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

Fritz Henglein
DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, Email: henglein@diku.dk

Abadi and Cardelli [1] present a series of type systems for their object calculi, four of which are first-order. Palsberg [22] has shown how typability in each one of these systems can be decided in time $O(n^3)$ and space $O(n^2)$, where n is the size of an untyped object expression, using an algorithm based on dynamic transitive closure.

In this paper we 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	$O(n \log^2 n)$	$O(n^2)$

Furthermore, our algorithms improve the space complexity from $O(n^2)$ to $O(n)$ in each case.

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

1. Introduction

Abadi and Cardelli have developed a theory of objects based on their ς-calculus, a simple calculus of pure objects. They model an object as a set of named methods, written $[l_1 = \varsigma(x_1)b_1, \ldots, l_k = \varsigma(x_k)b_k]$. Here, l_i denotes a method name (label) and $\varsigma(x_i)b_i$ the associated method. The variable x_i is bound by ς in the body b_i of the method. It denotes the host object, which contains the method; i.e., it denotes `self` and is thus called the *`self`-variable*. Methods are required to have unique names, and their order in an object is irrelevant; that is, objects are identified up to reordering of methods.

A method of an object a can be invoked by selecting its method name: $a.l$. Here a is the receiver object and l the name of the invoked method. What makes the calculus both simple and powerful is the operation of method update. Given object a with method named l it is possible to update a by assigning another method to l, written as $a.l \Leftarrow \varsigma(x)b$. In this fashion there is no need to distinguish between (updatable) fields and (nonupdatable) methods, as in most class-based languages, since fields can be modeled by methods whose bodies do not refer to the `self`-variable.

Variables, object formation, method invocation and method update define the complete language of untyped object expressions in the ς-calculus:

$$a, b ::= x \mid [l_i = \varsigma(x_i)b_i]_{i=1\ldots k} \mid a.l \mid a.l \Leftarrow \varsigma(x)b$$

The basic operational semantics is given by the following reduction rules:

$$[l_i = \varsigma(x_i)b_i]_{i=1\ldots k}.l_j \longrightarrow b_j\{x_j \mapsto [l_i = \varsigma(x_i)b_i]_{i=1\ldots k}\}, \text{ if } 1 \le j \le k$$

$$[l_i = \varsigma(x_i)b_i]_{i=1\ldots k}.l_j \Leftarrow \varsigma(x)b \longrightarrow [l_i = \varsigma(x_i)b_i, l_j = \varsigma(x)b]_{i=1\ldots j-1,j+1\ldots k}, \text{ if } 1 \le j \le k$$

The first rule expresses that a method invocation reduces to the body of the invoked method where the ς-bound `self`-variable is replaced by the receiver object in the invocation. The second rule expresses that a method update is literally performed by removing the existing method and splicing the replacing method into the object. (Note that method update requires that the updated method exists in the first place.)

In this paper we show how type inference can be performed efficiently for a number of first-order object type systems, which all share the property that they have type invariance of components in object subtyping. See Abadi and Cardelli's book [1] and Palsberg [22] for a motivation of the relevance of the calculi, their typing systems and the practical importance of efficient and easily implementable type inference.

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*. An object type is *proper* if it is not a type variable. The l_i in object types are pairwise distinct *method names*, and the order of *components* $l_i : A_i$ in an object type $[l_i : A_i]_{i=1\ldots n}$ is unimportant. *Recursive (object) types*

nite 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 components and infinite unfolding of recursive type occurrences. Here we deviate from Abadi and Cardelli [1], but follow Palsberg [22]. We treat the isomorphism between $A\{X \mapsto \mu X.A\}$ and $\mu X.A$ as *implicit* operations or even as an identity, instead of requiring the explicit *fold* and *unfold* constructs [1, Section 9] for mapping back and forth between the two types. For example, the following three types are equal to each other.

$$\mu X.[l_1 : [l_1 : X, l_2 : []], l_2 : []]$$
$$\mu Y.[l_1 : Y, l_2 : []]$$
$$[l_1 : \mu Y.[l_1 : Y, l_2 : []], l_2 : []]$$

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.

Henceforth we let x, y range over variables, a, b over object expressions, l over method names, e, i, j, k, m, n, t, v over nonnegative integers, X, Y, α, β, γ, δ, ϵ, μ, ν over type variables, A, B over object types, and E over type environments. This conventions extends to sub- and superscripted variants of these metavariables.

Throughout this paper we let x, y range over variables, a, b over object expressions, l over method names and e, i, j, k, m, n, t, v over nonnegative integers. This convention extends to sub- and superscripted versions of these metavariables.

1.2. Object type systems

The object calculi we shall investigate are:

- $\mathbf{Ob}_1$[1, Section 7, p. 83],
- $\mathbf{Ob}_{1\mu}$[1, Section 9, p. 114],
- $\mathbf{Ob}_{1<:}$[1, Section 8, p. 94], and
- $\mathbf{Ob}_{1<:\mu}$[1, Section 9, p. 117].

The inference rules for these systems are given in Figure 1.[1] Table 0 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. Note also that regular infinite trees denoted by recursive types are only allowed in the type systems that explicitly permit them. See Table 0. 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 $E \vdash_O a : A$.

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 1. *Let $\boldsymbol{Ob}^n_{1<:}$ (without recursive types) and $\boldsymbol{Ob}^n_{1<:\mu}$ (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 $E \vdash_{\boldsymbol{Ob}^n_{1<:}} e : A'$ and $A' \leq A$.*
2. *$E \vdash_{\boldsymbol{Ob}_{1<:\mu}} e : A$ if and only if there exists A' such that $E \vdash_{\boldsymbol{Ob}^n_{1<:\mu}} e : A'$ 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 $(\text{UPD})^n$ 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. *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, $E' = E'_1, x : A', E'_2$

TABLE 1. Type system definitions

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

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

(OBJ) $$\frac{E, x_i : [l_i : B_i]_{i=1\ldots k} \vdash b_i : B_i \qquad (\forall i : 1 \le i \le k)}{E \vdash [l_i = \varsigma(x_i) b_i]_{i=1\ldots k} : [l_i : B_i]_{i=1\ldots k}}$$

(SEL) $$\frac{E \vdash a : [l_i : B_i]_{i=1\ldots k}}{E \vdash a.l_j : B_j} \qquad (1 \le j \le k)$$

(UPD) $$\frac{E \vdash a : [l_i : B_i]_{i=1\ldots k} \qquad E, x : [l_i : B_i]_{i=1\ldots k} \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x) b : [l_i : B_i]_{i=1\ldots k}} \qquad (1 \le j \le k)$$

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

(REFL) $$\vdash A \le A$$

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

(OBSUB) $$\vdash [l_j : B_j]_{j=1\ldots k+m} \le [l_i : B_i]_{i=1\ldots k}$$

FIG. 1. Object type inference rules

$(\text{VAL})^n$ $$E, x : A, E'' \vdash x : A' \qquad (x \notin \mathit{domain}\, E'') \qquad A = A'$$

$(\text{OBJ})^n$ $$\frac{E, x_i : A \vdash b_i : B_i' \qquad (\forall i : 1 \le i \le k)}{E \vdash [l_i = \varsigma(x_i) b_i]_{i=1\ldots k} : A'} \qquad \left\{ \begin{array}{l} \{B_i' \le B_i\}_{1 \le i \le k} \\ A = [l_i : B_i]_{i=1\ldots k} \\ A' = A \end{array} \right\}$$

$(\text{SEL})^n$ $$\frac{E \vdash a : A}{E \vdash a.l : B} \qquad A \le [l : B]$$

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

FIG. 2. Normalized type inference rules

with $E_1' \le E_1, A' \le A$ and $E_2' \le E_2$. By Rule $(\text{VAL})^n$ we derive $E' \vdash_{O^n} x : A'$ and we are done since $A' \le A$.

