In *Information and Computation* © Academic Press. To appear.

# Type Inference for Variant Object Types

Michele Bugliesi
*Dipartimento di Informatica, Università Ca' Foscari di Venezia*
E-mail: michele@dsi.unive.it

and

Santiago M. Pericás-Geertsen
*Department of Computer Science, Boston University*
E-mail: santiago@cs.bu.edu

Existing type systems for object calculi [2] are based on invariant subtyping. Subtyping invariance is required for soundness of static typing in the presence of method overrides, but it is often in the way of the expressive power of the type system. Flexibility of static typing can be recovered in different ways: in first-order systems, by the adoption of object types with *variance annotations*, in second-order systems by resorting to *Self types*. Type inference is known to be P-complete for first-order systems of finite and recursive object types, and NP-complete for a restricted version of Self types. The complexity of type inference for systems with variance annotations is yet unknown.

This paper presents a new object type system based on the notion of *Split types*, a form of object types where every method is assigned two types, namely, an update type and a select type. The subtyping relation that arises for Split types is *variant* and, as a result, subtyping can be performed both in *width* and in *depth*. The new type system generalizes all the existing first-order type systems for objects, including systems based on variance annotations. Interestingly, the additional expressive power does not affect the complexity of the type inference problem, as we show by presenting an $O(n^3)$ inference algorithm.

## 1. INTRODUCTION

Type inference, the process of automatically inferring type information from untyped or partially typed programs, plays an important role in the static analysis of computer programs. Originally devised by Hindley [11] and independently by Milner [13], it has found its way into the design of several recent programming languages. Type inference may or may not be possible, depending on the language and the typing rules. If it can be carried out, type inference turns untyped programs into strongly typed ones. Modern languages such as Haskell [17], Java [9], and ML [14] were all designed with strong typing in mind. While functional languages such as ML and Haskell have successfully incorporated type inference in their design, type inference for object-oriented languages is considerably less developed and has yet to achieve the same degree of practical importance.

In this paper, we consider an *untyped object-calculus* based on the formulation presented by Abadi and Cardelli, also known as the *$\varsigma$-calculus* [2]. For this calculus, Abadi and Cardelli provide a suite of type systems addressing many of the typing problems encountered in the practice of OO programming. For some of these type systems, efficient (i.e. polynomial) inference algorithms have been studied in the recent literature. In [15], Palsberg presents a method for inferring recursive object types based on a reduction to the problem of solving recursive constraints. An $O(n^3)$ algorithm is presented and a proof that the underlying problem is PTIME-complete outlined. In [16], Palsberg and Jim extend the type system proposed in [15] with the inclusion of a *simple* form of *Self types* [2]. The new system is more powerful than the system of recursive types because it relies on a more flexible subtyping relation on object types, but at the same time imposes severe restrictions on the way methods can be updated: specifically, methods returning *self* cannot be updated. In spite of these restrictions, type inference in the new system is shown to be NP-complete.

Subtyping is a key feature in any type system for object calculi, but it does not coexist naturally with recursive types in the presence of method (and field) updates. Simple and perfectly sound examples fail to type check as a result of a poor interaction between the subtyping rules for recursive types and object types.

$$\text{(Sub } \mu\text{)} \quad \frac{E, X \leq Y \vdash A \leq B}{E \vdash \mu(X)A \leq \mu(Y)B} \qquad \text{(Sub Object)} \quad \frac{(J \subseteq I)}{E \vdash [\ell_i : B_i{}^{i \in I}] \leq [\ell_j : B_j{}^{j \in J}]}$$

The problem arises from the *invariant* restriction on the component types imposed by (Sub Object). As a consequence, although it is clear that a 2D point can "subsume" a 1D point in a context where the latter is expected, the two rules above prevent the expected relationship among those types. That is, if $P_1 \equiv \mu(X)[x : \mathsf{int}, \mathit{move} : X]$ is the type of a 1D point and $P_2 \equiv \mu(X)[x : \mathsf{int}, y : \mathsf{int}, \mathit{move} : X]$ is the type of a 2D point, using (Sub $\mu$) and (Sub Object) it is not derivable that $P_2 \leq P_1$. Unfortunately, the invariance requirement imposed by (Sub Object) is *necessary* for soundness: lifting that restriction turns the system *unsound*, i.e. a reduction of a typable term may generate a run-time type error.

EXAMPLE 1.1. Given $P_1$ and $P_2$ as defined above, suppose we change the typing rules so that $P_2 \leq P_1$. Let $p_2 = [x = \varsigma(s)s.\mathit{move}.y, y = 0, \mathit{move} = \varsigma(s)s.y := s.y + 1]$. This object has one integer field $y$, with value 0, and two methods. The method *move* returns a new object where the field $y$ is incremented by 1, and the method $x$ returns the value of the field $y$ as modified by *move*. It is easy to see that $p_2$ can be assigned the type $P_2$. Thus, if $p_1$ is an arbitrary term of proper type $P_1$ (i.e. with no field $y$) then the following term,

$$(p_2.\mathit{move} := p_1).x \qquad \text{(oops !!)}$$

is typable and generates a run-time error since the term $p_2$ can be assigned the type $P_1$ by subsumption. The rest follows directly from the definition of $P_1$ and the rule for typing updates: a run-time error is produced as a result of attempting to select $y$ from $p_1$, which by assumption is a proper term of type $P_1$.

Despite their more restricted subtyping rule, recursive object types still allow useful types to be derived for terms that seem to require variant subtyping.

EXAMPLE 1.2. Let $p_1 = [x = 0, \mathit{move} = \varsigma(s)s.x := s.x + 1]$ and $p'_2 = [x = 0, y = 0, \mathit{move} = \varsigma(s)s.y := s.y + 1]$. If $P_1 \equiv \mu(X)[x : \mathsf{int}, \mathit{move} : X]$ and $P_2 \equiv \mu(X)[x : \mathsf{int}, y : \mathsf{int}, \mathit{move} : X]$ then $p_1$ can be assigned the type $P_1$ and $p'_2$ can be assigned the type $P_2$. Now, consider the term $p'_2.\mathit{move} := p_1$. This term is typable with recursive types, as $p'_2$ can be assigned the type,

$$P \equiv [x : \mathsf{int}, y : \mathsf{int}, \mathit{move} : P_1].$$

This follows from the fact that $P \leq [x : \mathsf{int}, \mathit{move} : P_1] = P_1$, where the last equality holds by unfolding $P_1$. Consequently, the term $p'_2.\mathit{move} := p_1$ is typable in this system even though we cannot prove $P_2 \leq P_1$.

How large is the set of terms for which "useful" recursive types can be inferred ? In some cases, it is possible to find a type like that for Example 1.2. In other cases, however, a more powerful system is needed.

EXAMPLE 1.3. Let $p_0 = [\mathit{move} = \varsigma(s)s]$ and $p_2 = [x = \varsigma(s)s.\mathit{move}.y, y = 0, \mathit{move} = \varsigma(s)s.y := s.y + 1]$ and let $p$ be the term $[\ell = p_2].\ell := p_0$. Notice that in $p_2$, method $x$ refers (indirectly) to method $y$ via *move*. Using recursive types, the only common type that can be assigned to $p_0$ and $p_2$ for the update to type check is $[\,]$ (the empty object type). Contrary to Example 1.2, the dependency between $x$ and $y$ through *move*, does not allow us to assign the type:

$$[x : \mathsf{int}, y : \mathsf{int}, \mathit{move} : \mu(X)[\mathit{move} : X]]$$

to $p_2$ so that it can be subsumed to $\mu(X)[\mathit{move} : X]$. As a result, the most informative type for $p$ that can be inferred using recursive types is $[\ell : [\,]]$. An immediate consequence of this observation is that the term $p.\ell.\mathit{move}$ is not typable with recursive types.

To overcome these difficulties, Abadi and Cardelli propose several solutions. Among them, two solutions based on static typing have emerged as most interesting. The first is the use of *variance* annotations to surmount the restrictions imposed by invariant subtyping. Using variance annotations, the type of 2D points can be written as $P_2^+ \equiv \mu(X)[x : \mathsf{int}, y : \mathsf{int}, \mathit{move}^+ : X]$ where the superscript $^+$ on *move* signals that this method is read-only. With this restriction, 2D points can subsume 1D points, as $P_2^+ \leq P_1^+ \equiv \mu(X)[y : \mathsf{int}, \mathit{move}^+ : X]$ is validated by the subtyping rule. The price to pay, of course, is that the *move* method cannot be updated. The second and more refined solution is the system of Self types, which is based on a combination of recursive and bounded existential types. In this system, it is possible to prove subtyping relations like $P_2 \leq P_1$ as a result of the inclusion of a clever (and sound) update rule. This solution also has a price: the type inference problem for this system appears to be at least as complex as in the system of [16].

The system of Split types presented in this paper offers an alternative solution for combining subtyping, recursive types and method updates in a sound and flexible way. Split types are first-order types of the form $[\ell_i : (B_i^u, B_i^s)^{i \in I}]$. They are a variation of the recursive object types presented in [2], obtained by splitting the type of each method $\ell_i$ into two components. Intuitively, the component $B_i^u$ – or *update* component – is used to type an update for $\ell_i$, whereas the component $B_i^s$ – or *select* component – is used to type a selection for $\ell_i$. The operational behavior of the underlying calculus is not affected by this presentation of object types. That is, objects are still formed as a collection of methods of the form $[\ell_i = \varsigma(s)\, b_i{}^{i\in I}]$. Instead, the presence of two component types for each label allows subtyping over Split types to be defined *variantly*: more precisely, contravariantly in the update component and covariantly in the select component of each method type.

The idea of "splitting" types of updatable values has already been studied in existing type systems. It has first been applied to reference types in the design of the language Forsythe [20], and subsequently been adopted by other authors [8, 18, 22] for similar purposes. However, in the context of object calculi, Split types represent a technical novelty and their use has interesting consequences in terms of both typing power and practical significance.

- Variant subtyping interacts well with the subtyping rule for recursive types. The following rule, which is sound for Split types, allows subtyping to be performed both in *width* (as for the object types in [2]) and in *depth*.

$$\frac{E \vdash C_j^u \leq B_j^u \quad E \vdash B_j^s \leq C_j^s \quad (J \subseteq I)}{E \vdash [\ell_i : (B_i^u, B_i^s)^{i\in I}] \leq [\ell_i : (C_i^u, C_i^s)^{i\in J}]}$$

- Depth subtyping, in turn, induces a rich subtype hierarchy where (more informative) least upper bounds and greatest lower bounds exist for every pair of types. As a consequence of the additional subtyping power, the type system based on Split types generalizes all existing first-order type systems for objects, including those based on variance annotations. We demonstrate this by presenting (sub)type preserving encodings of all these systems into the system of Split types, and providing examples showing that the inclusion is strict (see Example 2.1, and Example 4.2). We also show that the system of simple Self types presented in [16] can be encoded using Split types.

- The additional expressive power of Split types does not affect the time complexity of the type inference problem, as we show by presenting a sound and complete $O(n^3)$ inference algorithm.

In the next section we introduce the system of Split types $\mathbf{Ob}^{\downarrow\uparrow}$, prove type soundness and discuss other useful properties. In section 3, we present the inference algorithm for $\mathbf{Ob}^{\downarrow\uparrow}$ and prove it sound and complete. In section 4, we analyze the relationships between our system and three other existing systems: recursive types with variance annotations and Self types from [2], and simple Self types from [16]. Section 5 concludes our presentation with final remarks.

## 2. THE SPLIT TYPES SYSTEM $\mathbf{Ob}^{\downarrow\uparrow}$

Let $s, s', x, x'$... range over a countably infinite set $\mathsf{Var}$ of term variables and $q, q', \ldots$ over a finite set of term constants. The set of terms is defined by the following productions:

$$a, b, c, d ::= q \mid s \mid [\ell_i = \varsigma(s)\, b_i{}^{i\in I}] \mid a.\ell \mid a.\ell \Leftarrow \varsigma(s) b$$

Terms of the form $[\ell_i = \varsigma(s)\, b_i{}^{i \in I}]$ denote objects, $a.\ell$ selects the label $\ell$ from $a$, and $a.\ell \Leftarrow \varsigma(s)b$ modifies the current body of $\ell$ in $a$ replacing it with $\varsigma(s)b$.[1] The set of free variables of a term $a$ is denoted by $\mathsf{FV}(a)$. As in [2], we write $[\cdots, \ell = b, \cdots]$ to stand for $[\cdots, \ell = \varsigma(s)b, \cdots]$ and $a.\ell := b$ to stand for $a.\ell \Leftarrow \varsigma(s)b$ whenever $s \notin \mathsf{FV}(b)$. We write $b\{s\}$ to emphasize that the variable $s$ may occur free in $b$ and $b\{\{c\}\}$ for the term that results from substituting $c$ for every free occurrence of $s$ in $b$.

### 2.1. Types and Subtypes

Let $\Sigma$ be a signature that includes the type constructors $\bot$, $\top$, $[\,]$ and $Q$, where $Q$ denotes primitive types such as $\mathsf{int}$, $\mathsf{bool}, \ldots$, etc. Let $\mathcal{L}$ be a (possibly infinite) set of labels or method names. A path $\pi$ is a finite string drawn from the set $\{\ell^u, \ell^s\}^*$ for $\ell \in \mathcal{L}$. The parity of a path $\pi$, symbolically $\mathsf{parity}(\pi)$, is the number of labels superscripted by $u$ it contains modulo 2. A type $A$ is a partial function from paths into $\Sigma$ whose domain is non-empty and prefix-closed, and with the property that $A(\pi\ell^u)$ and $A(\pi\ell^s)$ are defined only if $A(\pi) = [\,]$. The domain of a type $A$, denoted by $\mathsf{Dom}(A)$, is the set of paths on which the type is defined. Given a type $A$ and a path $\pi$ in $\mathsf{Dom}(A)$, we define $A{\downarrow}\pi$ to be the subtree of $A$ rooted at $A(\pi)$. A type is *regular* if and only if it contains finitely many different subtrees.

A Split type is a regular type over $\Sigma$. We denote with $\top$, $\bot$ and $Q$ the types $\{\epsilon \to \top\}$, $\{\epsilon \to \bot\}$ and $\{\epsilon \to Q\}$, respectively. A Split type $A$ can be written in "displayed" form $[\ell_i : (B_i^u, B_i^s)^{i \in I}]$ whenever $A(\epsilon) = [\,]$ and $A(\ell_i^u\pi) = B_i^u(\pi)$ and $A(\ell_i^s\pi) = B_i^s(\pi)$ for every path $\pi$ and every $i \in I$. Two Split types are equal if they are equal as regular trees. The letters $A$, $B$ and $C$ range over the set of Split types, and this set is denoted by $\mathcal{T}$.

DEFINITION 2.1. [Subtyping] Let $\leq_\Sigma$ be a partial order defined on $\Sigma$ such that $\bot \leq_\Sigma [\,] \leq_\Sigma \top$ and $\bot \leq_\Sigma Q \leq_\Sigma \top$ for every constant type $Q$. The subtype relation over Split types is denoted by $\leq$, and its symmetric relation by $\geq$. In addition, we define $\leq^s$ to be $\leq$ and $\leq^u$ to be $\geq$. If $A$ and $B$ are Split types and $\eta \in \{s, u\}$ then we write $A \leq B$ if and only if

1. $A(\epsilon) \leq_\Sigma B(\epsilon)$ and,
2. if $A(\epsilon) = B(\epsilon) = [\,]$ then $\forall\, \ell^\eta \in \mathsf{Dom}(B) \Rightarrow (\ell^\eta \in \mathsf{Dom}(A) \wedge A{\downarrow}\ell^\eta \leq^\eta B{\downarrow}\ell^\eta)$.

Clearly, $\leq$ is a partial order. In particular, two Split types $A$ and $B$ are equal (as regular trees), if and only if $A \leq B$ and $B \leq A$.

The following lemmas follow directly from the definition of the subtyping relation over Split types shown above.

