Depth Subtyping and Type Inference for Object Calculi

Michele Bugliesi
Dipartimento di Informatica
Università Ca' Foscari di Venezia
michele@dsi.unive.it
http://www.dsi.unive.it/~michele

Santiago M. Pericás-Geertsen
Department of Computer Science
Boston University
santiago@cs.bu.edu
http://www.cs.bu.edu/~santiago

Abstract

We present a new type system based on the notion of Split types. In this system, every method is assigned two types, namely, an update type and a select type. We show that the system of Split types is strictly more powerful than the system of recursive types and that type inference remains decidable and feasible. We include a number of interesting examples not typable with recursive types that are typable with Split types. In particular, we demonstrate that the subtyping relation needed to encode the λ-calculus into the Abadi and Cardelli's ς-calculus holds. We also present a polynomial-time algorithm that infers Split types and show that it is sound and complete with respect to the type system. We conclude our presentation by relating the typing power of Split types to the typing power of other systems, such as, the system of recursive types with variance annotations and the system of Self types.

1 Introduction

1.1 Background and Motivation

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 [Hin69] and independently by Milner [Mil78], 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 [PJHH+93], Java [GJS96], and ML [MTHM90] 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 ς-*calculus* [AC96].

Type inference for the ς-calculus has already been studied in the past. In [Pal95], 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 [PJ97], Palsberg and Jim extend the type system proposed in [Pal95] with the inclusion of a *restricted* form of *Self types* [AC96]. 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.

$$\text{(Sub } \mu\text{)} \qquad \frac{E, X \leq Y \vdash A \leq B}{E \vdash \mu(X)A \leq \mu(Y)B}$$

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

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 relation 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 μ) 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 error.[1]

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 $p'_2 = [x = \varsigma(s)s.move.y, y = 0, move = \varsigma(s)s.y := s.y + 1]$. It is easy to check that p'_2 has type P_2. Thus, if p_1 is an arbitrary term of (proper) type P_1 then the following term is typable and generates a run-time error,

$$(p'_2.move := p_1).x \qquad \text{(oops !!)}$$

because 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 $p_1 = [x = 0, move = \varsigma(s)s.x := s.x + 1]$ and $p_2 = [x = 0, y = 0, move = \varsigma(s)s.y := s.y + 1]$. If $P_1 \equiv \mu(X)[x : \mathsf{int}, move : X]$ and $P_2 \equiv \mu(X)[x : \mathsf{int}, y : \mathsf{int}, move : X]$ then p_1 can be assigned the type P_1 and p_2 can assigned the type P_2. Now, consider the term $p_2.move := p_1$. This term is typable with recursive types, as p_2 can be assigned the type,

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

This follows by observing that $P \leq [x : \mathsf{int}, move : P_1] = P_1$, where the last equality holds by unfolding P_1. Consequently, the term $p_2.move := p_1$ 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 many cases, it is possible to find a derivation like that for example 1.2. In other examples, however, a more powerful system is needed.

EXAMPLE 1.3. Let $p_0 = [move = \varsigma(s)s]$ and $p'_2 = [x = \varsigma(s)s.move.y, y = 0, move = \varsigma(s)s.y := s.y + 1]$ and let o 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,

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

to p'_2 so that it can be subsumed to $\mu(X)[move : X]$. As a result, the most informative type for o that can be inferred using recursive types is $[\ell : [\,]]$. An immediate consequence of this observation is that the term $o.\ell.move$ is not typable with recursive types. □

To overcome these difficulties, Abadi and Cardelli propose two solutions. 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 $P_2^+ \equiv \mu(X)[x : \mathsf{int}, y : \mathsf{int}, move^+ : X]$ where the superscript $^+$ on *move* signals that this method is read-only. With this restriction, 2D points can subsume 1D points, as $P_2^+ \leq P_1^+ \equiv \mu(X)[y : \mathsf{int}, move^+ : X]$ 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 restricted system of [PJ97].

The system of Split types presented in this paper offers a new and alternative solution for combining subtyping, recursive types and method updates in a sound and flexible way. The new system is *strictly* more powerful than the system of recursive types and allows a fairly elegant encoding of variance annotations. On the other hand, as we discuss in Section 6, Split types do not validate some of the subtypings available with Self types (the two systems are shown incomparable). However, type inference for Split types remains decidable and feasible.[2]

1.2 Split Types

Split types are object types of the form $\mu(X)[\ell_i : (B_i^u, B_i^s)^{i \in I}]$ where $B_i^u \leq B_i^s$ for every $i \in I$.[3] These types are a variant of the recursive object types presented in [AC96], obtained by splitting the type of each method ℓ_i into two components. Intuitively, the component B_i^u – or *update* component – is used to type an update for ℓ_i, whereas the component B_i^s – or *select* component – is used to type a selection for ℓ_i. The operational behavior of objects is not affected by this presentation of object types. In particular, objects are still formed as a collection of methods of the form $[\ell_i = \varsigma(s)\, b_i{}^{i \in I}]$.

The presence of two type components for each label allows a more flexible subtyping relation for object types than the invariant subtyping presented in [AC96]. The sub-

[1] We refer to as a "run-time error", an error that is traditionally prevented using a type system. For example, attempting to select or update a non-existing method from an object.