Rule (OBJ): Assume $E \vdash_O [l_i = \varsigma(x_i) b_i]_{i=1\ldots k} : A$ is derived by Rule (OBJ) from $E, x_i : A \vdash_O b_i : B_i$ for all i such that $1 \le i \le k$. Let $E' \le E$. Then $E', x_i : A \le E, x_i : A$, and by induction hypothesis we have $E', x_i : A \vdash_{O^n} B_i'$ with $B_i' \le B_i$ for $1 \le i \le k$. Thus we can apply Rule $(\text{OBJ})^n$ to derive $E' \vdash_{O^n} [l_i = \varsigma(x_i) b_i]_{i=1\ldots k} : A$.

Rule (SEL): Assume $E \vdash_O a.l : B$ is derived from $E \vdash_O a : A$ by Rule (SEL). In this case $A \le [l : B]$ since A must contain the component $l : B$. Let $E' \le E$. By induction hypothesis we have $E' \vdash_{O^n} a : A'$ with $A' \le A$. Since $\le$ is transitively closed we

and we obtain $E' \vdash_{O^n} a.l : B$.

Rule (UPD): Assume $E \vdash_O a.l \Leftarrow \varsigma(x)b : A$ is derived from $E \vdash_O a : A$ and $E, x : A \vdash_O b : B$ by Rule (UPD). Thus $A \leq [l : B]$ since A must contain the component $l : B$. 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 $E', x : A' \vdash B'$ for some $B' \leq B$. Note that we have $A' \leq [l : B]$ since $A' \leq A \leq [l : B]$, and $B' \leq B$. Thus Rule $(\text{UPD})^n$ is applicable and yields $E' \vdash_{O^n} a.l \Leftarrow \varsigma(x)b : A'$. 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. ∎

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 metavariables in Figure 2.

Constraints P are generated by the following grammar:

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

We call $A = B$ an *equational constraint* and $A \leq B$ a *subtyping constraint*. The special constraint $\bot$ denotes the *absurd constraint*, which is, by definition, unsolvable.

A *valuation* ρ is a mapping from type variables to object types, which is canonically extended to object types. We say ρ is *over finite 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, written $\rho \models A = B$, 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, written $\rho \models A \leq B$, 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 components or $\rho(B)$. A valuation ρ solves a set of constraints C, written $\rho \models C$, if it solves each individual constraint in it. In any of the above cases we say that ρ is a *solution* of $A = B, A \leq B$, respectively C. If ρ is over finite types, we say it is a *finite solution*. Finally, a set of constraints C *entails* a constraint P, written $C \models P$, if all solutions of C also solve P. We write $C \models_f P$ if all *finite* solutions of C also solve P. (Note that $C \models P$ implies $C \models_f P$, but not necessarily *vice versa*.)

Clearly, an object expression a is typable in $\mathbf{Ob}_{1<:\mu}$ if and only if the constraints C_a its derivation skeleton generates are *solvable*; that is, if and only if there is a valuation ρ (finite or not) such that $\rho \models C$. Similarly, a is typable in $\mathbf{Ob}_{1<:}$ if and only if C_a has a *finite* solution. This is summarized in the following lemma:

Lemma 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 method names and variables can be done in linear time [3, 2]. In particular, given an object expression of length n (measured in bits) we may assume that variables and method 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 generation: Example

Consider the object term

$$\mathit{fix} \equiv \begin{array}{rcl}[\mathit{arg} & = & \varsigma(x)x.\mathit{arg} \\ \mathit{val} & = & \varsigma(y)((y.\mathit{arg}).\mathit{arg} \Leftarrow \varsigma(z)y.\mathit{val}).\mathit{val}]\end{array}$$

(see [1, Section 6.4]). Its syntax tree, including unique type variables labeling each syntax node, is depicted in Figure 3. The constraints C_{fix} for *fix* are shown in Figure 4.

2.4. Constraint closure

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

1. Design a proof system for judgements $C \vdash \bot$ that is *contradictively complete*: $C \vdash \bot$ if and only if $C \models \bot$. (Note that C is unsolvable if and only if $C \models \bot$.) This guarantees that C is unsolvable if and only if $C \vdash \bot$ is derivable.

2. For given C, compute the set of *all* P such that $C \vdash P$. If $\bot$ is one of these constraints then C is unsolvable. If not, then C is solvable.

Inference rules of the proof system can be thought of as rewritings on constraint systems that 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.

Figures 5 and 6 provide inference rules for judgements of the form $C \vdash P$. We write $C \vdash P$ (as a statement) if the judgement $C \vdash P$ is derivable using the rules of Figure 5. We write $C \vdash_f P$ if $C \vdash P$ is derivable using the rules of both Figures 5 and 6.

Lemma 4. 1. *$C \models \bot$ if and only if $C \vdash \bot$.*

2. *$C \models_f \bot$ if and only if $C \vdash_f \bot$.*

Proof. (1) It is easy to check by rule induction that $C \vdash P$ implies $C \models P$ and thus, in particular, $C \vdash \bot$ implies $C \models \bot$.

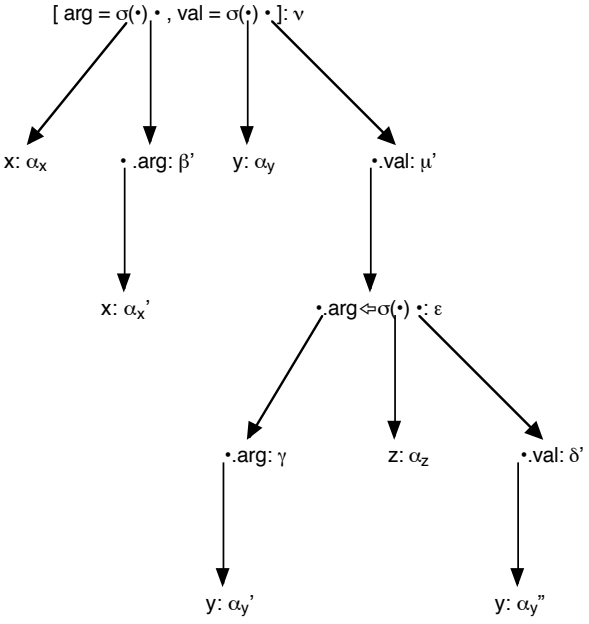

FIG. 3. Type variable annotated syntax tree for *fix*

$$
\begin{array}{rclcrcl}
\alpha_x & = & \alpha'_x & & \alpha'_x & \leq & [arg : \beta'] \\
\alpha_y & = & \alpha'_y & & \alpha'_y & \leq & [arg : \gamma] \\
\alpha_y & = & \alpha''_y & & \alpha''_y & \leq & [val : \delta'] \\
\gamma & = & \alpha_z & & \gamma & = & \epsilon \\
\delta' & \leq & \delta & & \gamma & \leq & [arg : \delta] \\
\epsilon & \leq & [val : \mu'] & & \beta' & \leq & \beta \\
\alpha_x & = & \nu & & \nu & = & [arg : \beta, val : \mu] \\
\mu' & \leq & \mu & & \alpha_x & = & \alpha_y
\end{array}
$$

FIG. 4. Constraints for *fix*

Conversely, assume that we cannot derive $C \vdash \bot$. Then we can define a solution ρ of C, using the general construction pioneered by Kozen, Palsberg and Schwartzbach [16]. Since Palsberg has already used this construction in object type inference [22], we only give a brief presentation here, without elaborating the correctness arguments.