LEMMA 2.1. *Assume $A \leq B$. Then, for every path $\pi \in \mathsf{Dom}(A) \cap \mathsf{Dom}(B)$ we have: (i) if $\mathsf{parity}(\pi) = 0$ then $A(\pi) \leq_\Sigma B(\pi)$, (ii) if $\mathsf{parity}(\pi) = 1$ then $B(\pi) \leq_\Sigma A(\pi)$.*

LEMMA 2.2. *Assume $A \leq B$. Then, for every path $\pi \in \mathsf{Dom}(A) \cap \mathsf{Dom}(B)$ for which $A(\pi) = B(\pi) = [\,]$ and every $\ell \in \mathcal{L}$ we have: (i) if $\mathsf{parity}(\pi) = 0$ and $\ell^\eta \in \mathsf{Dom}(B{\downarrow}\pi)$ then $\ell^\eta \in \mathsf{Dom}(A{\downarrow}\pi)$, (ii) if $\mathsf{parity}(\pi) = 1$ and $\ell^\eta \in \mathsf{Dom}(A{\downarrow}\pi)$ then $\ell^\eta \in \mathsf{Dom}(B{\downarrow}\pi)$.*

In defining the typing rules, we will find it convenient to introduce a different, albeit equivalent, formulation of subtyping in terms of inference rules: this will ease the comparisons

[1] Since the calculus we work with is functional, method replacement takes place on a copy of the object that is updated instead of on the object itself.

between the systems of Split Types and related type system for the ς-calculus (see Section 4). We first define the structure of typing and subtyping judgements.

### 2.2. Environments and Judgements

A *type environment* is a finite mapping from the set of term variables $\mathsf{Var}$ to the set of Split types. We let $E, E', ...$ range over the set of type environments, and define $\mathsf{Dom}(E) = \{s \mid \exists A.(s : A) \in E\}$ and $\mathsf{Ran}(E) = \{A \mid \exists s.(s : A) \in E\}$. A *subtype environment*, also ranged over by $E, E', \ldots$, is a set of subtyping constraints of the form $A \leq A'$ where $A$ and $A'$ are Split types.

A *type judgement* is a relation between type environments, terms and Split types, written as $E \vdash a : A$. A *subtype judgement* is a relation between subtype environments and Split types, written as $E \vdash A \leq B$. We let $\Im, \Im', ...$ range over typing and subtyping judgements and write $\vdash \Im$ as a shorthand for $\varnothing \vdash \Im$. Additionally, we write $\Im\{s\}$ to emphasize that $s$ may occur free in $\Im$, and $\Im\{\!\{c\}\!\}$ to denote the result of substituting every free occurrence of $s$ in $\Im$ for the term $c$. For conciseness, we often write $E \vdash \Im$ whenever the judgement is derivable and $E \vdash A_1 \leq A_2 \leq A_3 \leq ... \leq A_{n-1} \leq A_n$ whenever $E \vdash A_i \leq A_{i+1}$ is derivable for every $i \in 1..n - 1$.

### 2.3. Typing and Subtyping Rules

The system of Split types or $\mathbf{Ob}^{\downarrow\uparrow}$ is presented in Figure 1. The rules (Sub Object) and (Sub Comps) are part of the axiomatization of the subtyping relation $\leq$ from Definition 2.1. Specifically, if $A = [\ell_i : (B_i^u, B_i^s)^{i \in I}]$ and $A' = [\ell_i : (C_i^u, C_i^s)^{i \in J}]$ then by (Sub Object) and (Sub Comps) we can derive[2]

$$\frac{E \cup \{A \leq A'\} \vdash C_i^u \leq B_i^u \quad E \cup \{A \leq A'\} \vdash B_i^s \leq C_i^s \quad (\forall i \in J \subseteq I)}{E \vdash A \leq A'}$$

which is the co-inductive version of the subtyping rule for recursive types we discussed in the introduction. As anticipated, subtyping over Split types is contravariant in the update components and covariant in the select components. The rule (Sub Hist) allows us to derive any subtyping judgement present in the environment: this is the standard way to allow co-inductive reasoning in derivations of subtyping judgements for recursive types.

In what follows, we shall write $A \leq B$ or $\vdash A \leq B$ interchangeably. To justify that practice, we show that the axiomatization of subtyping given in Figure 1 is sound and complete with respect to Definition 2.1.

PROPOSITION 2.1. *$A \leq B$ if and only if $\vdash A \leq B$.*

*Proof.* **(Only if)** For this part we prove a stronger statement. Namely, if $E \vdash A \leq B$ and for every $C \leq D \in E$ we have $C \leq D$, then $A \leq B$. The proof is by induction on derivations. The cases for (Sub Bot) and (Sub Top) follow immediately from the definition of $\leq$. (Sub Hist) follows from the hypothesis, and (Sub Refl) from the fact that $\leq$ is a partial order, hence reflexive. Finally, (Sub Object) and (Sub Comps) follow by the induction hypothesis. Specifically, if $\ell_j^\eta \in \mathsf{Dom}(B)$ then clearly $\ell_j^\eta \in \mathsf{Dom}(A)$ for $j \in J$, since by the rule (Sub Object) it must be $J \subseteq I$. To show that $A \downarrow \ell_j^\eta \leq^\eta B \downarrow \ell_j^\eta$ for

[2]We could have defined subtyping over object types in terms of this derived rule instead of the two rules of Figure 1: the choice of two rules simplifies the comparison between ours and related object type systems in the literature (see Section 4).

$j \in J$, it suffices to note that $A \downarrow \ell_j^\eta = B_j^\eta$ and $B \downarrow \ell_j^\eta = C_j^\eta$, and then apply the induction hypothesis.

**(If)** We describe a procedure to construct a derivation for $\vdash A \leq B$ using the rules in Figure 1, and then show that the procedure succeeds if $A \leq B$. We use inequalities of the form $(A \leq B)_\pi$ where $\pi$ is a path.

Given $A \leq B$ construct the pair $(H, J)$ where, initially, $H = \varnothing$ and $J = \{ (A \leq B)_\epsilon \}$. Then, apply one of the following rules until (and if) a pair of the form $(H, \varnothing)$ is obtained. If $(C \leq D)_\pi \in J$ then:

1. If $C \leq D \in H$ then apply the procedure to the pair $(H, J \setminus \{ (C \leq D)_\pi \})$ according to (Sub Hist).

2. If $C = \bot$ or $D = \top$ or $C = D$ then apply the procedure to the pair $(H, J \setminus \{ (C \leq D)_\pi \})$ according to (Sub Bot), (Sub Top) or (Sub Refl), respectively.

3. Otherwise, if for every $\ell^\eta \in \mathsf{Dom}(D)$ we have $\ell^\eta \in \mathsf{Dom}(C)$ then apply the procedure to the pair $(H \cup \{C \leq D\}, J \cup \{ (C \downarrow \ell^s \leq D \downarrow \ell^s)_{\pi \ell^s},\ (D \downarrow \ell^u \leq C \downarrow \ell^u)_{\pi \ell^u} \})$, according to (Sub Object) and (Sub Comps).

If none of the rules can be applied and $J \neq \varnothing$ then the procedure fails. Clearly, this procedure always stops since it always checks $H$ before applying the last rule, and Split types (as regular types) contain finitely many different subtrees. In addition, it is easy to verify that the procedure constructs a derivation according the rules in Figure 1: each pair $(H, J)$ can be interpreted as the set of subtyping judgements $\{H \vdash A \leq B \mid (A \leq B)_\pi \in J\}$.

Thus, it suffices to show that if $A \leq B$ then the procedure succeeds. First observe that, given the initial pair $(\varnothing, \{(A \leq B)_\epsilon\})$, any pair $(H, J)$ constructed by the procedure satisfies the condition that if $(C \leq D)_\pi \in J$ then $\pi \in \mathsf{Dom}(A) \cap \mathsf{Dom}(B)$. Now suppose that $A \leq B$ but the procedure gets stuck while constructing a derivation. This happens if (i) $(C \leq \bot)_\pi \in J$ and $C \neq \bot$ or (ii) $(\top \leq D)_\pi \in J$ and $D \neq \top$ or (iii) $(C \leq D)_\pi \in J$ and there exists an $\ell^\eta \in \mathsf{Dom}(D)$ such that $\ell^\eta \notin \mathsf{Dom}(C)$. However, (i) and (ii) contradict Lemma 2.1, and (iii) contradicts Lemma 2.2. Hence, none of these cases may indeed arise and, consequently, given $A \leq B$ we can always construct a derivation for $\vdash A \leq B$. ■

The remaining rules in Figure 1 define the typing rules for terms. (Val Object) is the object-type introduction rule: each method in the object $a$ is typed under the assumption that the self variable $s$ has the same type as $a$. In the type $A$, each of the $B_i^u$'s is the actual type of the method body associated with the $i$'th label, and is also the update component. The corresponding select component, $B_i^s$, can be any supertype of the actual type of the method.

(Val Select) is the object-type elimination rule. If $A$ is an object type, the premises of the rule ensure that the recipient $a$ contains a method for $\ell_j$. Furthermore, the return type of the message is (any supertype of) the type that is currently associated with the select component of $\ell_j$ in the type $A$.

(Val Update) types method overrides. If $A$ is an object type, the premises ensure that $a$ has a method corresponding to $\ell_j$ and that the new method body is then required to have (a subtype of) the type found in the update component of $\ell_j$ in the type $A$.

An interesting aspect of (Val Select) and (Val Update) is that they do not impose any condition on the format of $A$. In particular, $A$ is not required to be an object type. The two rules would at first appear to be *structural* (in the sense of [2]). However, this is not

**Subtyping**

(Sub Object) $$\frac{\begin{array}{c}(A = [\ell_i : (B_i^u, B_i^s)^{i \in I}], \quad A' = [\ell_i : (C_i^u, C_i^s)^{i \in J}]) \\ E \cup \{A \le A'\} \vdash (B_i^u, B_i^s) \le (C_i^u, C_i^s)\ (\forall i \in J \subseteq I)\end{array}}{E \vdash A \le A'}$$

(Sub Comps) $$\frac{E \vdash C^u \le B^u \quad E \vdash B^s \le C^s}{E \vdash (B^u, B^s) \le (C^u, C^s)}$$

(Sub Hist) $$\frac{A \le A' \in E}{E \vdash A \le A'}$$

(Sub Refl) $$\frac{}{E \vdash A \le A}$$

(Sub Top) $$\frac{}{E \vdash A \le \top}$$

(Sub Bot) $$\frac{}{E \vdash \bot \le A}$$

**Typing**

(Val Const) $$\frac{\mathsf{type}(q) = Q}{E \vdash q : Q}$$

(Val Var) $$\frac{E(x) = A}{E \vdash x : A}$$

(Val Select) $$\frac{E \vdash a : A \quad \vdash A \le [\ell_j : (\bot, D)]}{E \vdash a.\ell_j : D}$$

(Val Update) $$\frac{E \vdash a : A \quad E, s : A \vdash b : D \quad \vdash A \le [\ell_j : (D, \top)]}{E \vdash a.\ell_j \Leftarrow \varsigma(s)\, b : A}$$

(Val Object) $$\frac{\begin{array}{c}(A = [\ell_i : (B_i^u, B_i^s)^{i \in I}],\ \ell_i \textit{ distinct}) \\ E, s : A \vdash b_i : B_i^u \quad \vdash B_i^u \le B_i^s \quad (\forall i \in I)\end{array}}{E \vdash [\ell_i = \varsigma(s)\, b_i{}^{i \in I}] : A}$$

(Val Subsume) $$\frac{E \vdash a : A \quad \vdash A \le A'}{E \vdash a : A'}$$

**FIG. 1.** Typing Rules for $\mathbf{Ob}^{\downarrow\uparrow}$.

the case since our subtyping relation is *non-structural* due to the presence of (Sub Bot) and (Sub Top). As a consequence, in both rules the type $A$ can, in fact, be the type $\bot$.

These observations raise the question of whether there really exist terms that can be assigned the type $\bot$ by the typing rules. Such terms do indeed exist: one example is the "undefined" term $\Omega \equiv [\ell = \varsigma(s)s.\ell].\ell$, for which the type $\bot$ can be derived as follows:

$$\frac{\dfrac{s : [\ell : (\bot, \bot)] \vdash s.\ell : \bot}{\vdash [\ell = \varsigma(s)s.\ell] : [\ell : (\bot, \bot)]}\ \text{(Val Object)} \qquad \vdash [\ell : (\bot, \bot)] \le [\ell : (\bot, \bot)]}{\vdash [\ell = \varsigma(s)s.\ell].\ell : \bot}\ \text{(Val Select)}$$

Given $\Omega : \bot$, it is now possible to construct well-typed, and seemingly unsound terms such as $\Omega.\ell'$, where $\ell'$ is some label different from $\ell$. At first, it may appear that evaluating this term will cause a run-time error since the term eventually tries to select the label $\ell'$ from an object that does not have it. At a closer look, however, we can see that the term is not unsound, as the label $\ell'$ will never be selected from $\Omega$. This is because $(i)$ $\Omega$ itself never reduces to an object, and $(ii)$ the reduction relation (see Definition 2.2) requires the receiver of a selection to reduce to an object. We shall return to this point later, after proving a few properties of the type system.

### 2.4. Soundness of the Type System

The first lemma proves some useful properties about the subtyping relation. Proposition 2.2 states that any derivable typing judgement for (closed) objects satisfies an important invariant for the update and select components of method types: specifically, it states that every method does not "advertise" (select component) more structure than it "may have" (update component). Lemmas 2.4 and 2.5 are standard, and functional to the proof of subject reduction.

LEMMA 2.3 (Subtyping).

*1.If* $E \vdash [\ell_i : (B_i^u, B_i^s)^{i \in I}] \leq A$, *then either* $A = \top$ *or* $A = [\ell_i : (C_i^u, C_i^s)^{i \in J}]$ *with* $J \subseteq I$, *and for every* $i \in J$ *we have* $E \vdash C_i^u \leq B_i^u$ *and* $E \vdash B_i^s \leq C_i^s$.

*2.If* $E \vdash A \leq [\ell_i : (C_i^u, C_i^s)^{i \in J}]$, *then either* $A = \bot$ *or* $A = [\ell_i : (B_i^u, B_i^s)^{i \in I}]$ *with* $J \subseteq I$, *and for every* $i \in J$ *we have* $E \vdash C_i^u \leq B_i^u$ *and* $E \vdash B_i^s \leq C_i^s$.

*Proof.* Easy induction on derivations. ■

PROPOSITION 2.2 (Typings). *Assume* $\vdash [\ell_i = \varsigma(s)\, b_i{}^{i \in I}] : A$. *Then either* $A = \top$, *or* $A = [\ell_i : (B_i^u, B_i^s)^{i \in J}]$ *with* $J \subseteq I$, *and for all* $j \in J$ *we have* $\vdash B_j^u \leq B_j^s$.

*Proof.* By induction on the derivation. An inspection of the typing rules shows that the judgement must be derived by (Val Object) followed by a number of subsumption steps. Then the proof follows by Lemma 2.3 and the format of the (Val Object) rule. ■

LEMMA 2.4 (Substitution). *Assume that* $E, x : C, E' \vdash \Im\{x\}$ *and* $E \vdash c : C$. *Then* $E, E' \vdash \Im\{\{c\}\}$.

LEMMA 2.5 (Bound Weakening). *Assume that* $E, x{:}C, E' \vdash \Im\{x\}$ *and* $\vdash C' \leq C$. *Then* $E, x{:}C', E' \vdash \Im\{x\}$.

The reduction relation $\leadsto$ over closed terms is defined below by a straightforward extension of the corresponding relation in [2], to deal with the case of constant terms. A *result* (or value) $v$ is defined to be either a constant or an object.