[2] By the word "feasible" we mean, that can be carried out in polynomial time.

[3] This restriction is required for the system to be sound. The reader is referred to appendix A for a proof of the subject reduction theorem.

typing relation is defined by the following rule:

(Sub Object)

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

This definition of subtyping guarantees that every pair of object types A and A' not only has a least upper bound $A \sqcup A'$, but also a greatest lower bound $A \sqcap A'$. This, in turn, has interesting and appealing consequences.

EXAMPLE 1.3. (Revisited) Let p_0 and p_2' be object terms as defined in the first part of this example. Define the Split types $S_0 \equiv \mu(X)[\mathit{move} : (X, X)]$ and $S_2 \equiv \mu(X)[x : (\mathsf{int}, \mathsf{int}), y : (\mathsf{int}, \mathsf{int}), \mathit{move} : (X, X)]$. The terms p_0 and p_2' can be assigned the types S_0 and S_2, respectively. The least upper bound between S_0 and S_2 is:

$$S_0 \sqcup S_2 \equiv \mu(X)[\mathit{move} : (S_0 \sqcap S_2, X)]$$

where $S_0 \sqcap S_2 \equiv \mu(X)[x : (\mathsf{int}, \mathsf{int}), y : (\mathsf{int}, \mathsf{int}), \mathit{move} : (S_0 \sqcup S_2), X)]$. By definition, $S_0 \leq S_0 \sqcup S_2$ and $S_2 \leq S_0 \sqcup S_2$. Therefore, the term $[\ell = p_2'].\ell := p_0$ can be assigned the type $[\ell : \mu(X)[\mathit{move} : (S_0 \sqcap S_2, X)]]$, which implies that the term $o.\ell.\mathit{move}$ is now typable. □

This last example shows that with Split types, it is possible to find more informative supertypes that are needed to type terms encountered in practice. Terms as those in Example 1.2 are also typable with Split types with similar (i.e., equally informative) types. Examples involving encodings of λ-terms into ς-terms are also typable with Split types without breaking the expected subtyping relations.

$$\text{(Sub} \rightarrow\text{)} \quad \frac{E \vdash A' \leq A \quad E \vdash B \leq B'}{E \vdash A \rightarrow B \leq A' \rightarrow B'}$$

Because subtyping between Split types is not *invariant*, but contravariant in the update components and covariant in the select components, it is possible the preserve the original subtyping relations after the translation.

1.3 Contributions of this Paper

The main contributions of this paper are:

- The introduction of a powerful type system for object types. Among systems for which type inference is decidable and feasible, ours is (to the best of our knowledge) the most powerful, i.e. it types strictly more terms than any other system in the literature.[4]
  - A precise comparison of the typing power of three systems: the Split types system, the Self types system and the recursive types system with and without variance annotations.
- A type system that supports the encoding of functions (λ-abstractions) in terms of objects in a way that the expected subtyping relations are preserved. As a result, the encoding proposed in [AC96] can be used without the need to support functions as primitives.

[4]Even though it is incomparable with the Self types system, it is yet unknown whether type inference for this system is decidable or not.

1.4 Future Work

- A precise determination of the complexity bound for the type inference algorithm presented. This algorithm is based on standard techniques for manipulating constraint sets such as *closure*, *consistency checks* and *simplification*. Polynomial-time algorithms are known for all these operations.
- The definition of a systematic method for extracting a type from the set of constraints computed by the type inference algorithm. Although the inference algorithm is equipped with sophisticated simplification methods that help reduce the size of the constraint set, the ability to read a type from the constraint set would be desirable both to display the output and to reduce the space complexity of modular type inference.

2 The Split Types System $\mathrm{Ob}^{\downarrow\uparrow}$

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

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

As in [AC96], 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 for c every free occurrence of s in b. The set of free variables of a term a is denoted by $\mathsf{FV}(a)$.

To avoid cluttering the notation in the type system, we will allow our types to be infinite (i.e., denote regular trees) without using an explicit finite representation. This was, of course, not possible while presenting the examples in the introduction; but from now on, the variables $A, B, C, D...$ will denote a (possible infinite) regular tree as opposed to some finite representation of it. We often use superscripted variables such as A^s or B^u to denote select and update types, respectively. In the absence of these superscripts, the convention is that the first component of a method type always refers to the *update* part while the second component always refers to the *select* part. The complete Split Types system is presented in figure 1.

A *type environment* is a finite mapping from the set of term variables Var to the set of types. Let $E, E', ...$ range

over the set of type environments and define $\mathsf{Dom}(E) = \{s \,|\, \exists A.(s : A) \in E\}$ and $\mathsf{Ran}(E) = \{A \,|\, \exists s.(s : A) \in E\}$. If E is a type environment then $E - \{s'\} = \{(s : A) \,|\, (s : A) \in E \text{ and } s \neq s'\}$.

