation for $C \vdash P$. □

The operator $\mathcal{F}$ gives us a general way of computing C^* using *fixed point iteration* [CP89]. We only discuss the outline of the correctness argument for our algorithm. For all nodes A the algorithm satisfies the following invariants after Phase I:

1. For all $[l_i : A_i]_{i\in\mathcal{I}} \in src(A)$ and $[l_j : B_j]_{j\in\mathcal{J}} \in snk(A)$ we have that $\mathcal{J} \subseteq \mathcal{I}$ and for all $j \in \mathcal{J}$, $A_j \sim B_j$ or $(A_j = B_j) \in W$.

2. For all $[l_i : A_i]_{i\in\mathcal{I}}, [l_j : B_j]_{j\in\mathcal{J}} \in snk(A)$ we have for all $k \in \mathcal{I} \cap \mathcal{J}$ that $A_k \sim B_k$ or $(A_k = B_k) \in W$.

Phase II preserves these invariants such that, if the algorithm terminates without failure, we have:

1. For all $[l_i : A_i]_{i\in\mathcal{I}} \in src(A)$ and $[l_j : B_j]_{j\in\mathcal{J}} \in snk(A)$ we have that $\mathcal{J} \subseteq \mathcal{I}$ and for all $j \in \mathcal{J}$, $A_j \sim B_j$.

2. For all $[l_i : A_i]_{i\in\mathcal{I}}, [l_j : B_j]_{j\in\mathcal{J}} \in snk(A)$ we have for all $k \in \mathcal{I} \cap \mathcal{J}$ that $A_k \sim B_k$.

Let C_t be the constraints represented by the flow edges and equivalence edges in the graph after termination. Then the above conditions guarantee that $C \cup \mathcal{F}(C_t^{\leq}) \subseteq C_t^{\leq}$. By Proposition 2.5 this implies that $C^* \subseteq C_t^{\leq}$. Since the algorithm has terminated without failure we have $\bot \notin C_t^{\leq}$ and thus $\bot \notin C^*$. By Lemma 2.4 it follows that C is solvable. If, on the other hand, the algorithm terminates with failure, then it must be that $C \vdash \bot$ is derivable since every constraint in the workset W and in the flow graph is derivable. Consequently, by Lemma 2.4 in this case C is unsolvable.

Theorem 2.6 *Given constraints C_a generated for an object expression a. If the algorithm of Section 2.4 terminates with success then a is typable in $\boldsymbol{Ob}_{1<:\mu}$. If it terminates with failure then a is untypable in $\boldsymbol{Ob}_{1<:\mu}$.*

By adding a circularity check corresponding to Rule (CYCLE) after the closure algorithm of Section 2.4 has terminated successfully, we obtain a correct algorithm for $\mathbf{Ob}_{1<:}$.

2.6 Analysis of algorithm

Given an object expression a of size n it is easy to see that the term graph representing all the types in C_a has $O(n)$ nodes and that C_a contains $O(n)$ constraints. We shall now investigate the total execution time taken by each one of steps 1–11 in Figure 4.

Step 1. Initializing W takes $O(n)$ time since C is assumed to be of size $O(n)$.

Step 2. Building the term graph for C takes time $O(n)$.

Steps 3 and 4. Adding the subtyping constraints of W to the graph and removing them from W takes time $O(n)$.

Step 5. For each X the set $snk(X)$ can be computed in time $O(n)$ using depth-first search (or any other linear-time search algorithm for that matter). Furthermore, using an array of length $O(n)$ the set of equational constraints $Y = Y'$ such that $[l_i : Y] \in snk(X)$ and $[l_i : Y'] \in snk(X)$ for some i can be computed in time $O(n)$. Since there are $O(n)$ variables in C, step 5 takes a total of $O(n^2)$ time.