An *object type automaton structure* is a pair $(\mathcal{S}, \Delta)$ where $\mathcal{S}$ is a finite set of *states* and Δ is a partial *transition function* that maps pairs consisting of a state and a method name to a state. Every state S in $(\mathcal{S}, \Delta)$ denotes a proper object type that has a component with method name l if and only if $\Delta(S, l)$ is defined. If $\Delta(S, l)$ is defined and $\Delta(S, l) = S'$, then the component type associated with l is (the object type denoted by) S'. In this fashion every state in an object type automaton structure denotes a unique proper object type.

Given a set of constraints C we let $\mathcal{T}(C)$ be the set of all proper object types occurring in C. We now define the states $\mathcal{S}$ of an object type automaton structure for C to be the powerset of $\mathcal{T}(C)$: $\mathcal{S} = 2^{\mathcal{T}(C)}$. For the transition function we require a number of auxiliary definitions:

$$
\begin{array}{rcl}
\uparrow_C(A) & = & \{B \mid B \in \mathcal{T}(C) \text{ and } C \vdash A \leq B\} \\
\uparrow_C(S) & = & \cup_{A \in S}(\uparrow_C(A)) \\
S|_l & = & \{B \mid (\exists A)\, A \in S \text{ and } A = [l : B, \ldots]\}
\end{array}
$$

Here S ranges over subsets of $\mathcal{T}(C)$. Now we can define the transition function as follows:

$$
\Delta(S, l) = \begin{cases} \uparrow_C(S|_l), & \text{if } S_l \neq \emptyset \\ \text{undefined}, & \text{otherwise} \end{cases}
$$

The inference rules have the *subterm property*: if $C \vdash P$ then every object type that occurs in P occurs also in C. Because of this, Δ is well-defined: if $\Delta(S, l)$ is defined then it is a subset of $\mathcal{T}(C)$.

We now define a mapping from type variables to subsets of $\mathcal{T}(C)$:

$$
\rho(X) = \uparrow_C(X).
$$

Since every subset denotes an object type this defines a valuation. It can now be checked that $\rho \models C$.

(2) Rule (CYCLE) is sound over finite types. This can be shown by contraposition: Assume $C \not\models_f \bot$; that is, C is solvable over finite types. We claim that $C \not\models_f B_0 \leq [l_1 : B_1, \ldots], \ldots, B_{k-1} \leq [l_k : B_k, \ldots], B_k \leq [l_0 : B_0, \ldots]$. For this it is sufficient to show that $\{B_0 \leq [l_1 : B_1, \ldots], \ldots, B_{k-1} \leq [l_k : B_k, \ldots], B_k \leq [l_0 : B_0, \ldots]\}$ is unsolvable over finite types. Let $d(B)$ be the *depth* of an object type, defined by

$$
\begin{array}{rcl}
d(X) & = & 0 \\
d([l_i : A_i]_{i=1\ldots k}) & = & 1 + \max\limits_{i=1\ldots k} d(A_i)
\end{array}
$$

on finite types. It is easy to see that $d(A) > d(B)$ if $A \leq [l : B, \ldots]$. Assume ρ is a finite solution of

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

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

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

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

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

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

$$\text{(DECOMP}^{\leq}\text{)} \qquad \frac{C \vdash [l_i : B_i]_{i=1\ldots k+m} \leq [l_i : B'_i]_{i=1\ldots k}}{C \vdash B_j = B'_j} \qquad (1 \leq j \leq k)$$

$$\text{(ABSURD}^{\leq}\text{)} \qquad \frac{C \vdash [l_i : B_i]_{i=1\ldots k} \leq [l : B', l'_j : B'_j]_{j=1\ldots m}}{C \vdash \bot} \qquad (l \notin \{l_i\}_{i=1\ldots k})$$

$$\text{(UNIQLAB)} \qquad \frac{C \vdash A \leq [l : B, \ldots] \quad C \vdash A \leq [l : B', \ldots]}{C \vdash B = B'}$$

FIG. 5. Entailment rules for recursive and finite types

$$\text{(CYCLE)} \quad \frac{C \vdash B_0 \leq [l_1 : B_1, \ldots], \ldots, B_{k-1} \leq [l_k : B_k, \ldots], B_k \leq [l_0 : B_0, \ldots]}{C \vdash \bot}$$

FIG. 6. Cycle rule for finite types

$B_0 \leq [l_1 : B_1, \ldots], \ldots, B_{k-1} \leq [l_k : B_k, \ldots], B_k \leq [l_0 : B_0, \ldots]$. Then we have $d(\rho(B_0)) > d(\rho(B_1)) > \ldots > d(\rho(B_{k-1})) > d(\rho(B_k)) > d(\rho(B_0))$ and thus $d(\rho(B_0)) > d(\rho(B_0))$, which is impossible. Thus we can conclude that $B_0 \leq [l_1 : B_1, \ldots], \ldots, B_{k-1} \leq [l_k : B_k, \ldots], B_k \leq [l_0 : B_0, \ldots]$ has no finite solution.

Conversely, if $C \not\vdash_f \bot$, rule (CYCLE) guarantees that the states of the term automaton structure above can be topologically ordered, which guarantees that the valuation ρ is in fact a finite solution. ∎

2.5. *Closure algorithm*

We now show how to efficiently compute the logical implications $C^* = \{P : C \vdash P\}$ of a constraint set C generated according to Lemma 3. Recall that the inference system in Figure 5 has the subterm 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 $\Omega(n^3)$ and space $\Omega(n^2)$. In this section we present an asymptotically improved algorithm that executes in time $O(n^2)$ and space $O(n)$.

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 type constructor labeled nodes may have children that can be reached along *tree edges*, each of which is labeled by a method name. We represent a constraint set C by a term graph where every type variable is represented by a unique node and every proper object type occurrence in C corresponds to a unique node in the graph. Furthermore, subtyping constraints in C are modeled by (directed) *flow edges* and equational constraints by (undirected) *equivalence edges*.[3] For simplicity we shall denote nodes by object types, even though this is a potential source of ambiguity as 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 $A \sim_G B$, if A reaches B in G along equivalence edges only. For a set of equational constraints W we write $A \sim_{G \cup W} B$ if A reaches B in G extended with all the equational constraints in W viewed as equivalence edges.

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

Our algorithm is presented in Figure 7. It checks whether a given set of constraints C is solvable 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. $X = [l_i : Y_i]_{i=1\ldots k}$
4. $X = Y$

We shall see that the algorithm computes all derivable logical implications C^* of an initial constraint set C if it terminates without failure. If the algorithm terminates with failure then the original constraint system has no solution.

Phase I (Initialization)

1. Build the term graph G representing all the types in C.
2. Initialize W to the set of equational constraints in C.
3. Add all subtyping constraints in C as flow edges to G.
4. For all X and l occurring in C do:
   - If $snk(X)|_l = \{X_1, \ldots, X_k\}, k \geq 2$, then for all $i = 2 \ldots k$ do:
     - if $X_1 \not\sim_{G \cup W} X_i$ then add $X_1 = X_i$ to W.