A *judgement* is a relation between type environments, terms and types written as $E \vdash M : A$ or as $\vdash A \leq B$ (in which case only types are related). Let $\Im, \Im', ...$ range over a set of judgements. As for terms, we write $\Im\{s\}$ and $\Im\{\{c\}\}$ to denote a judgement where s may occur free and a judgement where every occurrence of s is replaced by a term c, respectively. For conciseness, we often write $\vdash A_1 \leq A_2 \leq A_3 \leq ... \leq A_{n-1} \leq A_n$ whenever $\vdash A_1 \leq A_2$ and $\vdash A_2 \leq A_3$ ... and $\vdash A_{n-1} \leq A_n$ are derivable.

The lemma that follows says that every method does not "advertise" (select component) more structure than what it actually "has" (update component). This is an immediate consequence of how (Val Object) and (Sub Object) are defined.

Lemma 2.1 (Typings). *For every term a, if $\varnothing \vdash a : [\ell_i : (B_i^u, B_i^s)^{i \in I}]$ then it follows, for every $i \in I$, that $\vdash B_i^u \leq B_i^s$.* □

Proof. By induction on derivations. □

In $\mathbf{Ob}^{\downarrow\uparrow}$, every pair of types A and A' not only has a least upper bound (*lub*), as in many other systems, but also has a greatest lower bound (*glb*). A natural question that arises is whether the universe of types forms a lattice with respect to the subtyping relation. In other words, whether there exist definable object types $\bot$ and $\top$ such that for every type A, one has $\bot \leq A$ and $A \leq \top$. The answer is no: although – in the absence of constant types – we can take $\top \equiv [\,]$, there is no object type that can play the role of $\bot$. Consequently, system $\mathbf{Ob}^{\downarrow\uparrow}$ includes special types $\bot$ and $\top$ as primitive.

Definition 2.2 (Lubs and Glbs). Let the types $A \equiv [\ell_i : (B_i^u, B_i^s)^{i \in I}]$ and $A' \equiv [\ell_j : (C_j^u, C_j^s)^{j \in J}]$ be Split types. Lubs and glbs between A, A', $\top$ and $\bot$ are defined[5] as follows:

1. $\bot \sqcup A = A$, $\top \sqcup A = \top$,
2. $\top \sqcap A = A$, $\bot \sqcap A = \bot$,
3. $A \sqcup A' = [\ell_k : (B_k^u \sqcap C_k^u, B_k^s \sqcup C_k^s)^{k \in I \cap J}]$,
4. $A \sqcap A' = [\ell_k : (B_k^u \sqcup C_k^u, B_k^s \sqcap C_k^s)^{k \in I \cap J}, \ell_m : (B_m^u, B_m^s)^{m \in I - J}, \ell_n : (C_n^u, C_n^s)^{n \in J - I}]$. □

[5] Technically, this is not a proper definition, as $\sqcup$ and $\sqcap$ are defined recursively. A proper definition can be given by representing types as term automata and proving that the equations above are indeed satisfied (see [Pot98]).

Lemma 2.3. *For every pair of Split types A and A' we have $A \leq A \sqcup A'$ and $A' \leq A \sqcup A'$ and there exist no other type B, different from $A \sqcup A'$, such that $A \leq B$ and $A' \leq B$ and also $B \leq A \sqcup A'$. Similarly, for every Split types A and A' we have $A \sqcap A' \leq A$ and $A \sqcap A' \leq A'$ and there exist no other type C, different from $A \sqcap A'$, such that $C \leq A$ and $C \leq A'$ and also $C \leq A \sqcap A'$.* □

Proof. Follows directly from definition 2.2. □

3 Subject Reduction

Lemma 3.1 (Substitution). *If $E, x : C, E' \vdash \Im\{x\}$ and $E \vdash c : C$ then $E, E' \vdash \Im\{\{c\}\}$.* □

Proof. Easy induction on derivations. □

Lemma 3.2 (Bound Weakening). *If $E, x : C, E' \vdash \Im\{x\}$ and $C' \leq C$ then $E, x : C', E' \vdash \Im\{x\}$.* □

Proof. By induction on derivations. An interesting case is when $\Im\{x\}$ is $x : C$ and then $E, x : C, E' \vdash x : C$ is the conclusion of the (Val Var) rule. By (Val Var) we have $E, x : C', E' \vdash x : C'$, and by (Val Subsume) and the hypothesis $C' \leq C$ we can conclude $E, x : C', E' \vdash x : C$. □

The reduction relation $\rightsquigarrow$ is defined in [AC96]. We extend this reduction by adding the rule (Red Const) defined in the obvious way, i.e. $\vdash q \rightsquigarrow q$. Hence, a *result* is considered to be either a constant or an object. A theorem showing the absence of *stuck states* can be easily derived from the subject reduction theorem that follows.

Theorem 3.3 (Subject Reduction). *Let c be a closed term and v a result. Suppose $\vdash c \rightsquigarrow v$. If $\varnothing \vdash c : C$ then $\varnothing \vdash v : C$.* □

4 Encoding of the λ-calculus

In [AC96], the authors show that it is possible to encode the untyped λ-calculus into the untyped ς-calculus via a very simple transformation. They also explain the difficulties that result when types are added to the calculus. Specifically, they show that the expected subtyping relations between arrow types are *not* preserved due to the invariant subtyping enforced by their (Sub Object) rule.[6]

We consider the λ-calculus with constants. Terms in this calculus are specified by the usual grammar $a, b ::= q \,|\, x \,|\, \lambda(x)b \,|\, a(b)$. We let A, B range over the set of types defined by $A, B ::= Q \,|\, A \to B$. The transformation that follows is from [AC96], trivially extended to include constant terms.