Step 6. Extracting a constraint $A = B$ from W and deleting it can be done in constant time. Step 1 and Step 5 together insert a total of $O(n^2)$ equational constraints into W. As we shall see below, steps 8–10 also insert $O(n^2)$ into W. Thus this step takes a total of $O(n^2)$ time.

Step 7. Using a union/find data structure with path compression and union by weight or rank [Tar83] to represent the equivalence relation defined by the equivalence edges it takes 2 find operations and a pointer equality to determine whether $A \sim B$. Since this test is executed $O(n^2)$ times the accumulated cost of this step is $O(n^2\alpha(n^2, n)) = O(n^2)$.

Steps 8–10. For given A and B the algorithm in Figure 5 implements Step 8. It executes in time $O(n)$. One way to see this is to note that there is a total of $O(n)$ field name/node pairs in $src(A)$ and $snk(B)$ combined, and the total running time is proportional to the number of field name/node pairs. Similarly for Steps 9 and 10. Since each of steps 8–10 is executed at most $O(n)$ times the total cost attributable to

Let Arr be an array of size n.

1. For $i \in 1 \ldots n$ set Arr(i) to 0 (0 denotes an undefined node).
2. Set Count to 0. (Count counts the number of different l_j that occur in some $[l_j : B_j]_{j \in \mathcal{J}} \in snk(B)$.)
3. For all $[l_j : B_j]_{j \in \mathcal{J}} \in snk(B)$ do:
   - For all $j \in \mathcal{J}$, if Arr$(j) = 0$ then set Arr(j) to B_j and add 1 to Count.
4. For all $[l_i : A_i]_{i \in \mathcal{I}} \in src(A)$ do:
   - Initialize Count$'$ to Count.
   - For all $i \in \mathcal{I}$, if Arr$(i) \neq 0$ then add $A_i =$ Arr(i) to W and decrease Count$'$ by 1.
   - If Count$' \neq 0$ then terminate with failure.

Figure 5: Algorithm for computing Step 8.

Steps 8–10 is $O(n^2)$. Note also that Steps 8–10 add $O(n^2)$ constraints to W.

Step 11. Adding an equivalence edge is done be a single union operation, which takes amortized time $O(\alpha(n^2, n)) = O(1)$. Since nodes can be contracted in this fashion at most $O(n)$ times the accumulated cost attributed to this step is $O(n)$.

We have shown that the Algorithm in Figure 4 terminates in time $O(n^2)$ given a constraint system C_a generated from an object expression a of size n. What saves us from ending up with a worse running time is that we compute "reach" sets — and thus do transitive closure — but only on *sparse* graphs since we *do not add a single flow edge* during the algorithm, only *equivalence edges*. This is in contrast to Palsberg [Pal95] who, instead of contracting nodes as we do in Step 11, inserts a pair of flow edges and relies on general dynamic transitive closure as the backbone of his algorithm.

For system $\mathbf{Ob}_{1<:}$ we also need to check whether the resulting graph, after termination, has a cycle. This can be done using depth first search.

$$
\begin{array}{lll}
(\mathrm{VAL})^n & E, x : A, E'' \vdash x : A' \quad (x \notin \mathit{domain}\ E'') & A = A' \\[2ex]
(\mathrm{OBJ})^{n'} & \dfrac{E, x_i : A \vdash b_i : B_i' \qquad (\forall i \in \mathcal{I})}{E \vdash [l_i = \varsigma(x_i)b_i]_{i\in\mathcal{I}} : A'} & \left\{ \begin{array}{l} B_i' = B_i, i \in \mathcal{I} \\ A = [l_i : B_i]_{i\in\mathcal{I}} \\ A' = A \end{array} \right\} \\[4ex]
(\mathrm{SEL})^n & \dfrac{E \vdash a : A}{E \vdash a.l_j : B} & A \leq [l_j : B] \\[4ex]
(\mathrm{UPD})^{n'} & \dfrac{E \vdash a : A \qquad E, x : A' \vdash b : B'}{E \vdash a.l_j \Leftarrow \varsigma(x)b : A''} & \left\{ \begin{array}{l} A = A' \\ A = A'' \\ B' = B \\ A \leq [l_j : B] \end{array} \right\}
\end{array}
$$