DEFINITION 2.2. [Reduction]

- $\vdash c \leadsto c$ if $c = [\ell_i = \varsigma(s)\, b_i{}^{i \in I}]$ or $c$ is a constant.
- $\vdash a.\ell_j \leadsto v$ if $\vdash a \leadsto v' \equiv [\ell_i = \varsigma(s)\, b_i{}^{i \in I}]$ and $\vdash b_j\{\{v'\}\} \leadsto v$ for $j \in I$.
- $\vdash a.\ell_j \Leftarrow \varsigma(s)\, b \leadsto [\ell_j = \varsigma(s)\, b, \ell_i = \varsigma(s)\, b_i{}^{i \in I - \{j\}}]$ if $\vdash a \leadsto [\ell_i = \varsigma(s)\, b_i{}^{i \in I}]$ and $j \in I$.

THEOREM 2.1 (Subject Reduction). *Let* $c$ *be a closed term and* $v$ *a result. Suppose* $\vdash c \leadsto v$. *If* $\varnothing \vdash c : C$ *then* $\varnothing \vdash v : C$.

*Proof.* By induction on the derivation $\vdash c \leadsto v$. The cases when $c$ is a constant or an object are immediate, as in both cases $c \equiv v$. The remaining two cases are discussed below.

**(Select).** Suppose $\vdash a.\ell_j \leadsto v$. This must follow from $\vdash a \leadsto v' \equiv [\ell_i = \varsigma(s)\, b_i{}^{i \in I}]$, with $j \in I$, and from $\vdash b_j\{\{v'\}\} \leadsto v$. Assume that $\varnothing \vdash a.\ell_j : C$. This judgement must have been derived as follows:

(Val Select)
$$\dfrac{\varnothing \vdash a : A \quad \vdash A \leq [\ell_j : (\bot, D)]}{\varnothing \vdash a.\ell_j : D}$$
$$\vdots\,(\vdash D \leq C)$$
$$\varnothing \vdash a.\ell_j : C$$

Since $\vdash a \rightsquigarrow v'$ and $\varnothing \vdash a : A$, by induction hypothesis we have $\varnothing \vdash v' : A$. Since $v'$ is in object form, this last judgement must have been derived as shown below for a type $A' = [\ell_i : (B^u_i, B^s_i)^{i \in I}]$.

(Val Object)
$$\dfrac{s : A' \vdash b_i\{s\} : B^u_i : \quad \vdash B^u_i \leq B^s_i \quad (\forall i \in I)}{\varnothing \vdash v' : A'}$$
$$\vdots\,(\vdash A' \leq A)$$
$$\varnothing \vdash v' : A$$

Since $j \in I$, we have $s : A' \vdash b_j\{s\} : B^u_j$. From this judgement, and from $\varnothing \vdash v' : A'$, by Lemma 2.4 it follows that $\varnothing \vdash b_j\{\!\{v'\}\!\} : B^u_j$. By induction hypothesis, we now have $\varnothing \vdash v : B^u_j$. Since $\vdash B^u_j \leq B^s_j$, $\vdash A' \leq A$ and $\vdash A \leq [\ell_j : (\bot, D)]$, by Lemma 2.3 it follows that $\vdash B^u_j \leq B^s_j \leq D$. Since $\vdash D \leq C$, we have $\varnothing \vdash v : C$ by (Val Subsume).

**(Update).** Suppose $\vdash a.\ell_j \Leftarrow \varsigma(s)\, b \rightsquigarrow [\ell_j = \varsigma(s)\, b,\, \ell_i = \varsigma(s)\, b_i\ ^{i \in I - \{j\}}]$. This must be derived from $\vdash a \rightsquigarrow [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}]$ with $j \in I$. Assume that $\varnothing \vdash a.\ell_j \Leftarrow \varsigma(s)\, b : C$. This judgement must have been derived as follows:

(Val Update)
$$\dfrac{\varnothing \vdash a : A \quad \vdash A \leq [\ell_j : (D, \top)] \quad s : A \vdash b : D}{\varnothing \vdash a.\ell_j \Leftarrow \varsigma(s)\, b : A}$$
$$\vdots\,(\vdash A \leq C)$$
$$\varnothing \vdash a.\ell_j \Leftarrow \varsigma(s)\, b : C$$

By induction hypothesis, $\varnothing \vdash [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : A$. Then, for some Split type $A' = [\ell_i : (B^u_i, B^s_i)^{i \in I}]$, we must have:

(Val Object)
$$\dfrac{s : A' \vdash b_i : B^u_i \quad \vdash B^u_i \leq B^s_i \quad (\forall i \in I)}{\varnothing \vdash [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : A'}$$
$$\vdots\,(\vdash A' \leq A)$$
$$\varnothing \vdash [l_i = \varsigma(s)\, b_i\ ^{i \in I}] : A$$

Because $s : A \vdash b : D$ and $\vdash A' \leq A$, by Lemma 2.5 it follows that $s : A' \vdash b : D$. Furthermore, since $\vdash A' \leq A \leq [\ell_j : (D, \top)]$, by Lemma 2.3 we have $\vdash D \leq B^u_j$, and by (Val Subsume) $s : A' \vdash b : B^u_j$. Hence, using (Val Object) we have $\varnothing \vdash [\ell_j = \varsigma(s)\, b, \ell_i = \varsigma(s)\, b_i\ ^{i \in I - \{j\}}] : A'$, and the desired judgement follows from (Val Subsume) and the fact that $\vdash A' \leq A \leq C$. ■

A theorem showing the absence of *stuck states* can easily be derived from subject reduction. We first prove the following lemma.

LEMMA 2.6 (Divergent Terms). *Assume* $\vdash a : \bot$. *Then there exist no value* $v$ *such that* $\vdash a \leadsto v$.

*Proof.* By contradiction. Assume $\vdash a : \bot$ and $\vdash a \leadsto v$ for some value $v$. By subject reduction, we have $\vdash v : \bot$. Given that no constant has type $\bot$, the value $v$ must be an object. Impossible, as this would contradict Proposition 2.2. ■

The reduction rules of Definition 2.2 can directly be used as the definition of an interpreter for the calculus. Run-time errors for this interpreter correspond to pattern-matching failures (i.e., stuck states) when using the rules to evaluate a closed expression. An inspection of the rules shows that there are two situations which may cause an evaluation to get stuck: given $a.\ell$ (similarly, $a.\ell \Leftarrow \varsigma(s)\, b$) either $(i)$ $a$ evaluates to a value that is not an object, or $(ii)$ $a$ evaluates to an object that does not have $\ell$.

The following theorem proves the absence of such errors in the evaluation of a well-typed closed expression: type soundness follows from this result.

THEOREM 2.2 (Absence of Stuck States). *Let* $a$ *be a closed term for which we have* $\varnothing \vdash a : A$ *for some type* $A$. *Then:*

1. *if* $a = a'.\ell$ *and* $a' \leadsto r$, *then* $r = [\ldots, \ell = \varsigma(s)\, b, \ldots]$ *for some term* $b$,
2. *if* $a = a'.\ell \Leftarrow \varsigma(s)\, b'$ *and* $a' \leadsto r$, *then* $r = [\ldots, \ell = \varsigma(s)\, b, \ldots]$ *for some term* $b$.

*Proof.* We prove *1*, the proof of *2* is essentially the same. Given the shape of $a$, the judgement $\varnothing \vdash a : A$ must have been derived by (a number of subsumption steps followed by) an instance of (Val Select) from $\varnothing \vdash a' : A'$, with $\vdash A' \le [\ell : (\bot, A)]$. By Lemma 2.3, $A'$ is either $\bot$ or an object type containing the label $\ell$. By Subject Reduction, we know that $\varnothing \vdash r : A'$. Since $r$ is a result, (the contrapositive of) Lemma 2.6 implies that $A$ may not be $\bot$. Now, an inspection of the typing rules shows that $a'$ must be an object of the form $[\ldots, \ell = \varsigma(s)\, b, \ldots]$ as desired. ■

## 2.5. The Lattice of Split Types

We have already noted that variant subtyping induces a rich subtype hierarchy for Split types. In fact, this hierarchy turns out to be a lattice with $\bot$ and $\top$ as bottom and top elements. Least upper bounds and greatest lower bounds in this lattice can be defined by representing Split types as term automata, as suggested in [18]. Based on that representation we can prove the following proposition.

PROPOSITION 2.3 (Lubs and Glbs). *There exist two operators* $\sqcup$ *and* $\sqcap$ *such that:*

1. *For every type* $A$*:*
   - $\bot \sqcup A = A$, $\top \sqcup A = \top$,
   - $\top \sqcap A = A$, $\bot \sqcap A = \bot$,
2. *For every* $A = [\ell_i : (B_i^u, B_i^s)^{i \in I}]$ *and* $A' = [\ell_i : (C_i^u, C_i^s)^{i \in J}]$*:*
   - $A \sqcup A' = [\ell_k : (B_k^u \sqcap C_k^u, B_k^s \sqcup C_k^s)^{k \in I \cap J}]$,
   - $A \sqcap A' = [\ell_k : (B_k^u \sqcup C_k^u, B_k^s \sqcap C_k^s)^{k \in I \cap J},$
     $\ell_m : (B_m^u, B_m^s)^{m \in I - J},\ \ell_n : (C_n^u, C_n^s)^{n \in J - I}]$.

The presence of a lattice structure is a distinctive property of Split types, that does not have a counterpart in the first-order types systems of [2]. Specifically, greatest lower bounds do not exist for those systems. For example, due to invariant subtyping, the two types $[\ell : [\,]]$ and $[\ell : [\ell : [\,]]]$ have no common lower bound.

Least upper bounds, instead, do exist for finite and recursive object types, but they are "less informative" than least upper bounds of Split types. As a consequence, Split types provide typings for terms that fail to type check with recursive object types.

EXAMPLE 2.1. Consider the terms from Example 1.3.

$$\begin{aligned} p_2 &= [x = \varsigma(s)s.\mathit{move}.y, y = 0, \mathit{move} = \varsigma(s)s.y := s.y + 1] \\ p_0 &= [\mathit{move} = \varsigma(s)s] \\ p &= [\ell = p_2].\ell := p_0 \end{aligned}$$

Given these terms, we have shown that $p.\ell.\mathit{move}$ is not typable with recursive types, as the most informative type that can be assigned to $p$, is $[\ell : [\,]]$. With Split types, instead, we have:

$$\begin{aligned} p_2 &: P_2 && \textit{where } P_2 = [x : (\mathsf{int}, \mathsf{int}), y : (\mathsf{int}, \mathsf{int}), \mathit{move} = (P_2, P_2)] \\ p_0 &: P_0 && \textit{where } P_0 = [\mathit{move} : (P_0, P_0)] \\ p &: [\ell : (P, P)] && \textit{where } P = [\mathit{move} : (P_2 \sqcap P_0, P)] \end{aligned}$$

The typings $p_2 : P_2$ and $p_0 : P_0$ are derived by a routine application of the typing rules. As for the term $p$, observe that in order for the update to type check, we need to find a common super-type for $P_2$ and $P_0$. The typing $p : [\ell : (P, P)]$ arises as a consequence of this constraint, as $P = P_0 \sqcup P_2$. From $p : [\ell : (P, P)]$, one derives $p.\ell : [\mathit{move} : (P_2 \sqcap P_0, P)]$, and then $p.\ell.\mathit{move} : P$.

## 3. TYPE INFERENCE

In this section, we present an algorithm that infers type information for untyped $\mathbf{Ob}^{\downarrow\uparrow}$ terms. Following a common practice, the algorithm works by reducing the problem of finding a type derivation for a term to the problem of solving a set of subtyping constraints. The types involved in the reduction are the *inference types* defined by the following productions:

$$\sigma, \tau \in \mathcal{I} ::= \alpha \mid Q \mid [\ell_i : (\alpha_i, \beta_i)^{\,i \in I}]$$

We use Greek letters towards the beginning of the alphabet such as $\alpha, \beta, \ldots$ to range over a set of type variables $\mathsf{TVar}$, and Greek letters towards the end of the alphabet such as $\sigma, \tau, \ldots$ to range over the set of inference types $\mathcal{I}$. For every inference type $\tau$ we define $\mathsf{FV}(\tau)$ as the set of type variables occurring in $\tau$.

A *substitution* $\rho$ is a mapping from the set of type variables $\mathsf{TVar}$ to the set of $\mathcal{T}$ of Split types. We only need to consider substitutions with finite domains. The domain of a substitution $\rho$ is denoted by $\mathsf{Dom}(\rho)$. Any substitution $\rho$ can be lifted to a mapping from $\mathcal{I}$ to $\mathcal{T}$ in the standard way: to simplify the notation, we refer to both a substitution and its lifting by the same letter, typically $\rho$.

A *constraint* is a pair of inference types $\sigma$ and $\tau$ written as $\sigma \leq \tau$. We use the same symbol $\leq$ to denote both a constraint and the subtyping relation defined in Figure 1. The symbol $\vdash$ used as a prefix distinguishes a provable subtyping relation from a constraint.

**(I-Val Const)**: $\mathsf{type}(q) = Q$

$$(\mathsf{J} \cup \{\Gamma \triangleright q : \alpha\}, \mathsf{C}) \quad\Longrightarrow\quad (\mathsf{J}, \mathsf{C} \cup \{Q \leq \alpha\})$$

**(I-Val Var)**: $\Gamma(x) = A$

$$(\mathsf{J} \cup \{\Gamma \triangleright x : \alpha\}, \mathsf{C}) \quad\Longrightarrow\quad (\mathsf{J}, \mathsf{C} \cup \{A \leq \alpha\})$$

**(I-Val Select)**: *$\beta$ and $\gamma$ fresh*

$$(\mathsf{J} \cup \{\Gamma \triangleright a.\ell_j : \alpha\}, \mathsf{C}) \quad\Longrightarrow\quad (\mathsf{J} \cup \{\Gamma \triangleright a : \beta\}, \mathsf{C} \cup \{\beta \leq [\ell_j : (\gamma, \alpha)],\ \gamma \leq \alpha\})$$

**(I-Val Update)**: *$\beta$, $\gamma$ and $\delta$ fresh*

$$(\mathsf{J} \cup \{\Gamma \triangleright a.\ell_j \Leftarrow \varsigma(s)b : \alpha\}, \mathsf{C}) \quad\Longrightarrow\quad \begin{pmatrix} \mathsf{J} \cup \{\Gamma \triangleright a : \gamma,\ \Gamma, s : \gamma \triangleright b : \beta\}, \\ \mathsf{C} \cup \{\gamma \leq \alpha,\ \gamma \leq [\ell_j : (\beta, \delta)],\ \beta \leq \delta\} \end{pmatrix}$$

**(I-Val Object)**: *$\beta_i$ and $\gamma_i$ fresh*

$$(\mathsf{J} \cup \{\Gamma \triangleright [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : \alpha\}, \mathsf{C}) \quad\Longrightarrow\quad \begin{pmatrix} \mathsf{J} \cup \{\Gamma, s : [\ell_i : (\beta_i, \gamma_i)\ ^{i \in I}] \triangleright b_i : \beta_i\}^{i \in I}, \\ \mathsf{C} \cup \{[\ell_i : (\beta_i, \gamma_i)\ ^{i \in I}] \leq \alpha,\ \beta_i \leq \gamma_i\}^{i \in I} \end{pmatrix}$$

**FIG. 2.** Inference Rules.

For every constraint $\sigma \leq \tau$ define $\mathsf{FV}(\sigma \leq \tau) = \mathsf{FV}(\sigma) \cup \mathsf{FV}(\tau)$. If $\mathsf{C}$ is a constraint set, then $\mathsf{Dom}(\mathsf{C}) = \{\alpha \mid \alpha \leq \tau \in \mathsf{C} \text{ or } \tau \leq \alpha \in \mathsf{C}\}$ and $\mathsf{FV}(\mathsf{C}) = \cup_{r \in \mathsf{C}} \mathsf{FV}(r)$.