[6] A solution to this problem is outlined with the introduction of an extended system with variance annotations.

Terms

$$\text{(Val Const)}\quad \frac{\mathsf{type}(q) = Q}{E \vdash q : Q} \qquad \text{(Val Var)}\quad \frac{E(x) = A}{E \vdash x : A}$$

$$\text{(Val Select)}\quad \frac{E \vdash a : A \quad \vdash A \leq [\ell_j : (\bot, D)]}{E \vdash a.\ell_j : D} \quad (A \equiv [\ell_i : (B_i^u, B_i^s)^{i \in I}])$$

$$\text{(Val Update)}\quad \frac{E \vdash a : A \quad \vdash A \leq [\ell_j : (D, \top)] \quad E, s : A \vdash b : D}{E \vdash a.\ell_j \Leftarrow \varsigma(s)\, b : A} \quad (A \equiv [\ell_i : (B_i^u, B_i^s)^{i \in I}])$$

$$\text{(Val Object)}\quad \frac{E, s : [\ell_i : (B_i^u, B_i^s)^{i \in I}] \vdash b_i : B_i^u \quad \vdash B_i^u \leq B_i^s \quad (\forall i \in I)}{E \vdash [\ell_i = \varsigma(s)\, b_i{}^{i \in I}] : A} \quad (A \equiv [\ell_i : (B_i^u, B_i^s)^{i \in I}],\ \ell_i \textit{ distinct})$$

Subtyping

$$\text{(Val Subsume)}\quad \frac{E \vdash a : A \quad \vdash A \leq A'}{E \vdash a : A'} \qquad \text{(Sub Object)}\quad \frac{\vdash C_j^u \leq B_j^u \quad \vdash B_j^s \leq C_j^s \quad (J \subseteq I)}{\vdash [\ell_i : (B_i^u, B_i^s)^{i \in I}] \leq [\ell_j : (C_j^u, C_j^s)^{j \in J}]}$$

$$\text{(Sub Refl)}\quad \frac{}{\vdash A \leq A} \qquad \text{(Sub Trans)}\quad \frac{\vdash A \leq B \quad \vdash B \leq C}{\vdash A \leq C}$$

$$\text{(Sub Top)}\quad \frac{}{\vdash A \leq \top} \qquad \text{(Sub Bot)}\quad \frac{}{\vdash \bot \leq A}$$

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

Definition 4.1 (Encoding of λ-terms). By induction on the structure of λ-terms,

1. $[\![\, q \,]\!] = q$,
2. $[\![\, x \,]\!] = x$,
3. $[\![\, \lambda(x)b\{x\} \,]\!] = [arg = \varsigma(s)s.arg,\ val = \varsigma(s)[\![\, b\{x\} \,]\!]\{\{x := s.arg\}\}]$,
4. $[\![\, a(b) \,]\!] = ([\![\, a \,]\!].arg := [\![\, b \,]\!]).val.$ □

For simplicity, we use the same notation $[\![\ ]\!]$ for both the encoding of types and the encoding of terms. Moreover, if E is a type environment then we let $[\![\, E \,]\!] = \{(x : [\![\, A \,]\!]) \mid (x : A) \in E\}$.

Definition 4.2 (Encoding of types). By induction on the structure of types,

1. $[\![\, Q \,]\!] = Q$,
2. $[\![\, A \to B \,]\!] = [arg{:}([\![\, A \,]\!], \top),\ val{:}(\bot, [\![\, B \,]\!])]$. □

Next, we show that the encoding preserves the subtyping relations by proving that any derivation in the λ-calculus with subtyping (system $\mathbf{F}_{\leq}$) can be encoded as a derivation in $\mathbf{Ob}^{\downarrow\uparrow}$. The following lemma follows directly from definition 4.2.

Lemma 4.3 (Preservation of Subtyping). *Let A and B be arbitrary types. If $\vdash A \to B \leq A' \to B'$ is derivable in $\mathbf{F}_{\leq}$ then $\vdash [\![\, A \to B \,]\!] \leq [\![\, A' \to B' \,]\!]$ is derivable in $\mathbf{Ob}^{\downarrow\uparrow}$.* □

Theorem 4.4 (Preservation of Typing). *Let a be an arbitrary λ-term and A an arbitrary type. If $E \vdash a : A$ is derivable in $\mathbf{F}_{\leq}$ then $[\![\, E \,]\!] \vdash [\![\, a \,]\!] : [\![\, A \,]\!]$ is derivable in $\mathbf{Ob}^{\downarrow\uparrow}$.* □

5 Type Inference

The type inference algorithm collects a set of subtyping constraints, that follow directly from the typing rules in Figure 1, and then checks that it is satisfiable. The types occurring in a constraint set are more general than those defined in section 2. In addition to object types, constant types, $\top$ and $\bot$, they may also include free type variables.