Figure 6: Normalized type inference rules (without subtyping)

Theorem 2.7 *Given object expression a of size n it can be decided in time $O(n^2)$ whether a is typable in system $\boldsymbol{Ob}_{1<:}$ or $\boldsymbol{Ob}_{1<:\mu}$.*

3 Type inference for $\mathbf{Ob}_1$ and $\mathbf{Ob}_{1\mu}$

The above development can be repeated for object inference without subtyping. In this case we replace the subtyping constraints $B_i' \leq B_i, i \in 1 \ldots n$ in Rule $(\mathrm{OBJ})^n$ by $B_i' = B_i$ and $B' \leq B$ in Rule $(\mathrm{UPD})^n$ by $B' = B$. We thus arrive at the normalized inference rules presented in Figure 6. We write $E \vdash_{\mathbf{Ob}_{1\mu}{}^n} a : A$ or $E \vdash_{\mathbf{Ob}_1{}^n} a : A$ if $E \vdash a : A$ is derivable using the normalized rules of Figure 6 with, respectively without recursive types.

Lemma 3.1 *Let O be $\boldsymbol{Ob}_1$ or $\boldsymbol{Ob}_{1\mu}$. If $E \vdash_O a : A$ and $E' \leq E$ then there exists A' such that $E' \vdash_{O^n} e : A'$ and $A' \leq A$.*

Lemma 3.2 *Given object expression a there is a set of equational and subtyping constraints C_a such that C_a is solvable with/without recursive types if and only if a is typable in $\boldsymbol{Ob}_{1\mu}/\boldsymbol{Ob}_1$.*

Furthermore, C_a can be computed from a in linear time.

These lemmas guarantee that the algorithm in Section 2.4 is applicable and provides an $O(n^2)$ time decision procedure for both $\mathbf{Ob}_{1\mu}$ without the cycle check and for $\mathbf{Ob}_1$ with the cycle check. We shall see, however, that the absence of subtyping lets us design asymptotically better algorithms.

3.1 Type inference for $\mathbf{Ob}_{1\mu}$

Consider the constraints in C_a generated for an object expression according to Lemma 3.2. They fall into three categories:

1. $X = Y$ where X and Y are type variables;
2. $X = [l_i : X_i]_{i \in \mathcal{I}}$ where X, X_i are type variables;
3. $X \leq [l : Y]$ where X, Y are type variables.

Note that there are *no* constraints of the form $X \leq Y$ where both X and Y are type variables — this is what we shall exploit in the following.

Consider the algorithm 2.4. We shall enhance it such that it maintains two object types, $SRC(A)$ and $SNK(A)$, with each node A in the flow graph in such a fashion that the following invariants are satisfied during execution of the algorithm:

1. If $SRC(A) = 0$ (undefined type) then $src(A) = \emptyset$.
2. If $SRC(A) = [l_i : A_i]_{i \in \mathcal{I}}$ then for all $[l_j : B_j]_{j \in \mathcal{J}} \in src(A)$ we have $\mathcal{I} = \mathcal{J}$ and for all $i \in \mathcal{I}$, either $A_i \sim B_i$ or $(A_i = B_i) \in W$.
3. If $SNK(A) = [l_i : A_i]_{i \in \mathcal{I}}$ then for all $[l_j : B_j]_{j \in \mathcal{J}} \in snk(A)$ we have $\mathcal{J} \subseteq \mathcal{I}$ and for all $j \in \mathcal{J}$, either $A_j \sim B_j$ or $(A_j = B_j) \in W$.
4. If $SNK(A) = [l_i : A_i]_{i \in \mathcal{I}}$ then for all $i \in \mathcal{I}$ there exists $[l_j : B_j]_{j \in \mathcal{J}}$ such that $i \in \mathcal{J}$.