DEFINITION 3.1. [Constraint Solvability] Let $\mathsf{C}$ be a constraint set and $\rho$ be a substitution. We say that $\rho$ is a solution to $\mathsf{C}$ and write $\rho \models \mathsf{C}$, if $\mathsf{Dom}(\rho) \supseteq \mathsf{FV}(\mathsf{C})$ and for every constraint $\sigma \leq \tau$ in $\mathsf{C}$ one has $\vdash \rho(\sigma) \leq \rho(\tau)$. We say that a constraint set $\mathsf{C}$ is solvable if there exists a substitution $\rho$ such that $\rho \models \mathsf{C}$.

### 3.1. Generating Constraints

The type inference algorithm collects a set of subtyping constraints generated by the inference rules in figure 2: these rules implement the algorithmic version of the typing rules of Section 2, obtained by removing the subsumption rule and "plugging" it into the remaining rules when needed. The inference rules are formulated as rewriting rules for pairs of the form $(\mathsf{J}, \mathsf{C})$, where $\mathsf{J}$ is a set of judgements $\Gamma \triangleright a : \alpha$ and $\mathsf{C}$ is a set of constraints.

DEFINITION 3.2. [Rewriting] The transformation from rules to constraints is accomplished by an initialization step, followed by zero or more iteration steps.

*Init.* Form the initial pair $(\{\Gamma \triangleright a : \alpha\}, \varnothing)$, where $\alpha$ is a fresh type variable and $\Gamma$ an environment mapping the free variables of $a$ to fresh type variables.

*Iterate.* Let $(\mathsf{J}, \mathsf{C})$ be the current pair. If $\mathsf{J}$ is empty, then stop. Otherwise, select a judgement from $\mathsf{J}$, rewrite it using the appropriate rule from Figure 2 and repeat this step.

PROPOSITION 3.1 (Termination). *The rewriting process from Definition 3.2 always terminates.*

*Proof.* The proof follows easily by using the measure on $(\mathsf{J}, \mathsf{C})$ pairs defined in Figure 3. Defining $|(\mathsf{J}, \mathsf{C})| = \Sigma_{\Im \in \mathsf{J}} |\Im|$, the claim follows by observing that $|(\mathsf{J}, \mathsf{C})|$ strictly decreases after each step of the rewriting process and it is bound from below by $0$.

$$\begin{array}{rcl} |\Gamma \triangleright q : \alpha| & = & 1 \\ |\Gamma \triangleright x : \alpha| & = & 1 \\ |\Gamma \triangleright a.\ell_j : \alpha| & = & |\Gamma \triangleright a : \beta| + 1 \\ |\Gamma \triangleright a.\ell_j \Leftarrow \varsigma(s)b : \alpha| & = & |\Gamma \triangleright a : \gamma| + |\Gamma, s : \gamma \triangleright b : \beta| + 1 \\ |\Gamma \triangleright [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : \alpha| & = & \Sigma_{i \in I} |\Gamma, s : [\ell_i : (\beta_i, \gamma_i)\ ^{i \in I}] \triangleright b_i : \beta_i| + 1 \end{array}$$

**FIG. 3.** Measure on $(\mathsf{J}, \mathsf{C})$ pairs.

Note, further, that the rewriting always terminates with a pair $(\varnothing, \mathsf{C})$. To see that, observe that the only possibility for the rewriting to get stuck is when the selected judgement is $\Gamma \triangleright x : \alpha$ and $x \notin \mathsf{Dom}(\Gamma)$. This cannot happen, however, as $\mathsf{FV}(a) \subseteq \mathsf{Dom}(\Gamma)$ by construction, and an inspection of the rewriting rules shows that whenever $(\Gamma' \triangleright a' : \tau) \in \mathsf{J}$ we have $\mathsf{FV}(a') \subseteq \mathsf{Dom}(\Gamma')$. ■

Next, we show that the rewriting described in Definition 3.2 is sound and complete. That is, that solving constraints is equivalent to finding type derivations. The proof uses the following generation lemmas about the type system.

LEMMA 3.1 (Generation Lemmas).

*1.If $E \vdash x : B$, then $E(x) = A$ where $A$ is a type such that $\vdash A \leq B$.*

*2.If $E \vdash a.\ell : B$, then $E \vdash a : A$ for some type $A$ such that $\vdash A \leq [\ell : (\bot, B)]$.*

*3.If $E \vdash a.\ell \Leftarrow \varsigma(s)\, b : A$, then there exist types $A'$ and $B$ such that $\vdash A' \leq [\ell : (B, \top)]$ and $\vdash A' \leq A$, and also $E \vdash a : A'$ and $E, s : A' \vdash b : B$.*

*4.If $E \vdash [\ell_i = \varsigma(s)\, b_i{}^{i \in I}] : A$, then there exist a type $A' = [\ell_i : (B_i^u, B_i^s)^{i \in I}]$ such that $\vdash A' \leq A$, and also $E, s : A' \vdash b_i : B_i^u$ and $\vdash B_i^u \leq B_i^s$ for every $i \in I$.*

*Proof.* By induction on the derivation of the judgement in question. ■

DEFINITION 3.3. [Pair Satisfaction] We say that $\rho$ satisfies a pair $(\mathsf{J}, \mathsf{C})$, written as $\rho \models (\mathsf{J}, \mathsf{C})$, if $\rho \models \mathsf{C}$ and for every $\Gamma \triangleright a : \alpha$ in $\mathsf{J}$ the judgement $\rho(\Gamma) \vdash a : \rho(\alpha)$ is derivable in $\mathbf{Ob}^{\downarrow\uparrow}$.

LEMMA 3.2 (Rewriting is Sound). *Assume* $(\mathsf{J}, \mathsf{C}) \Longrightarrow (\mathsf{J}', \mathsf{C}')$. *Every substitution $\rho$ that satisfies* $(\mathsf{J}', \mathsf{C}')$ *also satisfies* $(\mathsf{J}, \mathsf{C})$.

*Proof.* By case analysis on the rewriting step.

**(I-Val Const)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J}, \mathsf{C} \cup \{Q \leq \alpha\})$. Clearly, $\rho \models (\mathsf{J}, \mathsf{C})$ and $\vdash \rho(Q) \leq \rho(\alpha)$. Since $\mathsf{type}(q) = Q$ and $\rho(Q) = Q$ for any type $Q$, it follows by (Val Const) that $\rho(\Gamma) \vdash q : \rho(Q)$ and by (Val Subsume) that $\rho(\Gamma) \vdash q : \rho(\alpha)$. Consequently, we have $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright q : \alpha\}, \mathsf{C})$.

**(I-Val Var)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J}, \mathsf{C} \cup \{A \leq \alpha\})$. Clearly, $\rho \models (\mathsf{J}, \mathsf{C})$ and $\vdash \rho(A) \leq \rho(\alpha)$. Since $\Gamma(x) = A$, it follows by (Val Var) that $\rho(\Gamma) \vdash x : \rho(A)$ and by (Val Subsume) that $\rho(\Gamma) \vdash x : \rho(\alpha)$. Consequently, we have $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright x : \alpha\}, \mathsf{C})$.

**(I-Val Select)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a : \beta\}, \mathsf{C} \cup \{\beta \leq [\ell_j : (\gamma, \alpha)], \gamma \leq \alpha\})$. We have $\rho \models \mathsf{C}$, $\rho(\Gamma) \vdash a : \rho(\beta)$, and also $\vdash \rho(\beta) \leq [\ell_j : (\rho(\gamma), \rho(\alpha))]$. From the last subtyping judgement, we obtain $\vdash \rho(\beta) \leq [\ell_j : (\bot, \rho(\alpha))]$, as $\bot \leq \rho(\gamma)$. Therefore, it follows by (Val Select) that $\rho(\Gamma) \vdash a.\ell_j : \rho(\alpha)$. Consequently, we have $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a.\ell_j : \alpha\}, \mathsf{C})$.

**(I-Val Update)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a : \gamma,\ \Gamma, s : \gamma \triangleright b : \beta\}, \mathsf{C} \cup \{\gamma \leq \alpha, \gamma \leq [\ell_j : (\beta, \delta)]\}, \delta \leq \top)$. Clearly, $\rho \models \mathsf{C}$ and $\vdash \rho(\gamma) \leq [\ell_j : (\rho(\beta), \top)]$, and also the judgements $\rho(\Gamma) \vdash a : \rho(\gamma)$ and $\rho(\Gamma), s : \rho(\gamma) \vdash b : \rho(\beta)$ are derivable. Therefore, it follows by (Val Update) that $\rho(\Gamma) \vdash a.\ell \Leftarrow \varsigma(s)b : \rho(\beta)$. Consequently, we have $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a\,.\,\ell \Leftarrow \varsigma(s)b : \alpha\}, \mathsf{C})$.

**(I-Val Object)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma, s : [\ell_i : (\beta_i, \gamma_i)\ ^{i \in I}] \triangleright b_i : \beta_i\}^{i \in I}, \mathsf{C} \cup \{[\ell_i : (\beta_i, \gamma_i)\ ^{i \in I}] \leq \alpha,\ \beta_i \leq \gamma_i\}^{i \in I})$. Clearly, $\vdash [\ell_i : (\rho(\beta_i), \rho(\gamma_i))\ ^{i \in I}] \leq \rho(\alpha)$ and $\vdash \rho(\beta_i) \leq \rho(\gamma_i)$, and also the judgements $\rho(\Gamma), s : [\ell_i : (\rho(\beta_i), \rho(\gamma_i))\ ^{i \in I}] \vdash b_i : \rho(\beta_i)$ are derivable. Therefore, it follows by (Val Object) and by (Val Subsume) that $\rho(\Gamma) \vdash [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : \rho(\alpha)$. Consequently, we have $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : \alpha\}, \mathsf{C})$. ■

LEMMA 3.3 (Rewriting is Complete). *Assume* $(\mathsf{J}, \mathsf{C}) \Longrightarrow (\mathsf{J}', \mathsf{C}')$. *For every substitution* $\rho$ *that satisfies* $(\mathsf{J}, \mathsf{C})$*, there exist substitutions* $\rho'$ *and* $\rho''$ *such that* $\rho' = \rho'' \circ \rho$ *and* $\mathit{Dom}(\rho'') \cap \mathit{Dom}(\rho) = \varnothing$ *and* $\rho'$ *satisfies* $(\mathsf{J}', \mathsf{C}')$.

*Proof.* By a case analysis on the rewriting step.

**(I-Val Const)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright q : \alpha\}, \mathsf{C})$. Clearly, $\rho \models (\mathsf{J}, \mathsf{C})$ and $\rho(\Gamma) \vdash q : \rho(\alpha)$. Therefore, if $\mathsf{type}(q) = Q$ then it must be $\vdash Q \leq \rho(\alpha)$. Consequently, since $\rho(Q) = Q$ for any type $Q$, we have $\rho' = \rho$ and $\rho' \models (\mathsf{J}, \mathsf{C} \cup \{Q \leq \alpha\})$.

**(I-Val Var)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright x : \alpha\}, \mathsf{C})$. Clearly, $\rho(\Gamma) \vdash x : \rho(\alpha)$. By Lemma 3.1.1., $(\rho(\Gamma))(x) = A$ for some type $A$ such that $\vdash A \leq \rho(\alpha)$. Consequently, we have $\rho' = \rho$ and $\rho' \models (\mathsf{J}, \mathsf{C} \cup \{A \leq \alpha\})$.

**(I-Val Select)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a.\ell : \alpha\}, \mathsf{C})$. Clearly, $\rho(\Gamma) \vdash a.\ell : \rho(\alpha)$ and by Lemma 3.1.2. $\rho(\Gamma) \vdash a : A$ is also derivable for some type $A$ such that $\vdash A \leq [\ell : (\bot, \rho(\alpha))]$. Let $\rho'' = \{\beta \mapsto A, \gamma \mapsto \bot\}$ where $\beta$ and $\gamma$ are the fresh variables chosen by the rewriting step. As a result, it follows by construction that $\rho' \models (\mathsf{J}', \mathsf{C}')$.

**(I-Val Update)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a.\ell \Leftarrow \varsigma(s)b : \alpha\}, \mathsf{C})$. Clearly, $\rho(\Gamma) \vdash a.\ell \Leftarrow \varsigma(s)b : \rho(\alpha)$ and by Lemma 3.1.3. $\rho(\Gamma) \vdash a : A'$ and $\rho(\Gamma), s : A' \vdash b : B$ for some types type $A'$ and $B$ such that $\vdash A' \leq [\ell : (B, \top)]$ and $\vdash A' \leq \rho(\alpha)$. Let $\rho'' = \{\alpha \mapsto A', \beta \mapsto B, \delta \mapsto \top\}$ where $\alpha$, $\beta$ and $\delta$ are the fresh variables chosen by the rewriting step. As a result, it follows by construction that $\rho' \models (\mathsf{J}', \mathsf{C}')$.

**(I-Val Object)** Let $\rho$ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : \alpha\}, \mathsf{C})$. Clearly, $\rho(\Gamma) \vdash [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : \rho(\alpha)$ and by Lemma 3.1.4. $\rho(\Gamma), s : [\ell_i : (B_i^u, B_i^s)\ ^{i \in I}] \vdash b_i : B_i^u$ and $\vdash B_i^u \leq B_i^s$ for $i \in I$. Let $\rho'' = \{\beta_i \mapsto B_i^u, \gamma_i \mapsto B_i^s\}^{i \in I}$ where $\beta_i$ and $\gamma_i$ are the fresh variables chosen by the rewriting step. As a result, it follows by construction that $\rho' \models (\mathsf{J}', \mathsf{C}')$. ■

THEOREM 3.1 (Rewriting is Sound and Complete). *Let* $a$ *be a term and* $\Gamma$ *a type environment* $\Gamma$ *such that* $\mathit{Dom}(\Gamma) = \mathit{FV}(a)$. *If* $(\{\Gamma \triangleright a : \alpha\}, \varnothing) \Longrightarrow^* (\varnothing, \mathit{C})$*, then for every substitution* $\rho$ *such that* $\rho \models \mathit{C}$*, the judgement* $\rho(\Gamma) \vdash a : \rho(\alpha)$ *is derivable in* $\mathbf{Ob}^{\downarrow\uparrow}$. *Conversely, if* $E \vdash a : A$ *is derivable in* $\mathbf{Ob}^{\downarrow\uparrow}$*, and* $\mathit{Dom}(E) = \mathit{FV}(a)$*, then there exist a set of constraints* $\mathit{C}$ *such that* $(\{\Gamma \triangleright a : \alpha\}, \varnothing) \Longrightarrow^* (\varnothing, \mathit{C})$ *and a substitution* $\rho$ *such that* $\rho \models \mathit{C}$ *and* $E = \rho(\Gamma)$ *and* $A = \rho(\alpha)$.

*Proof.* Take a substitution $\rho \models \mathsf{C}$. By definition, $\rho \models (\varnothing, \mathsf{C})$, and by Lemma 3.2 (and transitivity) $\rho \models (\{\Gamma \triangleright a : \alpha\}, \varnothing)$. Hence $\rho(\Gamma) \vdash a : \rho(\alpha)$ is derivable, as desired. Conversely, take $E \vdash a : A$ as in the hypothesis, $\Gamma$ and $\alpha$ as specified by the algorithm, and define a substitution $\rho$ as follows: $\rho(\alpha) = A$, and $\rho(\Gamma(x)) = E(x)$ for every $x \in \mathsf{Dom}(E)$. Then $E = \rho(\Gamma)$ and $A = \rho(\alpha)$ by construction, and clearly $\rho \models (\{\Gamma \triangleright a : \alpha\}, \varnothing)$, as

$E \vdash a : A$ is derivable by hypothesis. By Proposition 3.1, the rewriting terminates in a final state of the form $(\varnothing, \mathsf{C})$. Finally, $\rho \models (\varnothing, \mathsf{C})$, by Lemma 3.3, and hence $\rho \models \mathsf{C}$. ■

### 3.2. Solving Constraints

The method we adopt for deciding constraint solvability is inspired by the corresponding method presented by Palsberg in [15].