Phase II (Process all equational constraints)
While W is nonempty do:

5. Extract a constraint $A = B$ from W and delete it from W.
6. If $A \sim_G B$ then jump to the previous step. Otherwise continue with the following step.
7. For each l such that, for some A', B', $[l : A', \ldots] \in snk(A)$ and $[l : B', \ldots] \in snk(B)$ and $A' \not\sim_{G \cup W} B'$ do:
   - add $A' = B'$ to W.
8. For each $[l_i : A_i]_{i=1\ldots k} \in src(A)$ and $[l'_j : B_j]_{j=1\ldots m} \in snk(B)$ do:
   - if $\{l'_j\}_{j=1\ldots m} \not\subseteq \{l_i\}_{i=1\ldots k}$ then abort with failure; else:
   - for each i, j such that $1 \leq i \leq k, 1 \leq j \leq m$, $l_i = l'_j$ and $A_i \not\sim_{G \cup W} A'_j$ do:
     - add an $A_i = B_j$ to W.
9. For each $[l_i : A_i]_{i=1\ldots k} \in snk(A)$ and $[l'_j : B_j]_{j=1\ldots m} \in src(B)$ do:
   - if $\{l_i\}_{i=1\ldots k} \not\subseteq \{l'_j\}_{j=1\ldots m}$ then abort with failure; else:
   - for each i, j such that $1 \leq i \leq k, 1 \leq j \leq m$, $l_i = l'_j$ and $A_i \not\sim_{G \cup W} A'_j$ do:
     - add an $A_i = B_j$ to W.
10. Add an equivalence edge between A and B to G.

FIG. 7. Closure algorithm

2.6. Correctness of algorithm

Let us write $C^{\leq}$ for the set of 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 $C^* = \{P \mid C \vdash P\}$.

Proposition 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 5 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 $C \vdash P$. ∎

Upon entry to Step 5 the algorithm satisfies the following invariants for all nodes A occurring in C:

- For all $[l_i : A_i]_{i=1\ldots k} \in src(A)$ and $[l'_j : B_j]_{j=1\ldots m} \in snk(A)$ we have that $\{l'_j\}_{j=1\ldots m} \subseteq \{l_i\}_{i=1\ldots k}$ and for all i, j such that $1 \leq i \leq k, 1 \leq j \leq m$, if $l_i = l'_j$ then $A_i \sim_{G \cup W} B_j$.
- For all $[l_i : A_i]_{i=1\ldots k}, [l'_j : B_j]_{j=1\ldots m} \in snk(A)$ we have for all i, j such that $1 \leq i \leq k, 1 \leq j \leq m$, if $l_i = l'_j$ then $A_i \sim_{G \cup W} B_j$.

Each iteration consisting of Steps 5-10 in Phase II preserves these invariants. Consequently, if the algorithm terminates without failure, we have for all nodes A occurring in C (since W is empty upon termination):

1. For all $[l_i : A_i]_{i=1\ldots k} \in src(A)$ and $[l'_j : B_j]_{j=1\ldots m} \in snk(A)$ we have that $\{l'_j\}_{j=1\ldots m} \subseteq \{l_i\}_{i=1\ldots k}$ and for all i, j such that $1 \leq i \leq k, 1 \leq j \leq m$, if $l_i = l'_j$ then $A_i \sim_G B_j$.
2. For all $[l_i : A_i]_{i=1\ldots k}, [l'_j : B_j]_{j=1\ldots m} \in snk(A)$ we have for all i, j such that $1 \leq i \leq k, 1 \leq j \leq m$, if $l_i = l'_j$ then $A_i \sim_G B_j$.

Let C_i be the constraints represented by the flow edges and equivalence edges in the flow graph after execution of i steps, and let C_t be the constraints represented by the flow graph after successful termination. It is easy to see by induction on i that $C_i^{\leq} \subseteq C^*$ for all $i = 1 \ldots t$. The Invariants 1 and 2 above guarantee that $\mathcal{F}(C_t^{\leq}) \subseteq C_t^{\leq}$. To wit, assume we have a derivation of $C_t^{\leq} \vdash P$ that uses any of the rules (DECOMP$^{\leq}$), (ABSURD$^{\leq}$), (UNIQLAB) only once, as the very last step. If (DECOMP$^{\leq}$) is the last rule applied, Invariant 1 implies that P is already an element of $C_t^{\leq}$. (ABSURD$^{\leq}$) cannot be the last rule applied since this would yield a contradiction to Invariant 1. If (UNIQLAB) is the last rule applied, Invariant 2 implies that $P \in C_t^{\leq}$.

We also have $C \subseteq C_t^{\leq}$ since all the constraints in C are added to G and W in Steps 2 and 3, no flow or equivalence edge is ever removed from G, and for all equational constraints $A = B$ inserted into W we eventually have $A \sim_G B$.

Thus we have $C \cup \mathcal{F}(C_t^{\leq}) \subseteq C_t^{\leq}$. Proposition 5 implies that $C^* \subseteq C_t^{\leq}$ and thus $C^* = C_t^{\leq}$ in the case where the algorithm terminates without failure. Since the algorithm terminates without failure only if $\bot$ is not derived it follows by Lemma 4 that C is solvable. If, on the other hand, the algorithm terminates with failure, then it must be that $\bot \in C_i \subseteq C^*$ for some i. By Lemma 4 we can conclude that C is unsolvable in this case. Putting this together with Lemma 3 we get the following correctness theorem:

Theorem 6. *Given constraints C_a generated for an object expression a. If the algorithm in Figure 7 terminates with success then a is typable in $\mathbf{Ob}_{1<:\mu}$. If it terminates with failure then a is untypable in $\mathbf{Ob}_{1<:\mu}$.*

By applying a circularity check, corresponding to Rule (CYCLE), once the algorithm in Figure 7 has terminated successfully, we obtain a correct algorithm for $\mathbf{Ob}_{1<:}$.

2.7. Algorithm execution: Example

In order to illustrate how the algorithm of the previous section works let us reconsider the object expression *fix* and its constraints C_{fix}, listed in Figure 4. Figure 8 shows the flow graph and the contents of W at the beginning of the 7-th iteration of Phase II, just before constraint $\nu = [arg : \beta, val : \mu]$ is extracted in Step 5. Tree edges are shown as thick solid lines, flow edges as pointed dashed lines, and equivalence edges as thin solid lines.

Figure 9 shows the flow graph and W at the end of the 7-th iteration. One equational constraint has been added: $\beta' = \beta$. The equivalence edges and flow edges that are visited during Steps 7–9 are highlighted, and $\nu = [arg : \beta, val : \mu]$ has been added as an equivalence edge to the graph.

In the next iteration constraint $\alpha_x = \alpha_y$ is extracted from W.[4] Figure 10 shows the flow graph at the end of the iteration, with the edges that are traversed in Steps 7–9 highlighted and the equivalence edge for $\alpha_x = \alpha_y$ inserted. Note that two constraints have been added: $\gamma = \beta$ and $\delta' = \mu$.

In this fashion the algorithm executes until W is empty. Since it terminates without failure we can conclude that *fix* is typable in $\mathbf{Ob}_{1<:\mu}$. A cycle test on the final flow graph shows that it has no proper cycles and thus *fix* is already typable in $\mathbf{Ob}_{1<:}$.

2.8. 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 $v = O(n)$ nodes and that C_a contains $e = O(n)$ constraints. We shall now investigate the total execution time taken by each one of Steps 1–10 in Figure 7.