Then, instead of using $src(A), src(B), snk(A), snk(B)$ in Steps 8–10 we use $SRC(A), SRC(B), SNK(A), SNK(B)$, respectively. By representing $\mathcal{I}$ in type $[l_i : A_i]_{i \in \mathcal{I}}$ as a balanced search tree we check whether $j \in \mathcal{I}$ and retrieve A_j in time $O(\log |\mathcal{I}|) = O(\log n)$.

Recall that C_a for $\mathbf{Ob}_1/\mathbf{Ob}_{1\mu}$ contains no constraints of the form $X \leq Y$. Thus Step 4 in the algorithm is unnecessary, and, as consequence of this, Step 5 can be implemented in time $O(n)$. Steps 8 and 9 each take cumulative time $O(n \log n)$. Consider for example execution of Step 8 for a particular A and B. If $SRC(A) = 0$ then this step is finished immediately.

If $SRC(A) = [l_i : A_i]_{i \in \mathcal{I}}$ and $SNK(B) = [l_j : B_j]_{j \in \mathcal{J}}$ then we check, for each $j \in \mathcal{J}$, that $j \in \mathcal{I}$ and add $A_j = B_j$ to W. Since every B_j is added to W at most once in this fashion, the total cost of Step 8 is $O(n \log n)$. Step 10 is the bottleneck in our algorithm. In Step 10 the smaller of $SNK(A), SNK(B)$ is "merged" into the other to form $SNK(A)$ after contracting B into A. This guarantees that any given $[l : Y]$ in the original constraints is merged at most $O(\log n)$ times. Since each such individual merge takes time $O(\log n)$ and Step 10 is executed $O(n)$ times we have a total cost of $O(n \log^2 n)$ for Step 10. This gives us an $O(n \log^2 n)$ time algorithm for $\mathbf{Ob}_{1\mu}$.

Theorem 3.3 *Given object expression a of size n it can be decided time $O(n \log^2 n)$ whether a is typable in system $\boldsymbol{Ob}_{1\mu}$.*

3.2 Type inference of $\mathbf{Ob}_1$

The above algorithm can be further improved for object type inference without recursive types; that is, for system $\mathbf{Ob}_1$. In this case we process the equational constraints in Phase II in a special *depth-first search* order. Let us add all subtyping and equational constraints to the term graph for C_a as flow edges and equivalence edges, respectively. We now do a depth-first search of the flow graph where:

- tree edges are traversed in the *reverse* direction (from child to parent),
- subtyping edges are traversed in forward *and* reverse direction, and
- equivalence edges are traversed in either direction.

From a given node we follow all tree edges leading to a parent of the node *before* any other edges — in particular equivalence edges — are traversed. This is critical since during the search new equivalence edges are inserted in the graph, corresponding to Steps 8–10 in Figure 4. The order of edge processing guarantees that, once a node is *finished* no equivalence edges incident to that node will ever be added. In other words, once a node is finished — it has no more unvisited incident edges — it is truly finished even though additional equivalence edges may be inserted other places into the flow graph *after* the node has been finished.

The above-sketched depth-first search traversal induces a particular order in which the equational constraints in the workset W are processed. An equivalence edge in the flow graph is, at any given point of time, either *unprocessed, active* or *processed.* Initially all equivalence edges are unprocessed. Once a node is *finished* all its incident equivalence edges are marked active. When the depth-first search returns from traversing a tree edge (that is, it returns to the child node after having finished its parent node), all currently active equivalence edges are processed as a *batch* and then marked processed. These equivalence edges correspond to a set of equational constraints. An important property is that at any point in time the active unprocessed equivalence edges connect all the nodes incident to them. Using a global array we can process all the active equivalence edges in time proportional to the number of active equivalence edges plus incident tree edges. The number of new equivalence edges introduced in the graph is bounded by the number of incident tree edges. The *total* running time for processing *all* equivalence edges is thus $O(n)$.