DEFINITION 3.4. [Constraint System] A constraint set $\mathsf{C}$ is a *constraint system* if and only if for every constraint $\alpha \leq A \in \mathsf{C}$, either $A = \alpha$ or $\alpha \notin \mathsf{FV}(A)$.

PROPOSITION 3.2 (Rewriting vs Constraint Systems). *If* $(\{\Gamma \triangleright a{:}\alpha\}, \varnothing) \Longrightarrow^* (\varnothing, \mathsf{C})$ *then the constraint set* $\mathsf{C}$ *is a constraint system.*

*Proof.* By an inspection of the rewriting rules in Figure 2. ■

DEFINITION 3.5. [Constraint Graph] A *constraint graph* is a directed graph $G = (N, S \cup Q, L, \leq)$ consisting of two disjoint sets of directed edges $\leq$ and $L$, and three disjoint sets of nodes, $N$, $S$, and $Q$. Each edge in $\leq$ is labeled by $\leq$. Each edge in $L$ is labeled by either $\ell^u$ or $\ell^s$ for $\ell \in \mathcal{L}$; in addition, no cycle in the graph goes through an $L$ edge. The nodes of a constraint graph satisfy the following properties:

1. $S$ and $Q$ nodes have *no* outgoing $L$ edges,

2. $N$ nodes have finitely many outgoing $L$ edges, all to $S$ nodes, and those edges have distinct labels and are incident to different $S$ nodes. Furthermore, for every node $n \in N$, the following two conditions are satisfied:

(i) $n \stackrel{\ell^u}{\rightarrow} p \in G$ if and only if $n \stackrel{\ell^s}{\rightarrow} q \in G$ for every $\ell \in \mathcal{L}$
(ii) if $n \stackrel{\ell^u}{\rightarrow} p$ and $n \stackrel{\ell^s}{\rightarrow} q$ are in $G$, then also $p \stackrel{\leq}{\rightarrow} q$ is in $G$.

DEFINITION 3.6. [Solution of constraint graph] Let $G$ be a constraint graph. For each map $h : S \rightarrow \mathcal{T}$, define $\widehat{h} : (N \cup S \cup Q) \rightarrow \mathcal{T}$ as follows:

$$\widehat{h}(p) \;=\; \begin{cases} [\ell_i : (h(q_i), h(r_i))^{i \in I}] & \text{if } p \stackrel{\ell_i^u}{\rightarrow} q_i \text{ and } p \stackrel{\ell_i^s}{\rightarrow} r_i \text{ are the edges from } p \in N, \\ h(p) & \text{if } p \in S, \\ p & \text{if } p \in Q. \end{cases}$$

We say that $h : S \rightarrow \mathcal{T}$ is a *solution* to $G$ if for every $p \stackrel{\leq}{\rightarrow} q$ in $G$ we have $\widehat{h}(p) \leq \widehat{h}(q)$.

THEOREM 3.2. *Solving constraint graphs is equivalent to solving constraint systems.*

*Proof.* Given a constraint system $\mathsf{C}$, we construct a constraint graph as follows. Associate a unique $N$ node with every inference type $[\ell_i : (\alpha_i, \beta_i)^{i \in I}]$, a unique $S$ node with every type variable in $\mathsf{C}$, and a unique $Q$ node with all the occurrences of a primitive type in $\mathsf{C}$. From each $N$ node associated with $[\ell_i : (\alpha_i, \beta_i)^{i \in I}]$, define an $L$ edge labeled $\ell_i^u$ to $\alpha_i$, and an $L$ edge labeled $\ell_i^s$ to $\beta_i$. Finally, define the $\leq$ edges corresponding to the inequalities, in the obvious way. Clearly, the resulting graph is a constraint graph, which is solvable if and only if so is the constraint system. ■

DEFINITION 3.7. [Closure of constraint graphs] A constraint graph is *closed* if the edge relation $\leq$ is reflexive, transitive, and closed under the following rule that says that the dash edges exist whenever the solid ones do

$$
\begin{tikzcd}
\cdot \arrow[r, "\leq"] \arrow[d, "\ell^s"'] & \cdot \arrow[d, "\ell^s"] \\
\cdot \arrow[r, dashed, "\leq"] & \cdot
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\cdot \arrow[r, "\leq"] \arrow[d, "\ell^u"'] & \cdot \arrow[d, "\ell^u"] \\
\cdot & \cdot \arrow[l, dashed, "\geq"']
\end{tikzcd}
$$

Clearly, the closure of a constraint graph is again a constraint graph. Also, it is easy to verify that a constraint graph and its closure have the same set of solutions. To see that, note that any solution to the closure of a graph $G$ is also a solution of $G$ since $G$ has fewer constraints. The converse follows by the definition of $\leq$.

Next we introduce a notion of well-formedness for constraint graphs, by extending the corresponding definition for the AC-graphs of [15].

DEFINITION 3.8. [Well-formed constraint graph] A constraint graph is *well-formed* if and only it satisfies all of the following conditions:

- $\mathbf{W_1}$: for all nodes $p, q \in N$ with $p \overset{\leq}{\to} q$, if $q$ has an outgoing edge labeled $\ell^\eta$, then so does $p$;
- $\mathbf{W_2}$: there is no edge $p \overset{\leq}{\to} q$ with $p \in N$ and $q \in Q$, or with $p \in Q$ and $q \in N$;
- $\mathbf{W_3}$: there is no edge $p \overset{\leq}{\to} q$ with $p, q \in Q$ and $p \neq q$.

As defined, the notion of well-formedness presupposes that no subtyping is available over primitive types. Clearly, the definition can easily be extended to handle the desired subtyping relationships. For instance, had $\mathsf{int} \leq \mathsf{real}$ been allowed, condition $\mathbf{W_3}$ would have been rewritten as: if $p \overset{\leq}{\to} q \in G$ then $p = \mathsf{int}$ and $q = \mathsf{real}$.

In Theorem 3.3, we will show that closed constraint graphs are solvable if and only if they are well-formed. In that direction, we introduce the definition of a *canonical substitution* associated with a constraint graph. First we define the *depth* of a split type $A$, written $\mathsf{depth}(A)$, as the length of the longest path in $\mathsf{Dom}(A)$. Then we define the depth of a substitution $h : S \to \mathcal{T}$ by stipulating that $\mathsf{depth}(h) = \mathit{max}\{\mathsf{depth}(h(s)) \mid s \in \mathsf{Dom}(h)\}$.

DEFINITION 3.9. [Canonical substitution] Let $G$ be a closed constraint graph. For every $s \in S$, define the set $G^\uparrow(s) = \{\, p \in N \cup Q \mid s \overset{\leq}{\to} p \in G \,\}$. We define *canonical* substitution for $G$ any substitution $h_G : S \to \mathcal{T}$ that satisfies the following equation, for every $s \in S$:

$$
h_G(s) \;=\; \begin{cases} \top & \text{if } G^\uparrow(s) = \varnothing, \\ \sqcap \{\, \widehat{h_G}(p) \mid p \in G^\uparrow(s) \,\} & \text{otherwise,} \end{cases}
$$

where $\widehat{h}_G$ is the lifting of $h_G$ as introduced in Definition 3.6.

PROPOSITION 3.3. *If $G$ is a constraint graph, then a canonical substitution $h_G$ exists, and it is finite, i.e. it has finite depth.*

*Proof.* We give a constructive method for computing a canonical substitution. The construction relies on the following inductive definition of the family of substitutions

$\{h_i\}_{i \geqslant 0}$. For all states $s \in S$, define:

$$\begin{array}{lcl} h_0(s) & = & \top \\ h_{i+1}(s) & = & \sqcap \{ \widehat{h_i}(p) \mid p \in G^{\uparrow}(s) \} \quad \text{for } i > 0 \end{array}$$

where $\widehat{h_i}$ is the lifting of $h_i$ as introduced in Definition 3.6. Clearly, for every $i \geqslant 0$, $h_i$ is well defined and finite. To conclude the proof, we only have to show that there exists a $k$ such that $h_{k+1} = h_k$. If such $k$ exists, then clearly the substitution $h_k$ satisfies the recursive equation in the statement of the proposition. Hence we can choose $h_G = h_k$.

To show that $k$ exists we reason as follows. Given an $S$ node $s$ in $G$, consider the string formed by concatenating the labels of the $L$ edges traversed along a path from $s$ in $G$: call that string an *$L$-path from $s$*, and let $L(s)$ be the set of $L$-paths from $s$. Now define the depth of a node $s$ to be the length of the longest $L$-path in $L(s)$, and the depth of the graph $G$ —written $|G|$— to be the maximum depth of an $s$ node of $G$. Since $G$ is a constraint graph, $|G|$ is clearly finite. Furthermore, an inductive proof shows that for every $i \geqslant 0$, one has $\mathsf{depth}(h_i) \leqslant \mathsf{depth}(h_{i+1})$ and $\mathsf{depth}(h_i) \leqslant |G|$. The existence of the desired $k$ follows directly from the last two properties. ■

We can now show that any canonical substitution associated with a closed constraint graph $G$ is, in fact, a solution to $G$ provided that $G$ is well-formed. We first need the following lemma.

LEMMA 3.4. *Let $G$ be a closed constraint graph. If $p \overset{\leq}{\Rightarrow} q \in G$ and $p \in S$ then $\widehat{h_G}(p) \leq \widehat{h_G}(q)$.*

*Proof.* If $q \in S$ and $p \overset{\leq}{\Rightarrow} q \in G$ then, by definition, $G^{\uparrow}(q) \subseteq G^{\uparrow}(p)$, and consequently $\sqcap G^{\uparrow}(p) \leq \sqcap G^{\uparrow}(q)$. If $q \in N \cup Q$ and $p \overset{\leq}{\Rightarrow} q \in G$, then $\widehat{h_G}(q) \in G^{\uparrow}(p)$ and, consequently, $\sqcap G^{\uparrow}(p) \leq \widehat{h_G}(q)$. ■

THEOREM 3.3. *A closed constraint graph is solvable if and only if it is well-formed.*

*Proof.* Let $G$ be a closed constraint graph. Clearly, if $G$ is solvable then it is well-formed. For the converse, assume that $G$ is well-formed. We show that $h_G$ is a solution to $G$. Let $p \overset{\leq}{\Rightarrow} q \in G$: we argue by cases, depending on whether $p$ and $q$ are in the sets $N$, $S$ or $Q$.

$p \in S$ : The proof follows by Lemma 3.4.

$p, q \in N$ : Then, we have $q \overset{\ell^u}{\to} u$ and $q \overset{\ell^s}{\to} v \in G$ for some nodes $u, v \in S$. Since $G$ is well-formed, there must exist nodes $w, z \in S$ such that $p \overset{\ell^u}{\to} w$ and $p \overset{\ell^s}{\to} z \in G$. Since $G$ is closed, $u \overset{\leq}{\Rightarrow} w \in G$, and also $z \overset{\leq}{\Rightarrow} v \in G$. Then we have:

$$\widehat{h_G}(q){\downarrow}\ell^u = \widehat{h_G}(u) \leq \widehat{h_G}(w) = \widehat{h_G}(p){\downarrow}\ell^u$$

and

$$\widehat{h_G}(p){\downarrow}\ell^s = \widehat{h_G}(z) \leq \widehat{h_G}(v) = \widehat{h_G}(q){\downarrow}\ell^s$$

where the inequalities follow by Lemma 3.4 and the equalities follow by definition of the graph $G$.

$p \in N, q \in S$ : If $G^{\uparrow}(q) = \varnothing$, then $\widehat{h_G}(q) = \top$ and the proof follows immediately. Otherwise, suppose that $G^{\uparrow}(q) = \{p_1, \ldots, p_k\}$. Since $G$ is closed,we have $p \overset{\leq}{\Rightarrow} p_i \in G$ for $i \in 1..k$. Since $G$ is well-formed, the $p_i$'s are all $N$ nodes. Reasoning as in the previous case, for $i \in 1..k$, we obtain $\widehat{h_G}(p) \leq \widehat{h_G}(p_i)$. Now, from $\widehat{h_G}(q) = \sqcap \{\widehat{h_G}(p_1), \ldots, \widehat{h_G}(p_k)\}$ it follows that $\widehat{h_G}(p) \leq \widehat{h_G}(q)$ by the definition of $\sqcap$.

$p \in Q, q \in S$ : The proof is similar to the previous case. Consider again the set $G^{\uparrow}(q)$: if this set is empty, then $\widehat{h_G}(q) = \top$ and the proof follows immediately. Otherwise $G^{\uparrow}(q) = \{p_1, \ldots, p_k\}$, and from $G$ being closed we know $p \overset{\leq}{\Rightarrow} p_i \in G$ for $i \in 1..k$. Since $G$ is well-formed, $p_i = p$ for every $i \in 1..k$. The proof follows directly from this observation.

No other case applies, $G$ being well-formed by hypothesis. ■

### 3.3. Type Inference Algorithm

We are finally ready to define the inference algorithm and prove its main properties.

DEFINITION 3.10. [Inference Algorithm]

*Input:* A closed term $a$.

*1:* Construct the constraint system, and the constraint graph $G$;

*2:* Close $G$;

*3:* Check that $G$ is well-formed: if so, report **success**, otherwise **fail**.

THEOREM 3.4 (Soundness and Completeness). *Let $a$ be a closed term. Then $a$ is typable if and only if the inference algorithm reports* **success**.

*Proof.* By Theorem 3.1, $a$ is typable if and only if the constraint system generated by rewriting is solvable. By Theorem 3.2, the constraint system is solvable if and only if the corresponding constraint graph is solvable. By Theorem 3.3, the constraint graph is solvable if and only if it is well-formed. ■

PROPOSITION 3.4 (Time Complexity). *The total running time of the algorithm is $O(n^3)$ where $n$ is the size of the input term.*

*Proof.* Let $n$ be the size of the input term. The rewriting iterates $n$ times, generating a constraint system with $O(n)$ number of constraints. The corresponding constraint graph has $O(n)$ nodes. Thus step *1* takes $O(n)$. As explained in [15], closing the graph (step *2*) takes $O(n^3)$ and checking well-formedness (step *3*) takes $O(n^2)$. Therefore, the entire type inference algorithm requires $O(n^3)$ steps. ■

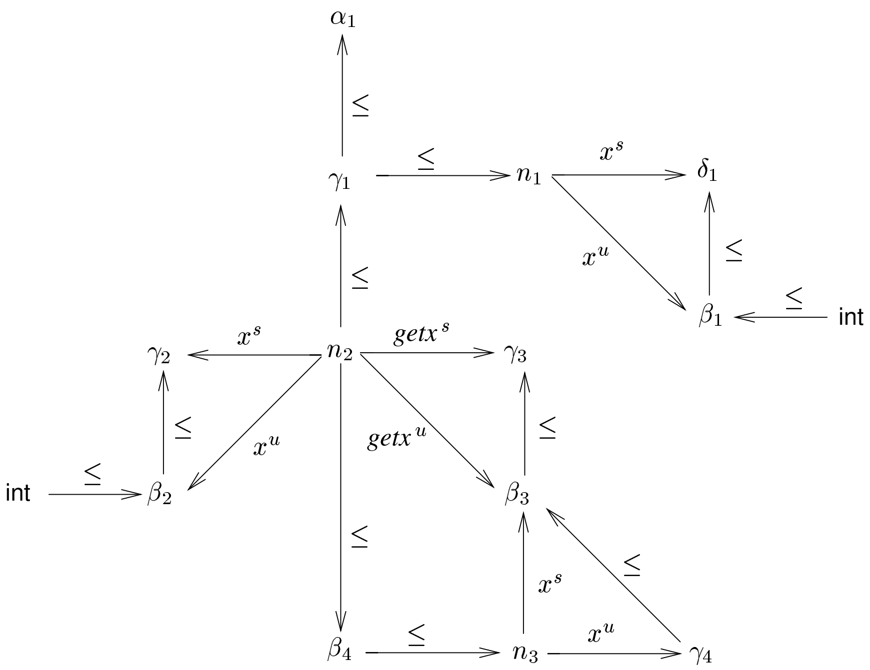
### 3.4. Examples

We conclude the description of the inference algorithm with a few simple examples.