Definition 5.1 (Inference Types). An *inference type* is defined by the following productions:

$$\sigma, \tau ::= \alpha \mid \bot \mid \top \mid [\ell_i : (\tau_i, \sigma_i)^{\,i \in I}]$$ □

We use Greek letters towards the beginning of the alphabet such as $\alpha, \beta, ...$ to range over a set of type variables, and Greek letters towards the end of the alphabet such as

$\sigma, \tau, ...$ to range over inference types. A *substitution* is a mapping from the set of type variables to the set of inference types. We reserve the letter ρ to range over substitutions.

Definition 5.2 (Constraint Satisfaction). Let C be a constraint set over a set of inference types and let ρ be a substitution. We say that ρ is a solution to C, and write $\rho \models \mathsf{C}$, if for every constraint $\sigma \leq \tau$ in C it is the case that $\rho(\sigma) \leq \rho(\tau)$. □

The type inference rules are, essentially, the rules of the system $\mathbf{Ob}^{\downarrow\uparrow}$. They are formulated as rewriting rules for pairs of the form (J, C), where J is a set of judgements of the form $\Gamma \triangleright a : \tau$ and C is a set of subtyping constraints. Type inference is a accomplished by a sequence of rewritings guided by the rules in Figure 2.

Definition 5.3 (Inference Algorithm). The inference algorithm is defined by an initialization step followed by an iteration step. The input to the algorithm is an untyped term a.

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

Iterate. Let (J, C) be the current pair. If J is empty, then stop. Otherwise, select a judgement from J and rewrite it using the appropriate rule from figure 2. □

Lemma 5.4. *The inference algorithm from definition 5.3 always terminates with an empty set of judgements.* □

Proof. First observe that the algorithm is well defined: the only possibility for the rewriting process to get stuck is when the selected judgement is $\Gamma \triangleright x : \alpha$ and $x \notin \mathsf{Dom}(\Gamma)$. This may not happen, however, as $\mathsf{Dom}(\Gamma) = \mathsf{FV}(a)$ by construction, and an inspection of the rewriting rules it is easily verified that whenever $(\Gamma' \triangleright a' : \tau) \in \mathsf{J}$, one has $\mathsf{FV}(a') \subseteq \mathsf{Dom}(\Gamma')$. Termination is immediate using the measure on (J, C) pairs defined in Figure 3. Defining $|(\mathsf{J}, \mathsf{C})| = \Sigma_{\Im \in \mathsf{J}} |\Im|$, the claim follows by observing that this measure, i.e. $|(\mathsf{J}, \mathsf{C})|$, strictly decreases after each step of the rewriting process and it is bound from below by the value 0. □

5.1 Soundness and Completeness of Inference

The following generation lemmas are needed to prove the main theorem of this section. This theorem states that the type inference algorithm from definition 5.3 is sound and complete with respect to the typing derivations in $\mathbf{Ob}^{\downarrow\uparrow}$.

Lemma 5.5 (Generation Lemmas).

1. *If $E \vdash x : B$ is derivable, then $E(x) = A$ where A is a type such that $\vdash A \leq B$.*
2. *If $E \vdash a.\ell : B$ is derivable, then $E \vdash a : A$ is also derivable for some type A such that $\vdash A \leq [\ell : (\bot, B)]$.*
3. *If $E \vdash a.\ell \Leftarrow \varsigma(s)\, b : A$ is derivable, then there exist types A' and B such that $\vdash A' \leq [\ell : (B, \top)]$ and $\vdash A' \leq A$, and the judgements $E \vdash a : A'$ and $E, s : A' \vdash b : B$ are both derivable.*
4. *If $E \vdash [\ell_i = \varsigma(s)\, b_i{}^{i \in I}] : A$ is derivable, then there exist a type $A' \equiv [\ell_i : (B_i^u, B_i^s)^{i \in I}]$ such that for every $i \in I$, the judgements $E, s : A' \vdash b_i : B_i^u$ are derivable and so are $\vdash B_i^u \leq B_i^s$ and $\vdash A' \leq A$.* □

We say that a substitution ρ satisfies a pair (J, C), symbolically $\rho \models (\mathsf{J}, \mathsf{C})$, if for every judgement $\Gamma \triangleright a : \alpha$ in J we have $\rho \models \mathsf{C}$ and $\rho(\Gamma) \vdash a : \rho(\alpha)$ is derivable. Let $\Longrightarrow$ be the relation defined in Figure 2, and let $\Longrightarrow^*$ be its reflexive and transitive closure.

Lemma 5.6 (Rewriting is Sound). *Let (J, C) and $(\mathsf{J}', \mathsf{C}')$ be pairs such that $(\mathsf{J}, \mathsf{C}) \Longrightarrow (\mathsf{J}', \mathsf{C}')$. Every substitution ρ that satisfies $(\mathsf{J}', \mathsf{C}')$ also satisfies (J, C).* □

Lemma 5.7 (Rewriting is Complete). *Let (J, C) and $(\mathsf{J}', \mathsf{C}')$ be pairs such that $(\mathsf{J}, \mathsf{C}) \Longrightarrow (\mathsf{J}', \mathsf{C}')$. Every substitution ρ that satisfies (J, C) also satisfies $(\mathsf{J}', \mathsf{C}')$.* □