An important point of the algorithm is that whenever a discovered node (node on the depth-first search stack) is visited that is "at least one tree edge away" then we can terminate with failure due to the (CYCLE) rule.

Theorem 3.4 *Given object expression a of size n it can be decided in time $O(n)$ whether a is typable in system $\boldsymbol{Ob}_1$.*

4 Related work

Palsberg [Pal95] has provided the first inference algorithms for the four calculi covered in this paper. He gives a careful presentation of the calculi and their type inference problems and thorough proofs of his results. Even though there are apparent differences in his and our work, at a deeper level our flow graph formalization is very close to his AC-graphs. The main difference — and the main message this paper is trying to convey — is that *equivalence relations are more efficiently maintainable than arbitrary binary relations*. Palsberg's algorithms do not exploit the invariance property of object subtyping, which gives *equivalences*. This is critical if one wants to "break" through the notorious n^3 bottleneck incurred by employing dynamic transitive closure [LP89, Yel93] or similar methods.

The four object inference systems are interesting in that they share the invariance property with *simple value flow analysis*, which permits computation in almost-linear time [Hen92]. Yet, there are two properties that make object type inference more difficult than simple value flow:

1. the invariance property is not as strong as in simple value flow analysis; in particular, from $[l_i : B_i] \leq A, [l_i : B'_i] \leq A$ we *cannot* conclude that $B_i = B'_i$.

2. the number of component types of an object type is not bounded by a program-independent constant.

It remains to be seen whether our bounds can be improved. Our bound for $\mathbf{Ob}_{1\mu}$ seems to be a particularly likely candidate for improvement.

Object-oriented type inference with covariant record subtyping (or function types with contra/co-variant subtyping) is not immediately amenable to our techniques since the best algorithmic techniques known at present require full-blown dynamic transitive closure or similar techniques with $\Theta(n^3)$ time complexity.

5 Conclusion and future work

We have shown how the invariance property of Abadi and Cardelli's object type systems can be exploited to design quadratic or subquadratic-time type inference algorithms. This improves the time bounds given by Palsberg for all four calculi studied.

The normalized constraint systems give rise to a principal typing property for object expressions, using subtype qualified type schemes with relatively few subtype qualifications. This is a topic for future work.

Acknowledgments

This work was inspired by the lectures of Martín Abadi, Luca Cardelli and Jens Palsberg at the ACM State of the Art Summer School on Functional and Object-Oriented Programming in Sobotka, Poland, September 8-14, 1996.

Thanks to Luca Cardelli for helping uncover a terrible oversight in an optimistic early approach, and to Mads Tofte for his probing questions on some aspects of the algorithms presented here. Jens Palsberg has been helpful by providing a crystal-clear presentation of his object inference algorithms during the Summer School.

References

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996. ISBN 0-387-94775-2.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, 1990. ISBN 0-262-03141-8 (MIT Press) and ISBN 0-07-013143-0 (McGraw-Hill).

[CP89] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1989.

[CP91] J. Cai and R. Paige. Look ma, no hashing, and no arrays neither. In January, editor, *Proc. 18th Annual ACM Symp. on Principles of Programming Languages (POPL), Orlando, Florida*, pages 143–154, 1991.

[CP95] Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science (TCS)*, 145(1-2), July 1995.

[Hen92] Fritz Henglein. Simple closure analysis. DIKU Semantics Report D-193, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, March 1992.

[LP89] J. La Poutré. New techniques for the union-find problem. Technical Report RUU-CS-89-19, Utrecht University, August 1989.

[Pal95] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.

[Tar83] R. Tarjan. *Data Structures and Network Flow Algorithms*, volume CMBS 44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[Yel93] Daniel Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993. Also available as IBM T.J. Watson Research Center Research Report.