EXAMPLE 3.1. Consider the term $[x = 0, \mathit{getx} = \varsigma(s)s.x].x := 1$. The subterms are: (i) the term itself, (ii) $[x = 0, \mathit{getx} = \varsigma(s)s.x]$, (iii) 1, (iv) 0, (v) $s.x$ and (vi) $s$. Applying the inference rules from Figure 2 we get the following constraint system:

$$
\begin{array}{rll}
\mathsf{C} \;=\; & \{\gamma_1 \leq \alpha_1, \gamma_1 \leq [x : (\beta_1, \delta_1)], \beta_1 \leq \delta_1, & \text{(i)} \\
& [x : (\beta_2, \gamma_2), \mathit{getx} : (\beta_3, \gamma_3)] \leq \gamma_1, \beta_2 \leq \gamma_2, \beta_3 \leq \gamma_3, & \text{(ii)} \\
& \mathsf{int} \leq \beta_1, & \text{(iii)} \\
& \mathsf{int} \leq \beta_2, & \text{(iv)} \\
& \beta_4 \leq [x : (\gamma_4, \beta_3)], \gamma_4 \leq \beta_3, & \text{(v)} \\
& [x : (\beta_2, \gamma_2), \mathit{getx} : (\beta_3, \gamma_3)] \leq \beta_4\} & \text{(vi)}
\end{array}
$$

Let $G$ be the corresponding constraint graph, displayed below. The nodes $n_1$, $n_2$ and $n_3$ are $N$ nodes that correspond to the types $[x : (\beta_1, \delta_1)]$, $[x : (\beta_2, \gamma_2), \mathit{getx} : (\beta_3, \gamma_3)]$ and $[x : (\gamma_4, \beta_3)]$, respectively. The $Q$ node $\mathsf{int}$ has been duplicated for displaying purposes. All the remaining nodes are $S$ nodes.




It easy to check that (the closure of) the graph $G$ is well-formed. The solution $h_G$ is constructed as follows. At the first iteration, we have $h_0(\psi) = \top$ for all $\psi \in S$. The $S$ nodes of $G$ can be partitioned into the sets $S_0 = \{\alpha_1, \beta_1, \beta_2, \beta_3, \gamma_2, \gamma_3, \gamma_4, \delta_1\}$ and $S_1 = \{\beta_4, \gamma_1\}$. For every $\psi \in S_0$ we have $G^{\uparrow}(\psi) = \varnothing$. For the nodes in $S_1$ we have: $G^{\uparrow}(\gamma_1) = \{n_1\}$ and $G^{\uparrow}(\beta_4) = \{n_3\}$. At the next iteration, we then have: $h_1(\psi) = h_0(\psi) = \top$ for every $\psi \in S_0$, $h_1(\gamma_1) = \widehat{h_0}(n_1)$ and $h_1(\beta_4) = \widehat{h_0}(n_3)$ where, by definition, $\widehat{h_0}(n_1) = \widehat{h_0}(n_3) = [x : (\top, \top)]$. A further iteration shows that $h_2 = h_1$. Thus we can choose $h_G = h_1$, and obtain:

$$
\begin{array}{rcl}
h_G(\psi) & = & \top \quad \text{for every } \psi \in S_0 \\
h_G(\gamma_1) & = & h_G(\beta_4) = [x : (\top, \top)]
\end{array}
$$

Given any solvable constraint graph $G$, the construction of the canonical substitution $h_G$ introduced in the proof of Proposition 3.3, and exemplified above, provides us with a systematic way for extracting a solution from $G$. Unfortunately, $h_G$ is not well-suited for display purposes, as it computes the least informative type for the input term. In the previous example, the type of the input term is the type that $h_G$ associates with the variable $\alpha_1$, i.e. $h_G(\alpha_1) = \top$. This is not a special case: an inspection of the rewrite rules of Figure 2 shows that the type variable associated with the input term does never receive an upper bound. It then follows by the definition of $h_G$ that the type associated with this variable is always $\top$.

It is possible, however, to derive more informative solutions to a well-formed graph: one solution is outlined next. For every node $s \in S$, let $G^{\downarrow}(s) = \{ p \in N \cup Q \mid p \overset{\leq}{\rightarrow} s \in G \}$. Then define

$$\kappa_G(s) \;=\; \begin{cases} \bot & \text{if } G^{\downarrow}(s) = \varnothing, \\ \sqcup \{ \widehat{\kappa_G}(p) \mid p \in G^{\downarrow}(s) \} & \text{otherwise.} \end{cases}$$

The substitutions $h_G$ and $\kappa_G$ are the "dual" of each other. The former associates each $S$ node in $G$ with its upper bounds while the latter associates each $S$ node in $G$ with its lower bounds; the reader can check that the proof of Theorem 3.3 goes through in essentially the same way if we replace $h_G$ with $\kappa_G$. Using the latter, for the term of Example 3.1, we obtain:

- $\kappa_G(\beta_1) = \kappa_G(\delta_1) = \kappa_G(\beta_2) = \kappa_G(\gamma_2) = \kappa_G(\beta_3) = \kappa_G(\gamma_3) = \mathsf{int}$,
- $\kappa_G(\gamma_4) = \bot$,
- $\kappa_G(\alpha_1) = \kappa_G(\gamma_1) = \kappa_G(\beta_4) = [x : (\mathsf{int}, \mathsf{int}), \mathit{getx} : (\mathsf{int}, \mathsf{int})]$.

Thus, the type of the input term would be $\widehat{\kappa_G}(\alpha_1) \;=\; [x : (\mathsf{int}, \mathsf{int}), \mathit{getx} : (\mathsf{int}, \mathsf{int})]]$, which is the type one would expect for the input term.

A minor difficulty with $\kappa_G$ is that, unlike $h_G$, it is not always finite.

EXAMPLE 3.2. Consider the term $[\ell = \varsigma(s)s]$. For this term, the algorithm generates the following constraint system:

$$\{[\ell : (\beta, \gamma)] \leq \alpha, \beta \leq \gamma, [\ell : (\beta, \gamma)] \leq \beta\}.$$

If we construct the corresponding constraint graph, and then close it, we easily see that it is well-formed. The type for the input term would then be $\kappa_G(\alpha_1) = [\ell : (\kappa_G(\beta), \kappa_G(\gamma))]$ where $\kappa_G(\beta)$ and $\kappa_G(\gamma)$ satisfy the recursive equations:

$$\begin{aligned} \kappa_G(\beta) \;&=\; \widehat{\kappa_G}([\ell : (\beta, \gamma)]) \;=\; [\ell : (\kappa_G(\beta), \kappa_G(\gamma))], \\ \kappa_G(\gamma) \;&=\; \widehat{\kappa_G}([\ell : (\beta, \gamma)]) \;=\; [\ell : (\kappa_G(\beta), \kappa_G(\gamma))]. \end{aligned}$$

Therefore, adopting $\kappa_G$ requires the ability to provide a finite representation for regular trees. Nevertheless, this additional complication appears to be worthwhile, given the more informative structure of the displayed solution. In this example, the displayed type would be the recursive type $[\ell : (\mu(\alpha)[\ell : (\alpha, \alpha)], \mu(\alpha)[\ell : (\alpha, \alpha)])]$.

EXAMPLE 3.3. As a final example, consider the term $[\,].\ell$. This terms is clearly unsound, as it attempts to select the $\ell$ label from the empty object. Running the inference algorithm, we obtain the following (closed) constraint graph:

$$\begin{tikzcd}
\beta \arrow[r, "\leq"] & n_1 \arrow[r, "\ell^s"] \arrow[dr, "\ell^u"'] & \alpha \\
n_2 \arrow[u, "\leq"] \arrow[ur, "\leq"'] & & \gamma \arrow[u, "\leq"']
\end{tikzcd}$$

The graph is not well-formed, as $n_1$ has an outgoing edge $\ell^\eta$ while $n_2$ does not. Therefore, the algorithm fails and rejects the term as unsound, as expected.

## 4. RELATIONSHIPS WITH OTHER OBJECT TYPE SYSTEMS

In [2], Abadi and Cardelli define a suite of type systems for the $\varsigma$-calculus. In this section we give a detailed comparison between our system and the first-order type systems defined in that book. We also provide a comparison with the system of Self types in [2], and with the system of *simple* Self types from [16].

### 4.1. Finite and Recursive Types

The system of Split types is easily seen to be more powerful than the system of recursive types (hence, more powerful than the system of finite types too). In fact, every derivation that uses finite or recursive types can be encoded as a derivation in our system. This follows from observing $(i)$ that recursive types à la Abadi and Cardelli can be coded as Split types in which the update and the select components of each method are identical, and $(ii)$ that invariant subtyping is a special case of our variant subtyping for Split types. Furthermore, the inclusion between the systems of recursive and Split types is strict, as there exist terms typable in the latter that are not typable in the former (cf. Example 2.1).

### 4.2. Types with Variance Annotations

As an enhancement to the system of finite and recursive types, Abadi and Cardelli propose a system where *variance* annotations are used to identify read-only and write-only methods. In this system, it is possible to (soundly) allow subtyping in depth over method types. Specifically, read-only methods can be subtyped covariantly while write-only methods can be subtyped contravariantly.

To ease the comparison with the system of Split types, we rely on a slightly different formulation of the type system with variance annotations from [2]. In our formulation, which we refer to as $\mathbf{Ob}^{\vee}$, types are represented as regular trees in ways similar to our Split types. Briefly, an $\mathbf{Ob}^{\vee}$ type is a partial function from a prefix-closed and non-empty set of paths to a signature that includes the type constructor $[\,]$ and a set of primitive type constructors $Q$. A path is a finite string drawn from the set $\{\ell\nu\}^*$ with $\nu \in \{^\circ, ^-, ^+\}$ and $\ell \in \mathcal{L}$. We employ the same "displayed" form we introduced for Split types, writing $[\ell_i\nu_i : {B_i}^{i\in I}]$ for the type $A$ such that $A(\epsilon) = [\,]$ and $A((\ell_i\nu_i)\pi) = B_i(\pi)$. A consequence of this representation is that recursive $\mathbf{Ob}^{\vee}$ types are equal, rather than isomorphic, to their unfoldings: this, in turn, implies that we can rely on the same untyped syntax of terms we used in our calculus, thus disregarding the term-level operators $\mathsf{fold}/\mathsf{unfold}$ from [2].

There is no loss of generality in these choices, it simply facilitates the definition of the encoding and the formal comparison between the two systems. The typing rules of $\mathbf{Ob}^{\vee}$ are given in Figure 4.

EXAMPLE 4.1. Going back to Example 1.3, the two terms $p_0$ and $p_2$ can now be given the following $\mathbf{Ob}^{\vee}$ types:

$$\begin{aligned} p_0 &= [\mathit{move} = \varsigma(s)s] \\ &\quad : P_0^+ = [\mathit{move}^+ : P_0^+] \\ p_2 &= [x = \varsigma(s)s.\mathit{move}.y, y = 0, \mathit{move} = \varsigma(s)s.y := s.y + 1] \\ &\quad : P_2^+ = [x^\circ : \mathsf{int}, y^\circ : \mathsf{int}, \mathit{move}^+ : P_2^+]. \end{aligned}$$

Furthermore, the subtyping rules of $\mathbf{Ob}^{\vee}$ validate the relationship $P_2^+ \leq P_0^+$. Consequently, they allow the derivation of the typing $[\ell = p_2].\ell := p_0 : [\ell : P_0^+]$, thus recovering the structural information that was lost with simple recursive types. There is a price to pay, however, as the variance annotations in the types $P_0^+$ and $P_2^+$ disallow updates on the *move* method.

Variance annotations can be modeled naturally with our Split types. The object type $[\ell_i\nu_i : B_i^{i\in I}]$ can be represented as the Split type $[\ell_i : (B_i^u, B_i^s)^{i\in I}]$, where for every $i \in I$ we have $(B_i^u, B_i^s) = (B_i, \top)$ when $\nu_i = {}^-$, $(B_i^u, B_i^s) = (\bot, B_i)$ when $\nu_i = {}^+$ and $(B_i^u, B_i^s) = (B_i, B_i)$ when $\nu_i = {}^\circ$. With this representation, the typing rules for method selection and method update validate the expected effects of the annotations. Selecting a write-only method returns a term of type $\top$, which cannot be used in any interesting context. Similarly, updating a read-only method is only allowed if the new method body has type $\bot$. As we noted, terms of type $\bot$ diverge, which again makes them of little use in any interesting context.

The encoding we just outlined is formalized in the next subsection.

#### 4.2.1. *Encoding* $\mathbf{Ob}^{\vee}$ *Typings with* $\mathbf{Ob}^{\downarrow\uparrow}$ *Typings*

We first define an encoding for types and judgements, and then show that the encoding of a derivation with variance annotations is a valid derivation in the system $\mathbf{Ob}^{\downarrow\uparrow}$.

DEFINITION 4.1. [Encodings of Types]

- $[\![\, Q \,]\!] = Q$ and $[\![\, \top \,]\!] = \top$,
- $[\![\, [\ell_i\nu_i : B_i^{i\in I}] \,]\!] = [\ell_i : [\![\, B_i \,]\!]^{\nu_i\, i\in I}]$,
  where $[\![\, B_i \,]\!]^\circ = ([\![\, B_i \,]\!], [\![\, B_i \,]\!])$, $\quad [\![\, B_i \,]\!]^+ = (\bot, [\![\, B_i \,]\!])$, $\quad [\![\, B_i \,]\!]^- = ([\![\, B_i \,]\!], \top)$.

Environments and judgements are encoded by lifting the type encoding in the obvious way. The only non-standard case is that of judgements of the form $E \vdash^{\vee} \nu B \leq \nu' B'$, whose encoding is $[\![\, E \,]\!] \vdash [\![\, B \,]\!]^{\nu} \leq [\![\, B' \,]\!]^{\nu'}$.

LEMMA 4.1 (Preservation of Subtyping). *Let $A$ and $B$ be arbitrary types in the system* $\mathbf{Ob}^{\vee}$.

*1. If $E \vdash^{\vee} A \leq A'$ then $[\![\, E \,]\!] \vdash [\![\, A \,]\!] \leq [\![\, A' \,]\!]$.*

*2. If $E \vdash^{\vee} \nu B \leq \nu' B'$ then $[\![\, E \,]\!] \vdash [\![\, B \,]\!]^{\nu} \leq [\![\, B' \,]\!]^{\nu'}$.*

*Proof.* By simultaneous induction on (1) and (2).

*(1).* Assume $E \vdash^{\vee} A \leq A'$. There are five cases to consider. Those for **(Sub Refl**$^{\vee}$**)**, **(Sub Hist**$^{\vee}$**)** and **(Sub Top**$^{\vee}$**)** are immediate. **(Sub Trans**$^{\vee}$**)** follows directly by induction hypothesis. For **(Sub Object**$^{\vee}$**)** we reason as follows. We have $A = [\ell_i\nu_i : B_i^{i\in I}]$ and

**Typing**: (Val Const$^{\vee}$), (Val Var$^{\vee}$) and (Val Subsume$^{\vee}$) as the corresponding rules in Figure1.

(Val Select$^{\vee}$) $$\frac{E \vdash^{\vee} a : A \quad \nu_j \in \{^{\circ},^{+}\} \quad j \in I}{E \vdash^{\vee} a.\ell_j : B_j} \quad (A = [\ell_i \nu_i : B_i{}^{i \in I}])$$

(Val Update$^{\vee}$) $$\frac{E \vdash^{\vee} a : A \quad E, s : A \vdash^{\vee} b : B_j \quad \nu_j \in \{^{\circ},^{-}\} \quad j \in I}{E \vdash^{\vee} a.\ell_j \Leftarrow \varsigma(s)b : A} \quad (A = [\ell_i \nu_i : B_i{}^{i \in I}])$$

(Val Object$^{\vee}$) $$\frac{E, s : A \vdash^{\vee} b_i : B_i \quad \forall i \in I}{E \vdash^{\vee} [\ell_i = \varsigma(s_i)b_i{}^{i \in I}] : A} \quad (A = [\ell_i \nu_i : B_i{}^{i \in I}])$$

**Subtyping**: (Sub Refl$^{\vee}$), (Sub Trans$^{\vee}$), (Sub Top$^{\vee}$) and (Sub Hist$^{\vee}$) as the corresponding rules in Figure 1.