Here we assume that a union/find data structure with path compression and union by weight or rank [30] is used to represent the equivalence relation defined by the equivalence edges. A union/find data structure represents every equivalence class of nodes by one of its elements, its *equivalence class representative (ecr)*. Adding a new equivalence edge joining distinct equivalence classes is done by a *union* operation that effectively *contracts* the joined equivalence classes into a single one and uses one of the two original ecr's as the new ecr. Note that a flow graph can be contracted at most $v - 1 = O(n)$ times. Access from a node to its ecr is

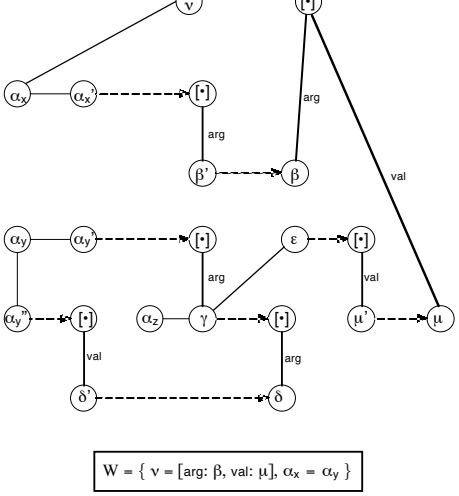

FIG. 8. Iteration 7: Flow graph just before Step 5

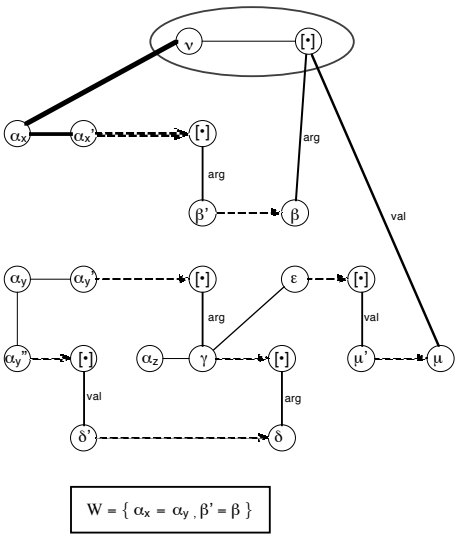

FIG. 9. Iteration 7: Flow graph after Step 10

done by a *find* operation. The amortized cost of a union operation on ecr's is $O(1)$. The cost of a find operation is $O(\alpha(m,n)$ where α is an inverse of the Ackerman function, m is the number of union and find functions executed on a set of n elements. Important in this context is that $\alpha(m,n) = O(1)$ if m grows asymptotically just a little bit more than n; e.g., if $m = \Omega(n \log\log n)$ [30, p. 26]. In particular, we have $\alpha(n^2, n) = O(1)$.

The workset W can be represented by a data structure of size $O(v) = O(n)$ that supports extracting, deleting and adding of an element in time $O(1)$. Recalling that nodes are represented by the interval $[1 \ldots v]$ this can be done by a doubly-linked list together with a bit array of length v. This can also be accomplished even without arrays [20].

Step 1. Building the term graph G for C_a takes time $O(v) = O(n)$.

Step 2. Initializing W to the set of equational constraints from C_a takes time $O(e) = O(n)$.

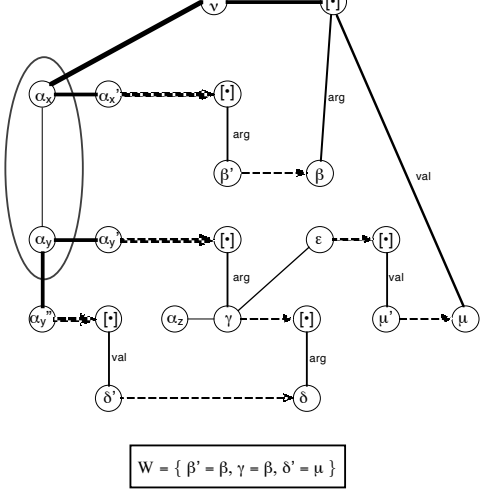

FIG. 10. Iteration 8: Flow graph after Step 10

Step 3. Adding all the subtyping constraints from C_a to G takes time $O(e) = O(n)$.

Step 4. Using a single array of size $O(e) = O(n)$ — similar to the implementation of Step 7 in Figure 11 — Step 4 requires time $O(nc)$, where c is the amortized cost of a union/find operation. We shall see that $c = O(1)$. Thus the cumulative cost of Step 4 is $O(n)$.

Step 5. Extracting a node A from W and deleting it can be done in constant time. Step 4 makes a total of $O(n)$ insertions into W. Note that every time a node is added to W in Steps 7–9 the number of equivalence classes in induced by $G \cup W$ is decreased

Let Arr be an array of size $e = O(n)$.

1. (Preprocessing) For all l occurring in C_a set $\mathrm{Arr}(l)$ to 0 (0 denotes an undefined node).
2. Set Count to 0. (Count counts the number of all method names occurring in $snk(A)$.)
3. (Step 7) For all $[l'_j : A'_j]_{j=1\ldots m} \in snk(A) \cup snk(B)$ do:
   - For all $j = 1 \ldots m$,
     - if $\mathrm{Arr}(l'_j) = 0$ then set $\mathrm{Arr}(l'_j)$ to A'_j and add 1 to Count; else
     - if $\mathrm{Arr}(l'_j) \not\sim_{G \cup W} A'_j$ then add $\mathrm{Arr}(l'_j) = A'_j$ to W.
4. (Steps 8 and 9) For all $[l_i : A_i]_{i=1\ldots k} \in src(A) \cup src(B)$ do:
   - Initialize Count′ to Count.
   - For all $i = 1 \ldots k$ such that $\mathrm{Arr}(l_i) \neq 0$:
     - decrease Count′ by 1;
     - if $\mathrm{Arr}(l_i) \not\sim_{G \cup W} A_i$ then add $\mathrm{Arr}(l_i) = A_i$ to W.
   - If Count′ $\neq 0$ then abort with failure.

FIG. 11. Algorithm for computing Steps 7–9

$$\text{(VAL)}\quad E, x : A, E'' \vdash x : A' \quad (x \notin \mathit{domain}\, E'') \qquad A = A'$$

$$\text{(OBJ)}\quad \frac{E, x_i : A \vdash b_i : B_i' \qquad (\forall i : 1 \le i \le k)}{E \vdash [l_i = \varsigma(x_i) b_i]_{i=1\ldots k} : A'} \qquad \left\{ \begin{array}{l} \{B_i' = B_i\}_{i=1\ldots k} \\ A = [l_i : B_i]_{i=1\ldots k} \\ A' = A \end{array} \right\}$$

$$\text{(SEL)}\quad \frac{E \vdash a : A}{E \vdash a.l : B} \qquad A \le [l : B]$$

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

FIG. 12. Normalized type inference rules (without subtyping)

by 1. Thus there can be at most $v - 1 = O(n)$ insertions into W in Steps 7–9 altogether. Thus this step takes a total of $O(n)$ time.

Step 6. Checking whether $A \sim_G B$ takes time $c = O(1)$. Since this operation is executed $O(v) = O(n)$ times, the cumulative cost of Step 6 is $O(n)$.

Steps 7–9. For given A and B the algorithm in Figure 11 implements Steps 7–9. It executes in time $O(nc)$. One way to see this is to note that there is a total of $O(e) = O(n)$ method name occurrences in $src(A)$ and $snk(A)$ combined, and the total running time is a linear function of the sum of method name occurrences and the size of G, multiplied with the cost of a union/find operation. Since Steps 7–9 are executed at most $O(v) = O(n)$ times the total cost attributable to Steps 7–9 is $O(n^2 c)$. Now, since there are $O(n^2)$ union/find operations on $v = O(n)$ elements in total we have that $c = O(\alpha(n^2, n)) = O(1)$. As a consequence, Steps 7–9 take time $O(n^2)$.

Step 10. Adding an equivalence edge to G takes time $c = O(1)$. Since this is done $O(v) = O(n)$ times the cumulative costs of Step 10 is $O(n)$.

We have shown that the Algorithm in Figure 7 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 [22] who, instead of contracting nodes as we do in Step 10, 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 in time $O(n)$.

The algorithm executes in linear space since $O(n)$ equational constraints are added to W and the flow graph G can be represented in space $O(n)$.