Theorem 5.8 (Inference is Sound and Complete). *For every term a and every type environment Γ such that $\mathsf{Dom}(\Gamma) = \mathsf{FV}(a)$. If $(\{\Gamma \triangleright a : \alpha\}, \varnothing) \Longrightarrow^* (\varnothing, \mathsf{C})$, then for every substitution ρ such that $\rho \models \mathsf{C}$, 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 $\mathsf{Dom}(E) = \mathsf{FV}(a)$, then $(\{\Gamma \triangleright a : \alpha\}, \varnothing) \Longrightarrow^* (\varnothing, \mathsf{C})$ and there exists a substitution ρ such that $\rho \models \mathsf{C}$ and $E = \rho(\Gamma)$ and $A = \rho(\alpha)$.* □

5.2 Implementation

In the current implementation, the inference algorithm works in a bottom-up fashion (with respect to the type derivation) by reducing the initial judgement to a set of subtyping constraints. This set of subtyping constrains is closed and checked for consistency by an incremental closure algorithm that is similar to that described in [Pot98], extended to support Split types. The closure algorithm is invoked each time a new constraint is added to the set. The constraint set is stored in the form of a constraint *graph* where each variable has a unique constructed bound, and a constraint *base* that stores constraints relating type variables. The use of variant subtyping over Split types makes it possible to use the aforementioned representation: two constraints over the same variable, such as $\sigma_1 \leq \alpha \leq \tau_1$ and $\sigma_2 \leq \alpha \leq \tau_2$, can always be merged to form $\sigma_1 \sqcup \sigma_2 \leq \alpha \leq \tau_1 \sqcap \tau_2$, as $\sigma_1 \sqcup \sigma_2$ and $\tau_1 \sqcap \tau_2$ are legal Split types.

$$
\begin{array}{lcl}
\textbf{(I-Val Var)}:\ \Gamma(x) = A & & \\
(\mathsf{J} \cup \{\Gamma \triangleright x : \alpha\}, \mathsf{C}) & \Longrightarrow & (\mathsf{J}, \mathsf{C} \cup \{A \leq \alpha\}) \\
\\
\textbf{(I-Val Select)}:\ \alpha \textit{ fresh} & & \\
(\mathsf{J} \cup \{\Gamma \triangleright a.\ell : \beta\}, \mathsf{C}) & \Longrightarrow & (\mathsf{J} \cup \{\Gamma \triangleright a : \alpha\}, \mathsf{C} \cup \{\alpha \leq [\ell : (\bot, \beta)]\}) \\
\\
\textbf{(I-Val Update)}:\ \alpha \textit{ fresh} & & \\
(\mathsf{J} \cup \{\Gamma \triangleright a.\ell \Leftarrow \varsigma(s)b : \gamma\}, \mathsf{C}) & \Longrightarrow & \left(\begin{array}{l} \mathsf{J} \cup \{\Gamma \triangleright a : \alpha, \Gamma, s : \alpha \triangleright b : \beta\}, \\ \mathsf{C} \cup \{\alpha \leq \gamma, \alpha \leq [\ell : (\beta, \top)]\} \end{array} \right) \\
\\
\textbf{(I-Val Object)}:\ \beta_i \textit{ and } \gamma_i \textit{ fresh} & & \\
(\mathsf{J} \cup \{\Gamma \triangleright [\ell_i = \varsigma(s)\, b_i\ ^{i \in I}] : \alpha\}, \mathsf{C}) & \Longrightarrow & \left(\begin{array}{l} \mathsf{J} \cup \{\Gamma, s : [\ell_i : (\beta_i, \gamma_i)\ ^{i \in I}] \triangleright b_i : \beta_i\}, \\ \mathsf{C} \cup \{[\ell_i : (\beta_i, \gamma_i)\ ^{i \in I}] \leq \alpha,\ \beta_i \leq \gamma_i\} \end{array} \right)
\end{array}
$$

Figure 2. Inference Rules.

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

Figure 3. Measure on (J, C) pairs.

The closure algorithm iteratively decomposes the constraints into their elementary components reporting a failure if an inconsistency is detected during the process. At the end of the inference process, a series of (polynomial time) simplification steps are applied to the graph to obtain a more compact representation. Again, variant subtyping over Split types permits us to include most of the simplifications described in [Pot98].

At the time of writing we do not have a precise estimate on the size of the final constraint graph resulting from type inference. Experience shows that the simplification methods of [Pot98] are just as effective for our system. In Figure 4, we give a sample run of the algorithm on a term from [AC96] that implements a restorable counter. After minimization, we can identify `Bs7` and `Bs8`, and consequently `Bu3` and `Bu4`; allowing us to display the result as in Figure 5.

As we mentioned in section 1.4, plans for future work include the definition of a systematic method for extracting a "flow type" from the simplified set of constraints computed by the algorithm. The "flow type" would not necessarily be a bona fide Split type. However, it seems possible (and desirable) to envisage a method for computing a textual representation of the constraint set by decorating the component types of a Split type with labels representing the interdependencies between them.

6 Relationships with other Type Systems

In [AC96], Abadi and Cardelli define a suite of type systems for the ς-calculus. What follows is a comparison between our system and some of the systems defined in that book.

Finite and Recursive Types. We have already shown, at least informally, that our system is more powerful than the system of recursive types (hence, more powerful than the system of finite types too). In fact, it is immediate to give a formal proof of this claim, noting (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.