(Sub Object$^{\vee}$) $$\frac{E \cup \{A \le A'\} \vdash^{\vee} \nu_j B_j \le \nu'_j B'_j \quad \forall j \in J \subseteq I}{E \vdash^{\vee} A \le A'} \quad \begin{pmatrix} A = [\ell_i \nu_i : B_i{}^{i \in I}] \\ A' = [\ell_j \nu'_j : B'_j{}^{j \in J}] \end{pmatrix}$$

(Sub Invariant) $$\frac{}{E \vdash^{\vee} {}^{\circ}B \le {}^{\circ}B}$$ (Sub Contravariant) $$\frac{E \vdash^{\vee} B' \le B \quad \nu \in \{^{\circ},^{-}\}}{E \vdash^{\vee} \nu B \le {}^{-}B'}$$

(Sub Covariant) $$\frac{E \vdash^{\vee} B \le B' \quad \nu \in \{^{\circ},^{+}\}}{E \vdash^{\vee} \nu B \le {}^{+}B'}$$

**FIG. 4.** Typing Rules of $\mathbf{Ob}^{\vee}$.

$A' = [\ell_i \nu_i{}' : B_i'^{i \in J}]$ with $J \subseteq I$, and for $i \in J$ we also have $\vdash^{\vee} \nu_i B_i \le \nu'_i B'_i$. By induction hypothesis (2), it follows that $[\![ E ]\!] \vdash [\![ B_i ]\!]_i^{\nu} \le [\![ B'_i ]\!]^{\nu'_i}$, and then the desired judgement derives by (Sub Object).

*(2).* Assume $\vdash^{\vee} \nu B \le \nu' B'$, and consider the three possible cases for the last rule in the derivation:

**(Sub Invariant)** Then $B = B'$ and $\nu = \nu' = {}^{\circ}$. By definition $[\![ B ]\!]^{\circ} = ([\![ B ]\!], [\![ B ]\!])$. Now $[\![ E ]\!] \vdash [\![ B ]\!] \le [\![ B ]\!]$ derives by (Sub Refl), and $[\![ E ]\!] \vdash [\![ B ]\!]^{\circ} \le [\![ B ]\!]^{\circ}$ by (Sub Component).

**(Sub Covariant)** Then $\nu \in \{^{\circ},^{+}\}$, $\nu' = {}^{+}$ and $E \vdash^{\vee} B \le B'$. From the last judgement, by induction hypothesis (1), we have $(i)$ $[\![ E ]\!] \vdash [\![ B ]\!] \le [\![ B' ]\!]$. By definition, $[\![ B' ]\!]^{\nu'} = (\bot, [\![ B ]\!])$, so we distinguish two subcases for $\nu$. If $\nu = {}^{\circ}$ then $[\![ B ]\!]^{\nu} = ([\![ B ]\!], [\![ B ]\!])$ and $[\![ E ]\!] \vdash [\![ B ]\!]^{\nu} \le [\![ B' ]\!]^{\nu'}$ derives by (Sub Components) from $(i)$ and from $[\![ E ]\!] \vdash \bot \le [\![ B ]\!]$ (which in turn derives by (Sub Bot)). If instead $\nu = {}^{+}$ then $[\![ B ]\!]^{\nu} = ([\![ B ]\!], [\![ B ]\!])$, and the desired judgement derives from $(i)$ and from $[\![ E ]\!] \vdash \bot \le \bot$ (which derives by (Sub Refl)).

**(Sub Contravariant)** Then $\nu \in \{^{\circ},^{-}\}$, $\nu' = {}^{-}$ and $E \vdash^{\vee} B' \le B$. From the last judgement, by induction hypothesis (1), we have $(ii)$ $[\![ E ]\!] \vdash [\![ B' ]\!] \le [\![ B ]\!]$. By definition, $[\![ B' ]\!]^{\nu'} = ([\![ B ]\!], \top)$, and we distinguish two subcases for $\nu$. If $\nu = {}^{\circ}$, then $[\![ B ]\!]^{\nu} = ([\![ B ]\!], [\![ B ]\!])$ and $[\![ E ]\!] \vdash [\![ B ]\!]^{\nu} \le [\![ B' ]\!]^{\nu'}$ derives by (Sub Components) from $(ii)$ and from $[\![ E ]\!] \vdash [\![ B ]\!] \le \top$ (which in turn derives by (Sub Top)). If instead $\nu = {}^{-}$, then $[\![ B ]\!]^{\nu} =$

$([\![ B ]\!], [\![ B ]\!])$ and the desired judgement derives from $(ii)$ and from $[\![ E ]\!] \vdash \top \leq \top$ (which derives by (Sub Refl)). ■

THEOREM 4.1 (Preservation of Typing). *If $E \vdash^{\vee} a : A$ is derivable, then so is $[\![ E \vdash^{\vee} a : A ]\!]$.*

*Proof.* By induction on the $\mathbf{Ob}^{\vee}$ derivation of $E \vdash^{\vee} a : A$. The cases (Val Select$^{\vee}$) and (Val Update$^{\vee}$) follow directly by induction hypothesis and the definition of the type encoding. The cases (Val Subsume$^{\vee}$) and (Val Object$^{\vee}$) follow by induction hypothesis, the definition of the type encoding and Lemma 4.1. ■

EXAMPLE 4.2. Given the encoding just described, it is easy to verify that the following types can be derived for the terms $p_0$ and $p_2$:

$$\begin{array}{rcl} p_0 & = & [\mathit{move} = \varsigma(s)s] \\ & & : [\![ P_0^+ ]\!] = [\mathit{move} : (\bot, [\![ P_0^+ ]\!])] \\ p_2 & = & [x = \varsigma(s)s.\mathit{move}.y, y = 0, \mathit{move} = \varsigma(s)s.y := s.y + 1] \\ & & : [\![ P_2^+ ]\!] = [x : (\mathsf{int}, \mathsf{int}), y : (\mathsf{int}, \mathsf{int}), \mathit{move} : (\bot, [\![ P_2^+ ]\!])] \end{array}$$

As their corresponding variant object types, these Split types validate the desired subtyping relationships.

#### 4.2.2. *Encoding of typed λ-terms*

By Theorem 4.1 it follows that our system $\mathbf{Ob}^{\downarrow\uparrow}$ is at least as powerful as the system $\mathbf{Ob}^{\vee}$. As we shall prove shortly, the inclusion is in fact *strict*. A further consequence of Theorem 4.1 is that the simply typed λ-calculus, with subtyping, can be encoded in $\mathbf{Ob}^{\downarrow\uparrow}$ via a (sub)type preserving transformation. This transformation is obtained directly by applying $(i)$ Abadi and Cardelli's encoding of typed λ-terms into ς-terms, and $(ii)$ the encoding of $\mathbf{Ob}^{\vee}$ types we just illustrated. As a result of the composite translation, a function $\lambda(x : A)b\{x\}$ with argument type $A$ and result type $B$ is encoded by the following term:

$$\begin{array}{l} [\![ \lambda(x : A)b\{x\} : A \to B ]\!] = \\ \qquad [arg = \varsigma(s : [arg : ([\![ A ]\!], [\![ A ]\!]), val : ([\![ B ]\!], [\![ B ]\!])])\, s.arg, \\ \qquad\; val = \varsigma(s : [arg : ([\![ A ]\!], [\![ A ]\!]), val : ([\![ B ]\!], [\![ B ]\!])])\, [\![ b\{x\} ]\!]\{\{x := s.arg\}\}] \end{array}$$

Now, defining $[\![ A \to B ]\!] = [arg{:}([\![ A ]\!], \top),\ val{:}(\bot, [\![ B ]\!])]$, we obtain a type constructor for functions that is contravariant in its input type and covariant in its output type. Finally, by the typing rules of $\mathbf{Ob}^{\downarrow\uparrow}$, we obtain:

$$\begin{array}{rcl} [\![ \lambda(x : A)b\{x\} : A \to B ]\!] & : & [arg : ([\![ A ]\!], [\![ A ]\!]), val : ([\![ B ]\!], [\![ B ]\!])] \\ & \leq & [arg : ([\![ A ]\!], \top), val : (\bot, [\![ B ]\!])] \\ & = & [\![ A \to B ]\!] \end{array}$$

#### 4.2.3. $\mathbf{Ob}^{\downarrow\uparrow}$ *is more powerful that* $\mathbf{Ob}^{\vee}$

There is a simple and intuitive reason why Split types are more expressive than $\mathbf{Ob}^{\vee}$ types: there are "more" Split types than there are $\mathbf{Ob}^{\vee}$ types for the same (untyped) term. This is easily understood when we look at the encoding we just defined, and observe that only very specific Split types are used to encode $\mathbf{Ob}^{\vee}$ types. As a consequence, there exist terms that are typable in $\mathbf{Ob}^{\downarrow\uparrow}$ that are not typable in $\mathbf{Ob}^{\vee}$.

EXAMPLE 4.3. To make the example more readable, we work with an enriched calculus that includes $\lambda$-abstractions, monomorphic lets, primitive operators and subtyping over primitive types. As we discussed above, $\lambda$-abstractions can be encoded in the core calculus. The remaining extensions do not cause any loss of generality as a similar, but more contrived, example can be given relying only on object terms and types.

Let $\mathtt{div} : \mathsf{int} \times \mathsf{int} \to \mathsf{int}$ and $/ : \mathsf{real} \times \mathsf{real} \to \mathsf{real}$ denote the operators of integer division and real division, respectively, and assume that $\mathsf{int} \leq \mathsf{real}$. Consider the terms:

$$\begin{aligned} p_1 &\triangleq [a = 1, \ell = \varsigma(s)\, s.a \;\mathtt{div}\; 2] \\ p_2 &\triangleq [a = 1.0] \end{aligned}$$

It is easy to verify that the following judgements are derivable in $\mathbf{Ob}^{\downarrow\uparrow}$.

$$\begin{aligned} &\vdash\; p_1 : [a : (\mathsf{int}, \mathsf{int}), \ell : (\mathsf{int}, \mathsf{int})] \\ &\vdash\; p_2 : [a : (\mathsf{real}, \mathsf{real})] \end{aligned}$$

Now, since the judgements $\vdash [a : (\mathsf{int}, \mathsf{int}), \ell : (\mathsf{real}, \mathsf{real})] \leq [a : (\mathsf{int}, \mathsf{int})] \leq [a : (\mathsf{int}, \mathsf{real})]$ and $\vdash [a : (\mathsf{real}, \mathsf{real})] \leq [a : (\mathsf{int}, \mathsf{real})]$ are all derivable, by subsumption we have,

$$\begin{aligned} &\vdash\; p_1 : [a : (\mathsf{int}, \mathsf{real})] \\ &\vdash\; p_2 : [a : (\mathsf{int}, \mathsf{real})] \end{aligned}$$

Next, consider the following expression:

$$\mathtt{let}\; f = \lambda(x)(x.a := 2).a \;\mathtt{in}\; f(p_1)/f(p_2)$$

The judgement $x : [a : (\mathsf{int}, \mathsf{real})] \vdash (x.a := 2).a : \mathsf{real}$ is derivable in $\mathbf{Ob}^{\downarrow\uparrow}$. The constant 2 has type $\mathsf{int}$, thus it may legally be used to update $x$'s field $a$, whose update type is also $\mathsf{int}$. Selecting $a$ from $x$ returns the type $\mathsf{real}$ as advertised by the select type of $a$ in $x$. From the last judgement, we derive

$$\vdash\; f : [a : (\mathsf{int}, \mathsf{real})] \to \mathsf{real}$$

Therefore, both the applications of $f(p_1)$ and $f(p_2)$ in the the body of the `let` expression type check, hence, so does the expression.

Next, we show that the `let` expression does not type check in the system $\mathbf{Ob}^{\vee}$. As mentioned, the essence of the problem is that $\mathbf{Ob}^{\vee}$ has "fewer" types than $\mathbf{Ob}^{\downarrow\uparrow}$. In particular, there exists no $\mathbf{Ob}^{\vee}$ type corresponding to the Split type $[a : (\mathsf{int}, \mathsf{real})]$, the typing failure is a direct consequence of this fact.

Consider again the two terms $p_1$ and $p_2$. In $\mathbf{Ob}^{\vee}$ we derive:

$$\begin{aligned} &\vdash^{\vee}\; p_1 : [a^{\circ} : \mathsf{int}, \ell^{\circ} : \mathsf{real}] \\ &\vdash^{\vee}\; p_2 : [a^{\circ} : \mathsf{real}] \end{aligned}$$

It should be noted, in particular, that typing $p_1$ requires $a$ to have type $\mathsf{int}$ (as opposed to $\mathsf{real}$) as `div` requires its arguments to have type $\mathsf{int}$. Also, it is not difficult to see that these two $\mathbf{Ob}^{\vee}$ types are minimum for $p_1$ and $p_2$.

Now, to type check the applications $f(p_1)$ and $f(p_2)$, we can try and maximize the input type of $f$, so that the types of $p_1$ and $p_2$ can be subsumed to that type. Unfortunately, the two maximal types for the input parameter of $f$ are $[a^\circ : \mathsf{real}]$ and $[a^\circ : \mathsf{int}]$. To see that, we may reason as follows. The type of $x$ in the body of the lambda abstraction must clearly be an object type, which must contain the field $a$. The field $a$, in turn, must be invariant as it is both updated and selected. Consequently, $x$ may be assigned the two $\mathbf{Ob}^{\vee}$ types $[a^\circ : \mathsf{real}]$ and $[a^\circ : \mathsf{int}]$, or any subtype thereof, but no proper supertype. Since the two types in question are incomparable in $\mathbf{Ob}^{\vee}$, they are maximal.

To conclude, note that neither $[a^\circ : \mathsf{real}]$ nor $[a^\circ : \mathsf{int}]$ is a supertype of *both* the type of $p_1$ and the type of $p_2$. As a consequence, only one of the two applications $f(p_1)$ and $f(p_2)$ type checks (but not both) and therefore the `let` expression itself fails to type check.

### 4.3. Self Types

The system of Self types[3] from [2] is built around two main ideas. First, Self types are defined as a combination of recursive types and existential types in such a way that the desirable subtyping relationships hold. Second, a special typing rule is included for method updates in order to preserve soundness. We illustrate these ideas with an example. In the system of Self types, a 2D object can be assigned the following type (using the syntax of Self types this type would be written as $\varsigma(X)[x : \mathsf{int}, y : \mathsf{int}, \mathit{move} : X]$):

$$\mu(X)\exists(Y \leq X)[x : \mathsf{int}, y : \mathsf{int}, \mathit{move} : Y]$$

There are two important aspects to this type. First, it validates the subtyping $\mu(X)\exists(Y \leq X)[x : \mathsf{int}, y : \mathsf{int}, \mathit{move} : Y] \leq \mu(X)\exists(Y \leq X)[x : \mathsf{int}, \mathit{move} : Y]$ because subtyping over bounded existentials is covariant on the bounds. Second, it hides the "actual" type of self: the existential quantifier is introduced at the time of object formation – when the real type of self is known – and then abstracted away from the type. Abstracting over the type of self restricts the way by which methods returning self can be updated. The typing rule for method update is given below:

$$\frac{\begin{array}{c}(A \equiv \varsigma(X)[\ldots, \ell : B\{X\}, \ldots]) \\ E \vdash a : A \quad E, Y \leq A, s : Y \vdash b : B\{\!\{Y\}\!\}\end{array}}{E \vdash a.\ell \Leftarrow \varsigma(s)b : A}$$

The intuitive reading of this rule is as follows. The current type $A$ of the term $a$ may be the result of several subsumption steps; so it only conveys partial knowledge about the structure of $a$. Consequently, when updating the method $\ell$ of $a$, we can only assume that the actual type of $a$ (hence of the self variable $s$) is *some* type $Y \leq A$. Furthermore, if the original type of $\ell$ depended on the type of self, we must now prove that the type of the new body depends on the type variable $Y$. In other words, methods returning self can only be updated with methods that either return self or an updated self. Thus, for example, if we let $o = [\mathit{move} = \varsigma(s)s]$, then the term $o.\mathit{move} := o$ is not typable with Self types since $o$ is not self or an updated self (i.e., it is equal to self but not self itself !), while the term $o.\mathit{move} \Leftarrow \varsigma(s)s$ is perfectly typable.