Theorem 7. *Given object expression a of size n it can be decided in time $O(n^2)$ and space $O(n)$ whether a is typable in system $\mathbf{Ob}_{1<:}$ or $\mathbf{Ob}_{1<:\mu}$.*

There are a number of more-or-less obvious optimizations and implementation choices that should be applied to the Algorithm of Figure 7 to ensure practically improved performance. The present form was chosen to facilitate the analysis of both correctness and complexity. Since these optimizations give no asymptotic improvements, they are omitted here.

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 $B_i' \le B_i, i \in 1 \ldots n$ in Rule $(\text{OBJ})^n$ by $B_i' = B_i$ and $B' \le B$ in Rule $(\text{UPD})^n$ by $B' = B$. We thus arrive at the presentation of the inference rules defining $\mathbf{Ob}_{1\mu}$ and $\mathbf{Ob}_1$, presented in Figure 12. Note that these are the same rules as in Figure 1: they are linearized in order to make the constraint extraction obvious.

Lemma 8. *Given object expression a there is a set of equational and subtyping constraints C_a such that C_a is solvable over recursive types, respectively finite types, if and only if a is typable in $\mathbf{Ob}_{1\mu}$, respectively $\mathbf{Ob}_1$.*

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

These lemmas guarantee that the algorithm in Section 2.5 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 faster algorithms.

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

Consider the constraints in C_a generated for an object expression according to Lemma 8. They fall into three categories:

1. $X = Y$ where X and Y are type variables;
2. $X = [l_i : X_i]_{i=1\ldots k}$ 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 in Figure 18. It is derived from the general Algorithm in Figure 7 in the following fashion:

- Because of the absence of constraints of the form $X \leq Y$ computation of $src(A)$ is not necessary. Instead, we can use $find(A)$, which returns the equivalence class representative of A.
- Similarly, computing $snk(A)$ is avoided by maintaining a proper object type $SNK(X)$ for each type variable that is also an ecr. It contains a component $l : A$ if and only if $[l : A]$ (from an original constraint of the form $X \leq [l : Y]$) is reachable from X in G.
- Steps 7–10 of Figure 7 have been turned into four disjoint cases, depending on whether the ecr's of A and B — which can be thought of as representing the sources of A and B, respectively — are proper types or type variables.

Let us analyze the performance of the Algorithm in Figure 18. By representing $\{l_i\}_{i=1\ldots k}$ in type $[l_i : A_i]_{i=1\ldots k}$ as a balanced search tree we can check whether $l = l_i$ for some $i, 1 \leq i \leq k$, and retrieve A_i in time $O(\log k) = O(\log n)$. If we use hashing then this bound can be improved to $O(1)$ probabilistic or expected time.

Steps 1-4 can be implemented in time $O(n)$. The total runtime cost of Steps 5 and 6 is $O(ic)$, where i is the total number of insertions into W and c is the amortized cost of a union/find operation. We shall see that there are at most $v = O(n)$ insertions and $c = O(\alpha(n \log n, n)) = O(1)$. Thus Steps 5 and 6 together account for time $O(n)$.

We can implement each of Steps 8–10 by a loop over $\{l_i\}_{i=1\ldots k}$ and checking whether each method name is present in $\{l'_j\}_{j=1\ldots m}$. Since every occurrence of l_i in C_a is processed at most once in this fashion and each insertion takes $O(\log m) = O(\log n)$ time, Steps 8–10 take time $O(n \log n)$. Step 7 is the bottleneck in our algorithm. In this case the smaller of $SNK(X), SNK(Y)$ is *merged* into the other to form $SNK(X)$ after unioning X and Y. This guarantees that any component $l : Y$ from an original constraint $X \leq [l : Y]$ is inserted in a merge at most $O(\log n)$ times. Since each such individual insertion takes time $O(\log n)$ and there can be at most $O(n \log n)$ such insertions in total, we have a total cost of $O(n \log^2 n)$ for Step 7. This gives us an $O(n \log^2 n)$ time algorithm for $\mathbf{Ob}_{1\mu}$.

Note that that there are no more than $v - 1 = O(n)$ insertions of equational constraints into W. Also the flow graph, including all representations of $SNK(A)$, requires $O(n)$ space. This shows that the algorithm requires only linear space.

Theorem 9. *Given object expression a of size n it can be decided in time $O(n \log^2 n)$ and space $O(n)$ whether a is typable in system $\mathbf{Ob}_{1\mu}$.*

Using hashing the time upper bound is improved to $O(n \log n)$ time, though this is not deterministic worst case, but either a probabilistic upper bound or an upper bound for the expected time.

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. This gives rise to the most interesting algorithm. From an object-oriented perspective $\mathbf{Ob}_1$ is not very interesting since it has neither subtyping nor recursive types. Our algorithm appears to be applicable to and relevant for ML-style type inference for records, however. As such it seems to improve on the asymptotic worst-case performance of previously known algorithms in that area.

The type inference algorithm for $\mathbf{Ob}_1$ is given in Figure 13. We describe its underlying ideas and give a brief analysis of its time complexity. A thorough, detailed proof of its correctness and complexity will have to await treatment in a separate paper.

We add all subtyping and equational constraints to the term graph for C_a as flow edges and equivalence edges, respectively. We do not use a union/find representation for the equivalence edges, however, but represent them as adjacency lists $Equiv(A)$ for each node A. Initially $Equiv(A)$ contains the set of all nodes B such that either $A = B$ or $B = A$ occurs in C_a. Similarly, $Subrel(A)$ contains the set of all B such that either $A \leq B$ or $B \leq A$ is in C_a. Finally, for each proper object type $[l_i : B_i]_{i=1\ldots k}$ we stipulate that each B_i has a pointer to its parent: $Parent(B_i) = [l_i : B_i]_{i=1\ldots k}$ for $1 \leq i \leq k$. For nodes without parents, their parent pointer is set to 0.

We now perform 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

From a given node we follow its tree edge leading to its parent — if any — *before* any other edges are traversed. This is critical since during the search new equivalence edges are inserted in the graph. The or-

DFS:

- currentlevel := 0;
- initialize stack to empty stack;
- for each node A in C do:
  - color(A) := WHITE;
- for each node A in C do:
  - DFS(A)

DFS(A):

- If color(A) = WHITE then:
  - color(A) := GRAY;
  - level(A) := currentlevel;
  - add A to the top of the stack;
  - if $Parent(A) \neq 0$ then:
    - increment currentlevel with 1;
    - push the empty set onto the stack;
    - DFS($Parent(A)$);
    - process(top of the stack);
    - pop the stack;
    - decrement currentlevel with 1.
  - for each $B \in Subrel(A)$ do:
    - DFS(B);
  - while there exists $B \in Equiv(A)$ do:
    - delete B from $Equiv(A)$;
    - DFS(B);
  - color(A) := BLACK and return.
- If color(A) = GRAY then:
  - if currentlevel = level(A) then return, else abort with failure.
- If color(A) = BLACK then return.

process(S):

- For each proper object type $[l_i : A_i]_{i=1\ldots k} \in S$ do:
  - for $i = 1 \ldots k$ do:
    - if Arr(l_i) = 0 then Arr(l_i) := A_i;
    - else add Arr(l_i) to $Equiv(A_i)$ and add A_i to $Equiv$(Arr(l_i)).
- Reset all elements of Arr to 0.

FIG. 13. Depth-first search based algorithm for $\mathbf{Ob}_1$

finished no equivalence edges incident to that node will ever be added, even though additional equivalence edges may still be inserted other places.