Types with Variance Annotations. As an enhancement to the system of first-order 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 these components. Specifically, read-only methods can be subtyped covariantly while write-only methods can be subtyped contravariantly. Going back to Example 1.3, the terms p_0 and p_2' can be given the following types with variance annotations:

$$
\begin{array}{lcl}
p_0 & : & P_0^+ \equiv \mu(X)[\mathit{move}^+ : X] \\
p_2' & : & P_2^+ \equiv \mu(X)[x :, \mathsf{int}, y : \mathsf{int}, \mathit{move}^+ : X].
\end{array}
$$

```
counter = [cont = 0,
           get = @(s)s.cont,
           inc = @(s)(s.backup <= @(z)z.cont := s.cont).cont := s.cont+1,
           backup = @(s)s.cont := 0];
counter : T |
    [cont:(int,int), get:(int,int), inc:(Bu3,Bs7), backup:(Bu4,Bs8)] <= T,
    [cont:(int,int), get:(int,int), inc:(Bu3,Bs7), backup:(Bu4,Bs8)] <= Bs7,
    [cont:(int,int), get:(int,int), inc:(Bu3,Bs7), backup:(Bu4,Bs8)] <= Bs8,
    Bu4 <= Bs8, Bu3 <= Bs7
```

Figure 4. Type of a Restorable Counter.

```
counter : [cont:(int,int), get:(int,int), inc:(Bu,Bs), backup:(Bu,Bs)] |
          [cont:(int,int), get:(int,int), inc:(Bu,Bs), backup:(Bu,Bs)] <= Bs,
          Bu <= Bs
```

Figure 5. Simplified Type of a Restorable Counter.

Now, the subtyping rules for types with variant annotations validate the relationship $P_2^+ \leq P_0^+$, and therefore allow the following 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. We illustrate the idea using finite types, although the same reasoning applies to recursive types just as well. 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 $i \in I$,

- $B_i^u = B_i$ and $B_i^s = \top$ when $\nu_i = {}^-$,
- $B_i^u = \bot$ and $B_i^s = B_i$ when $\nu_i = {}^+$,
- $B_i^u = B_i^s = B_i$ 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 $\bot$. The type $\bot$ (viewed as a set of terms) is not inhabited by any term and, consequently, updates to read-only method are not allowed.

In accordance to the encoding just outlined, it is not difficult to verify that the following types can be derived for the terms p_0 and p_2':

$$\begin{array}{lcl} p_0 & : & \mu(X)[\mathit{move} : (\bot, X)] \\ p_2' & : & \mu(X)[x : (\mathsf{int}, \mathsf{int}), y : (\mathsf{int}, \mathsf{int}), \mathit{move} : (\bot, X)] \end{array}$$

As expected, these types validate the desired subtyping relationships, but prevent updates to the *move* method. However, using Split types, we can find a more flexible typing that validates the desired subtyping relationships and still allows updates to the *move* method (see Example 1.3 in Section 1.2).

Self Types. The relationship between our system and the system of Self Types from [AC96] is a subtle one. The system of Self Types is built around two main ideas.[7] First, object 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 type,[8]

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

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 in the bounds and covariant in the bodies. 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. This abstraction over the type of self, restricts the way by which methods returning self can be updated. The typing rule for method update is the following,

$$\frac{(A \equiv \varsigma(X)[..., \ell : B\{X\}, ...]) \quad \dfrac{E \vdash a : A \quad E, Y \leq A, s : Y \vdash b : B\{\{Y\}\}}{E \vdash a.\ell \Leftarrow \varsigma(s)b : A}}{}$$

[7] We are referring the the system of *Primitive Covariant Self Types* in Chap. 16 of [AC96].

[8] Abadi and Cardelli introduce a new binder for Self types, and denote this type as $\varsigma(X)[x : \mathsf{int}, y : \mathsf{int}, \mathit{move} : X]$.

The intuitive reading of this rule is the following. 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 ℓ of a, we can only assume that the actual type of the object (hence of the self variable s) is *some* type $Y \leq A$. Furthermore, if the original type of ℓ 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.

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.

EXAMPLE 6.1. Consider again the terms from 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 define o as follows,

$$o = [\ell = p_2'].\ell := p_0$$

In that example, we have shown that the term o can be assigned the Split type $[\ell : S_0 \sqcup S_2]$ where $S_0 \sqcup S_2 \equiv \mu(X)[\mathit{move} : (S_0 \sqcap S_2, X)]$. Consider then the term $(o.\ell).\mathit{move} \Leftarrow \varsigma(s)s$, which is typable with Self types.[9] Since $o : [\ell : S_0 \sqcup S_2]$, it follows by (Val Select) and by unfolding the type that $o.\ell : [\mathit{move} : (S_0 \sqcap S_2, S_0 \sqcup S_2)]$. To type $(o.\ell).\mathit{move} \Leftarrow \varsigma(s)s$ we must use the rule (Val Update). To prove the premises of this rule, we need to show that $s : S_0 \sqcup S_2 \vdash s : D$ for a some Split type D such that $\vdash [\mathit{move} : (S_0 \sqcap S_2, S_0 \sqcup S_2)] \leq [\mathit{move} : (D, \top)]$. By definition of subtyping over Split types, this implies that $\vdash D \leq S_0 \sqcap S_2$. Consequently, since $\vdash S_0 \sqcap S_2 \leq S_0 \sqcup S_2$ then, by transitivity, $\vdash D \leq S_0 \sqcup S_2$ and this immediately shows that with the assumption $s : S_0 \sqcup S_2$ it is impossible to prove that $s : D$, i.e. it is impossible to prove $s : S_0 \sqcup S_2 \vdash s : D$.