[3] We are referring the the system of *Primitive Covariant Self types* in Chap. 16 of [2].

This last example shows that our system is not less powerful than the system of Self types, as both updates are typable with Split types. Unfortunately, however, there also exist terms that are typable with Self types but not typable in our system. The reason for that is that Split types fail to validate all the subtyping relationships that are available for Self types. The problem can be explained informally as follows. At first, we may think that the two component types available with Split types can be used to trace the internal and external types of an object. More precisely, that the update component can be used as a placeholder for the internal type (the self type) while the select component can be used as the external type. In this view, subtyping à la Self types would be possible by always keeping the update component unchanged in order to remember the original type of self.

Unfortunately, it is not difficult to see that this idea does not work properly, as it fails to capture the abstraction provided by the existential quantifier in the definition of Self types: indeed, it results into "too concrete" a representation that causes a loss of expressive power and of provable subtyping relationships. Consequently, there exist terms typable with Self types that are not typable with Split types.

EXAMPLE 4.4. Consider the terms from Example 2.1.

$$\begin{aligned}
p_2 &= [x = \varsigma(s)s.\mathit{move}.y, y = 0, \mathit{move} = \varsigma(s)s.y := s.y + 1] \\
&\quad : P_2 = [x : (\mathsf{int}, \mathsf{int}), y : (\mathsf{int}, \mathsf{int}), \mathit{move} = (P_2, P_2)] \\
p_0 &= [\mathit{move} = \varsigma(s)s] \\
&\quad : P_0 = [\mathit{move} : (P_0, P_0)] \\
p &= [\ell = p_2].\ell := p_0 \\
&\quad : [\ell : (P, P)] \quad \mathit{where}\ P = [\mathit{move} : (P_2 \sqcap P_0, P)]
\end{aligned}$$

Now consider the update $(p.\ell).\mathit{move} \Leftarrow \varsigma(s)s$. With Self types, this term can be given the type $\varsigma(X)[\mathit{move} : X]$. With Split types, instead the term is not typable. Rather than giving a formal proof, we can argue informally as follows. From $p : [\ell : (P, P)]$, in $\mathbf{Ob}^{\downarrow\uparrow}$ we derive

$$p.\ell \ : \ [\mathit{move} : (P_2 \sqcap P_0, P)]$$

To type the update $(p.\ell).\mathit{move} \Leftarrow \varsigma(s)s$, by the rule (Val Update), we must derive

$$s : [\mathit{move} : (P_2 \sqcap P_0, P)] \vdash s : D$$

for a type $D$ such that $[\mathit{move} : (P_2 \sqcap P_0, P)] \leq [\mathit{move} : (D, \top)]$, i.e. such that $D \leq P_2 \sqcap P_0$. It is not difficult to see that no such type exists. We argue by contradiction, assuming the existence of a type $D \leq P_2 \sqcap P_0$ such that $s : [\mathit{move} : (P_2 \sqcap P_0, P)] \vdash s : D$ is derivable. Clearly, this judgement is derivable only if $[\mathit{move} : (P_2 \sqcap P_0, P)] \leq D$. By transitivity we have $[\mathit{move} : (P_2 \sqcap P_0, P)] \leq P_0 \sqcap P_2$. This relation is clearly false, as the type on the right has more methods than the one on the left. Hence, no such $D$ exists.

### 4.4. Simple Self Types

In [16], Palsberg and Jim extend the system of recursive types from [2] with (in their words) a "tiny drop of Self Types". In this extension, types include the special type constant `selftype`, and subtyping is covariant over those methods that can be assigned the constant `selftype`. Among others, an important difference between this system and the system of

Self Types defined in [2] is that the former does not allow methods of type `selftype` to be updated, a restriction that is required to preserve type soundness.

Types in [16] (or *simple* Selftypes) range over the set defined by the grammar $A, B ::= \texttt{selftype} \mid X \mid \mu(X)[\ell_i : B_i{}^{i\in I}]$. Types of the form $\mu(X)B$ are identified with their infinite unfoldings under the rule $\mu(X)B \to B\{\{X := \mu(X)B\}\}$. Subtyping is defined by stipulating that $X \leq_\varsigma X$ for every type variable $X$, $\texttt{selftype} \leq_\varsigma \texttt{selftype}$ and if $A$ are $B$ are of the form $[\ell_i : B_i{}^{i\in I}]$ then $A \leq_\varsigma B$ if and only if:

$$\forall \ell \in \mathsf{Dom}(B) \Rightarrow (\ell \in \mathsf{Dom}(A) \wedge A \downarrow \ell = B \downarrow \ell)$$

The reader is referred to [16] for a complete description of all the typing rules available in the system. For conciseness, we refer to this system as $\mathbf{Ob}^\varsigma$.

It follows immediately from the above definition of subtyping, that simple Self types can be encoded within the system of recursive types with variance annotations, hence with our Split types. Specifically, methods that are assigned the type simple `selftype`, can be encoded as the method type $(\perp, A)$ where $A$ is the type of self. This, in fact, is equivalent to labeling the method type with the variance annotation $^+$ so that it can be subtyped covariantly but not updated.

DEFINITION 4.2. [Simple Self Types in $\mathbf{Ob}^{\downarrow\uparrow}$] The encoding is defined by induction on the structure of Self types.[4] Let $\nu \in \{^+, ^-\}$ in:

1. $[\![ X ]\!]_Y^\nu = X$,
2. $[\![ \texttt{selftype} ]\!]_Y^+ = Y$,
3. $[\![ \texttt{selftype} ]\!]_Y^- = \perp$,
4. $[\![ \mu(X)[\ell_i : B_i{}^{i\in I}] ]\!]_Y^\nu = \mu(X)[\ell_i : ([\![ B_i ]\!]_X^-, [\![ B_i ]\!]_X^+)\ {}^{i\in I}]$.

For every Self type $A$ define the Split type $[\![ A ]\!]$ to be $[\![ A ]\!]_Z^+$ for some fresh type variable $Z$.[5] This definition is trivially lifted to type environments.

LEMMA 4.2. *Assume $A \leq_\varsigma B$ with $A$ and $B$ simple Self types. Then $[\![ A ]\!] \leq [\![ B ]\!]$.*

*Proof.* If $A$ and $B$ are type variables or $A$ and $B$ are both `selftype` then the result is immediate. Suppose $A = \mu(X)[\ell_i : B_i{}^{i\in I}]$ and $B = \mu(Y)[\ell_j : C_j{}^{j\in J}]$. Let $K \subseteq J$ be such that for every $k \in K$ we have $C_k = \texttt{selftype}$. Then,

$$\begin{array}{rcll}
[\![ A ]\!] & = & \mu(X)[\ell_i : ([\![ B_i ]\!]_X^-, [\![ B_i ]\!]_X^+)\ {}^{i\in I-K}, \ell_k : (\perp, X)\ {}^{k\in K}] & (B_i \neq \texttt{selftype}) \\
& = & \mu(X)[\ell_i : ([\![ B_i ]\!], [\![ B_i ]\!])\ {}^{i\in I-K}, \ell_k : (\perp, X)\ {}^{k\in K}] & \\
& = & [\ell_i : ([\![ B_i ]\!], [\![ B_i ]\!])\ {}^{i\in I-K}, \ell_k : (\perp, [\![ A ]\!])\ {}^{k\in K}] &
\end{array}$$

$$\begin{array}{rcll}
[\![ B ]\!] & = & \mu(Y)[\ell_j : ([\![ C_j ]\!]_Y^-, [\![ C_j ]\!]_Y^+)\ {}^{j\in J-K}, \ell_k : (\perp, Y)\ {}^{k\in K}] & (C_j \neq \texttt{selftype}) \\
& = & \mu(Y)[\ell_j : ([\![ C_j ]\!], [\![ C_j ]\!])\ {}^{j\in J-K}, \ell_k : (\perp, Y)\ {}^{k\in K}] & \\
& = & [\ell_j : ([\![ C_j ]\!], [\![ C_j ]\!])\ {}^{j\in J-K}, \ell_k : (\perp, [\![ B ]\!])\ {}^{k\in K}] &
\end{array}$$

That $[\![ A ]\!] \leq [\![ B ]\!]$ follows directly from the hypothesis that $A \leq_\varsigma B$ and the definitions of the relations $\leq_\varsigma$ and $\leq$. ■

[4] To ease the comparison, we use a finite representation for Split types using type variables and $\mu$-binders.
[5] We assume, with no loss of generality, that all $\mu$-bound variables are unique.

A consequence of the last lemma is that the encoding of simple Self types in $\mathbf{Ob}^{\mathrm{V}}$ preserves typability, i.e. $E \vdash a : A$ is derivable in $\mathbf{Ob}^{\varsigma}$ then $[\![ E ]\!] \vdash a : [\![ A ]\!]$ is derivable in $\mathbf{Ob}^{\downarrow\uparrow}$. Therefore, our type inference algorithm is complete for the system $\mathbf{Ob}^{\varsigma}$.

## 5. CONCLUSIONS

We have presented a new type system for objects together with an efficient inference algorithm. Given an input term, the algorithm derives a set of subtyping constraints, and then checks that the set is solvable. We have proved that the new type system is more powerful than all the existing first-order systems for objects, including systems with variance annotations and simple Self types. We have also described effective ways for extracting solutions from constraint sets whenever these are solvable. Of course, for modular type inference one needs the constraint set, or equivalently, the corresponding constraint graph. In any case, the size of these data structures is linear with respect to the input term, so modular type inference is still feasible and efficient.

The type inference problem we have addressed is related to that considered by other authors in the literature whose work has not yet been mentioned. In [10], Henglein studies type inference for the object calculi of Abadi and Cardelli and presents an algorithm that improves the $O(n^3)$ bound established by Palsberg in [15]. In particular, he shows that the inference problem for the system of recursive object types with subtyping can be solved in $O(n^2)$. Unfortunately, Henglein's method cannot be applied to our type system, as the key ingredient for "breaking through the $n^3$ barrier" in his algorithm is the invariant rule for object subtyping. In a series of papers [7, 6, 23], Eifrig, Smith and Trifonov study the inference problem for a polymorphic type system that includes both functions and objects, and develop powerful simplification methods for the constraint sets generated during the inference. Some of these methods have independently been studied (and improved) by Pottier [18], and are part of our current implementation of the inference algorithm.

An interesting question is whether the technique we have described can be used to infer types with variance information from non-annotated terms. Our conjecture is that this is not the case, for the reasons that follow. Palsberg and Jim show that type inference in their system is NP-complete. Intuitively, the problem for their system is in NP because for every method that returns self we have to choose between the type `selftype` or the type of the object (i.e., a recursive type). If we choose the former, then subtyping in depth is permitted but the method can no longer be updated. Conversely, if we choose the latter, then subtyping in depth is not permitted but the method can be updated. However, if we can guess which methods require the type `selftype` then we can check the typability of the term in polynomial time.

Type inference with variance annotations poses similar problems, albeit in a different context. One might initially assume that all method labels can be annotated as invariant. When needed, invariant annotations can be promoted to variant ones as dictated by the typing rules. The problem arises from the absence of least upper bounds: for instance, the two types types $[\ell^{\circ} : [\,]]$ and $[\ell^{\circ} : [\ell^{\circ} : [\,]]]$ have two incomparable upper bounds: $[\ell^{+} : [\,]]$, and $[\ell^{-} : [\ell^{\circ} : [\,]]]$. As for the system of simple Self types, it should be possible to show that the inference problem is in NP since, if we can guess which variance annotation are needed, then we can check the typability of the term in polynomial time using our techniques.

In [16], the problem is proved NP-hard via a reduction from SAT. The fact that simple Self types can be encoded with variance annotations seems to suggest that a similar reduction should be possible. Future plans may include work in that direction.

## ACKNOWLEDGMENT

We thank Craig Chambers, Patric Cousot and Alan Mycroft for discussions and feedback at the "Workshop on Types and Abstract Interpretation" held in Padova, Italy, May 1999. Special thanks to Jens Palsberg for his insightful comments and suggestions. Also, to the members of the Church Group at Boston University, and to Assaf J. Kfoury for feedback reading earlier drafts. Comments of the anonymous referees were very helpful to improve the presentation. The first author was partially supported by the Italian M.U.R.S.T. Project *Constructive Methods in Topology, Algebra and Program Analysis*, 1999–2000.

## REFERENCES

1. R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
2. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
3. K. B. Bruce, J. Crabtree, T. P. Murtagh, R. van Gent, A. Dimock, and R. Muller. Safe and decidable type checking in an object-oriented language. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pp. 29–46, oct 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.
4. F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Inf. & Comput.*, 92:48–80, 1991.
5. D. Duggan and A. Compagnoni. Subtyping for object type constructors. *Proc. 6th Int. Workshop on Foundations of Object-Oriented Languages (FOOL6)*, January 1999. Electronic proceedings: `ftp://ftp.cs.williams.edu/pub/kim/FOOL6/duggan.ps`.
6. J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, pp. 169–184, 1995.
7. J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. 1995 Mathematical Foundations of Programming Semantics Conf.* Elsevier, 1995.
8. C. Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.
9. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
10. F. Henglein. Breaking through the $n^3$ barrier: Faster object type inference. In *Proc. 4th Int. Workshop on Foundations of Object-Oriented Languages (FOOL4)*, January 1997. Electronic proceedings: `http://www.cis.upenn.edu/ bcpierce/fool/henglein.ps.gz`.
11. J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions American Math. Society*, 146:29–60, 1969.
12. D. McAllester. Inferring recursive data types. Unpublished, 1996.
13. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
14. R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1990.
15. J. Palsberg. Efficient inference of object types. *Inf. & Comput.*, 123:198–209, 1995.
16. J. Palsberg and T. Jim. Type inference with simple selftypes is np-complete. *Nordic Journal of Computing*, 4(2): 259-286, 1997.
17. S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: A technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conf.*, 1993.
18. F. Pottier. *Type Inference in the Presence of Subtyping: from Theory to Practice*. PhD thesis, Universié Paris VII, 1998.
19. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extended abstract in *Proc. LICS 93*, IEEE Computer Society Press.
20. J. C. Reynolds. Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 28, 1996.
21. D. Rémy and J. Vouillon. Objective ml: An effective object-oriented extension to ml. *Theory and Practice of Object Systems*, 1998.
22. S. Smith and T. Wang. Polyvariant Flow Analysis with Constrained Types. In G. Smolka, ed., *ESOP2000: 9th European Symposium on Programming, ESOP 2000. Joint Conf. ETAPS 2000*, vol. 1782 of *LNCS*, pp. 382–396. Springer-Verlag. 2000.

23. V. Trifonov and S. Smith. Subtyping constrained types. In R. Cousot and D. A. Gaudel, eds., *SAS'96: Static Analysis, Third International Symposium*, vol. 1145 of *LNCS*, pp. 349–365. Springer-Verlag, 1996.

24. J. Tiuryn and M. Wand. Type reconstruction with recursive types and atomic subtyping. In M.-C. Gaudel and J.-P. Jouannaud, eds., *TAPSOFT'93: Theory and Practice of Software Development, Proc. 4th Intern. Joint Conf. CAAP/FASE*, vol. 668 of *LNCS*, pp. 686–701. Springer-Verlag, 1993.