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 prop-

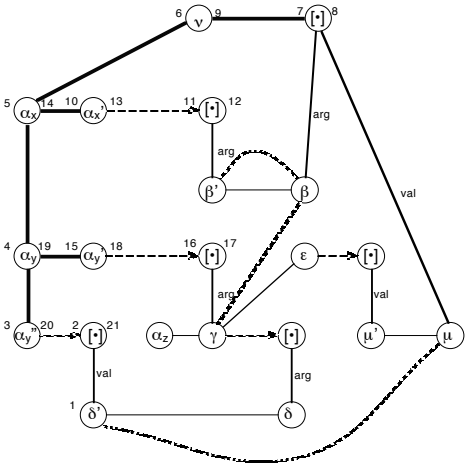

FIG. 14. DFS algorithm (1)

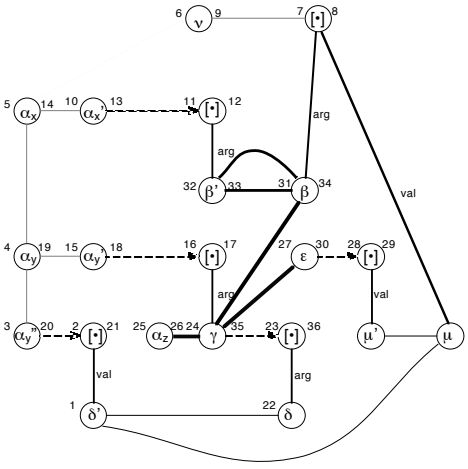

FIG. 15. DFS algorithm (2)

edges connect all the nodes incident to them. We exploit this in the algorithm by collecting not the equivalence edges themselves, but their incident nodes (and of those we only need the proper object types).

Using a single, global array we can process all the nodes in 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 total number of incident tree edges; that is by the total number of method name occurrences in the object expression. 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)

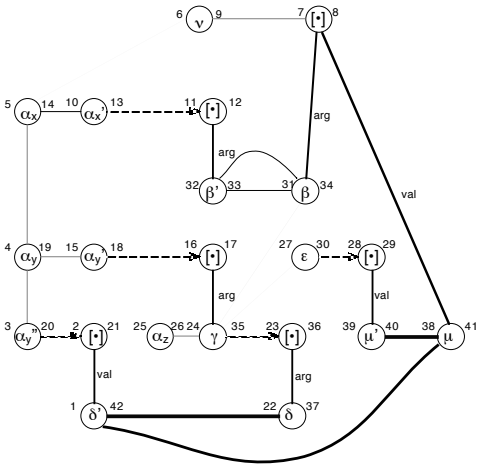

FIG. 16. DFS algorithm (3)

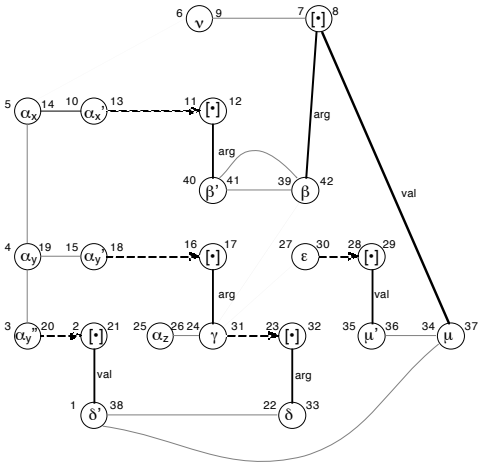

FIG. 17. DFS algorithm (4)

is visited that is "at least one tree edge away" then we can terminate with failure due to the (CYCLE) rule.

Theorem 10. *Given object expression a of size n it can be decided in time $O(n)$ and space $O(n)$ whether a is typable in system $\boldsymbol{Ob}_1$.*

We demonstrate the algorithm's execution on our running example, the object term *fix*. Figure 14 shows the depth-first-search algorithm after execution of a number of steps. The first node visited is δ'. The numbers labeling nodes are *discovery* and *finish* times [4, Section 23.3]. They show in which order the nodes have been processed so far. The algorithm is about to finish an object type constructor labeled node and return to δ'. The currently active equivalence edges are indicated by somewhat thicker lines (not to be confused with the tree edges, which are labeled with method names). At this point the new equational constraints $\delta' = \mu, \gamma = \beta, \beta' = \beta$ are added as new equivalence edges, indicated by dashed lines, and all the active equivalence edges are marked processed

Figure 14, 15, 16 and 17 show the algorithm in successive stages. In each case the numbers labeling nodes are discovery and finish times. Note that the flow graph contains *two* equivalence edges between nodes β' and β. This is because our algorithm does not check whether an equivalence edge between two nodes already exists between inserting a new one. Such a check is neither necessary nor algorithmically desirable.

4. Related work

Type inference for object-oriented languages has been studied for records and record operations without subtyping or with named subtyping [32, 14, 33, 25, 26], and for records with variant subtyping [7, 6], possibly based on value flow analysis [21]. None of these works, however, deal with invariant object subtyping.

Palsberg [22] 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 [17, 34] 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 [12]. 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 components 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) and recursive types 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. Results by Heintze/McAllester [11] and Melski/Reps [18] suggest that it is unlikely that these asymptotic bounds are easily improved. (They *can* be improved in the absence of recursive types.)

Finally a remark about *unrelated* work: Heintze and McAllester [10] and Mossin [19] have given value flow analysis algorithms that compute the value flow relation between constructors and destructors in a simply-typed higher-order functional program in time $O(n^2)$ *assuming the size of types of all subexpressions is bounded by some constant* k.[5] Apart from the runtime complexity bound there is little that our results have in common with those of Heintze/McAllester and Mossin:

- Their value flow analysis setting corresponds to a subtyping framework with co- and contravariant subtyping. In our setting we have invariant subtyping, which is, from a value flow perspective, less precise.
- Their bound only holds for simply typed programs whose types are constant-bounded. Taken over *all* typed programs their algorithms degenerate to taking exponential time, and taken over all *untyped* programs, type checking and computing the value flow closure as presented takes doubly-exponential time. This makes their algorithms asymptotically inferior to ordinary transitive-closure based value flow analysis.
  In contrast, we give general algorithms for typability in $\mathbf{Ob}_{1<:\mu}$ and $\mathbf{Ob}_{1<:}$ with guaranteed worst-case time behavior $O(n^2)$ over *all* untyped object expressions of size n.

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, all of which running in linear space. This improves the time and space bounds given by Palsberg for all four calculi studied. The arguably singly most important improvement in our algorithms is that they run in linear space, in contrast to Palsberg's algorithms, which require quadratic space. This increases dramatically the size of problem instances that can realistically be processed.

The techniques and results presented here can be used in object type systems for real-scale programming languages, in particular those that employ an invariant subtyping rule for object types. Even in cases where (co- and contra-)variant subtyping is allowed we expect that our basic method of eliminating as many subtyping constraints in favor of equational constraints results in good practical performance.

Even though our linear-time algorithm for $\mathbf{Ob}_1$ seems to be of little relevance in object type inference it appears to be directly applicable to ML-polymorphic type inference with records. As such it seems to be an asymptotic improvement over previously known algorithms. This is a topic for future work.

The normalized constraint systems give rise to a principal typing property for object expressions, using subtype qualified type schemes with relatively few subtype qualifications. Principal typings and their simplification by both syntactic and semantic entailment are the subject of a substantial number of papers; see *e.g.* [9, 5, 15, 29, 28, 6, 13, 23, 31, 24, 8]. All of these, however, operate in a context where co- or contravariant subtyping is admitted and exploited during simplification. Since these simplifications are, in principle, unsound for invariant subtyping, principal typings with invariant object subtyping and their simplification is another 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 an 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.

Thanks to the two referees for their corrections and numerous good suggestions. One of the referees made two very important observations that I had overlooked: that the algorithms presented here run in linear space, which is probably practically more important than the improvements in time complexity, and that the linear-time algorithm for $\mathbf{Ob}_1$ is applicable to and relevant for simple type inference with records.

This research was partially supported by the Danish Research Council, Project *DART*.

Notes

1. For simplicity's sake we use somewhat informal notation in that we do not axiomatize well-formedness of type environments.
2. Uniqueness is up to renaming of type variables.
3. Note that there are three kinds of edges in flow graphs: tree edges, flow edges and equivalence edges.
4. The actual order in which constraints are extracted is completely irrelevant for the correctness and analysis of the algorithm.
5. Heintze and McAllester entitle their paper a linear time control flow analysis. That is a bit misleading since preprocessing and *single-query* (what can be reached from this point?) processing time is $O(n)$. To compute the value flow for the whole program in general $\Theta(n)$ such queries are required. Furthermore, they refer to their analysis as 0CFA, which, unfortunately, has been used in a different sense by Shivers [27], who introduced the term.

References

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

[2] 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.

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

[4] 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).

[5] P. Curtis. Constrained quantification in polymorhic type analysis. Technical Report CSL-90-1, Xerox Parc, February 1990.

[6] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proc. 10th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Austin, Texas*, number 10 in ACM SIGPLAN Notices, Vol. 30, pages 169–184. ACM Press, October 1995.

[7] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. 11th Conf. on the Mathematical Foundations of Programming Semantics (MFPS)*, April 1995.

[8] Manuel Fähndrich and Alex Aiken. Making set-constraint program analyses scale. In *Proc. Workshop on Set Constraints, Cambridge, Massachussetts*, 1996.

[9] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. Int'l J't Conf. on Theory and Practice of Software Development*, pages 167–183, Barcelona, Spain, March 1989. Springer-Verlag.

[10] Nevin Heintze and David McAllester. Linear time subtransitive control flow analysis. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 261–272, http://www.acm.org, 1997. ACM, ACM Press.

[11] Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proc. 22nd Annual IEEE Symp. on Logic in Computer Science (LICS), Warsaw, Poland*. IEEE, IEEE Computer Society Press, June 1997.

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

[13] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proc. 22nd ACM Symp. on Principles of Programming Languages (POPL), San Francisco, California*, pages 176–185, January 1995.

[14] L. Jategaonkar and J. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proc. ACM LISP and Functional Programming Conf.*, pages 198–211. ACM, July 1988.

[15] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), San Francisco, California*, pages 193–204. ACM, ACM Press, June 1992. also in LISP Pointers, Vol. V, Number 1, January-March 1992.

[16] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science (MSCS)*, 5(1), 1995. Also presented at the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL), 1993.

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

[18] David Melski and Thomas Reps. Interconvertibility of set constraints and context-free language reachability. In *Proc. ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 74–89, http://www.acm.org, June 1997. ACM, ACM Press.

[19] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. Technical report diku-tr-97/1, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, 1997. http://www.diku.dk/research/published/97-1.ps.gz.

[20] Robert Paige and Zhe Yang. High level reading and data structure compilation. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL), Paris, France*, pages 456–469, http://www.acm.org, January 1997. ACM, ACM Press.

[21] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Phoenix, Arizona*, pages 146–161. ACM Press, October 1991.

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

[23] François Pottier. Simplifying subtyping constraints. In *Proc. Int'l Conf. on Functional Programming (ICFP), Philadelphia, Pennsylvania*, pages 122–133. ACM Press, May 1996.

[24] Jakob Rehof. Minimal typings in atomic subtyping. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programmin gLanguages (POPL), Paris, France*, pages 278–291. ACM Press, January 1997.

[25] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 67–95. MIT Press, 1994. Also available as Research Report 1431, May 1991, INRIA-Rocquencourt, France. Previous results presented at POPL '89.

[26] John Reppy and John Riecke. Simple objects for standard ml. In *Proc. ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania*, pages 171–180, http://www.diku.dk, May 1996. ACM, ACM Press.

[27] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, May 1991.

[28] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[29] Geoffrey Seward Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Cornell University, August 1991.

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

[31] Valery Trifonov and Scott Smith. Subtying constrained types. In Radhia Cousot and David A. Schmidt, editors, *Proc. 3rd Int'l Static Analysis Symposium (SAS), Aachen, Germany*, volume 1145 of *Lecture Notes in Computer Science (LNCS)*, pages 349–365. Springer-Verlag, September 1995.

[32] M. Wand. Complete type inference for simple objects. In *Proc. Symp. on Logic in Comp. Sci.*, pages 37–44, June 1987.

[33] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 1989 IEEE 4th Annual Symp. on Logic in Computer Science (LICS)*, pages 92–97. IEEE Computer Society Press, 1989.

[34] 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.

Phase I (Initialization)

1. Build the term graph G representing all the types in C. Set $SNK(X) = []$ for all type variables X occurring in C.
2. Initialize W to the set of equational constraints in C.

3–4. For each subtyping constraint $X \leq [l : Y]$ in C do:

- if $SNK(X) = [l : Y', \ldots]$ and $Y \not\sim_{G \cup W} Y'$ then add $Y = Y'$ to W; else
- add component $l : Y'$ to $SNK(X)$.

Phase II (Process all equational constraints)
While W is nonempty do:

5. Extract a constraint $A = B$ from W and delete it from W.
6. If $A \sim_G B$ then jump to the previous step. Otherwise continue with the following step.
7. If $\mathit{find}(A) = X, \mathit{find}(B) = Y, SNK(X) = [l_i : A_i]_{i=1\ldots k}$ and $SNK(Y) = [l'_j : B_j]_{j=1\ldots m}$ then:
   - For all i such that $1 \leq i \leq k$ do:
     - if $l_i = l'_j$ for some j where $1 \leq j \leq m$, then: if $A_i \not\sim_{G \cup W} B_j$ add $A_i = B_j$ to W (end inner if); else
     - add component $l_i : A_i$ to $SNK(Y)$.
   - Union X and Y such that their new ecr is Y.
8. If $\mathit{find}(A) = [l_i : B_i]_{i=1\ldots k}$ and $\mathit{find}(B) = [l'_j : B_j]_{j=1\ldots m}$ then:
   - If $\{l_i\}_{i=1\ldots k} \neq \{l'_j\}_{j=1\ldots m}\}$ then abort with failure.
   - For all i, j such that $1 \leq i \leq k, 1 \leq j \leq m$ and $l_i = l'_j$ add $A_i = B_j$ to W.
   - Union $[l_i : B_i]_{i=1\ldots k}$ and $[l'_j : B_j]_{j=1\ldots m}$.
9. If $\mathit{find}(A) = [l_i : B_i]_{i=1\ldots k}, \mathit{find}(B) = X$ and $SNK(X) = [l'_j : B_j]_{j=1\ldots m}$ then:
   - If $\{l'_j\}_{j=1\ldots m} \not\subseteq \{l_i\}_{i=1\ldots k}$ then abort with failure.
   - For all i, j such that $1 \leq i \leq k, 1 \leq j \leq m$ and $l_i = l'_j$ add $A_i = B_j$ to W.
   - Union $[l_i : B_i]_{i=1\ldots k}$ and X such that their new ecr is $[l_i : B_i]_{i=1\ldots k}$.
10. If $\mathit{find}(A) = X, \mathit{find}(B) = [l'_j : B_j]_{j=1\ldots m}$ and $SNK(X) = [l_i : B_i]_{i=1\ldots k}$ then do as for Step 7.

FIG. 18. Closure algorithm for $\mathbf{Ob}_{1\mu}$