It could of course be argued that there might exist other types for o that would turn $(o.\ell).\mathit{move} \Leftarrow \varsigma(s)s$ into a typable term in our system. Unfortunately, running the inference algorithm on this term – which we have proven to be complete – shows that this is not the case. To ease the notation we use the following shorthands: $S_2(\gamma, \beta) = [x : (\mathsf{int}, \mathsf{int}),\ y : (\mathsf{int}, \mathsf{int}),\ \mathit{move} : (\gamma, \beta)]$ and $S_0(\gamma, \beta) = [\mathit{move} : (\gamma, \beta)]$. The result of running the algorithm on the term o is shown in figure 6. The constraint $\gamma \leq [y : (\bot, \mathsf{int})]$ in the typing of p_2' results from the dependency of the method x on the field y. This constraint forces any update on the method *move* of p_2' to provide an object with a field y: this is required for soundness, as the method x assumes that a call to *move* returns an object with a field y. The problem is that the inference algorithm carries this constraint along in the typing of o as well, even though the type of o does not mention x, which therefore may no longer be invoked. Continuing our experiment, we then have:

$$o.\ell : [\mathit{move} : (\gamma, \beta)] \quad | \\ \{[\mathit{move} : (\gamma, \beta)] \leq \beta,\ \gamma \leq [y : (\bot, \mathsf{int})],\ \gamma \leq \beta\}$$

It follows that, to type the update $(o.\ell).\mathit{move} \Leftarrow \varsigma(s)s$, we need to derive the judgement $s : [\mathit{move} : (\gamma, \beta)] \vdash s : D$ for some type D such that $\vdash [\mathit{move} : (\gamma, \beta)] \leq [\mathit{move} : (D, \top)]$, i.e., such that $\vdash D \leq \gamma$. From the last constraint, and from $\gamma \leq [y : (\bot, \mathsf{int})]$, by transitivity, $D \leq [y : (\bot, \mathsf{int})]$. On the other hand, the judgement $s : [\mathit{move} : (\gamma, \beta)] \vdash s : D$ is only derivable if $[\mathit{move} : (\gamma, \beta)] \leq D$. In other words, the typing of the update would require a type D such that $[\mathit{move} : (\gamma, \beta)] \leq D \leq [y : (\bot, \mathsf{int})]$. Clearly, no such type exists and therefore the term $(o.\ell).\mathit{move} \Leftarrow \varsigma(s)s$ is not typable in our system. □

The question of whether type inference for Self Types is decidable, is still open. This problem is believed to be at least as complex as type inference for the system of [Pal95], due to the underlying interpretation of Self types in terms of bounded existentials.

[9] This term can be given the type $\varsigma(X)[\mathit{move} : X]$ in the Self types system.

Acknowledgements

The idea of Split Types was inspired by Francois Pottier's work on type inference for ML. We would like to thank Jens Palsberg for exposing us to that work, and for his insightful comments and suggestions. Special thanks to Craig Chambers, Patric Cousot and Alan Mycroft for discussions and feedback at the "Workshop on Types and Abstract Interpretation" held in Padova, Italy, on May 17-18 1999. Also, to the members of the Church Group, in particular to Assaf J. Kfoury for reading earlier drafts and giving us feedback. The simplification algorithms used in the inference algorithm were implemented by Marina Baldan as part of her Laurea thesis at the Department of Mathematics of the University of Padova.

References

[AC93] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

$$
\begin{array}{rcl}
p_0 : S_0(\gamma,\beta) & | & \{S_0(\gamma,\beta) \le \beta, \gamma \le \beta\} \\
p_2' : S_2(\gamma,\beta) & | & \{S_2(\gamma,\beta) \le \beta,\ \gamma \le [y : (\bot, \mathsf{int})],\ \gamma \le \beta\} \\
[\ell = p_2] : [\ell : (S_2(\gamma,\beta), S_2(\gamma,\beta))] & | & \{S_2(\gamma,\beta) \le \beta,\ \gamma \le [y : (\bot, \mathsf{int})],\ \gamma \le \beta\} \\
o : [\ell : (S_0(\gamma,\beta),\ S_0(\gamma,\beta))] & | & \{S_0(\gamma,\beta) \le \beta,\ \gamma \le [y : (\bot, \mathsf{int})],\ \gamma \le \beta\}
\end{array}
$$

Figure 6. Inferred Type for $[\ell = p_2'].\ell := p_0$.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AW93] A. S. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93, Conf. Funct. Program. Lang. Comput. Arch.*, pp. 31–41. ACM, 1993.

[BCM+93] 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.

[CC91] F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Inf. & Comput.*, 92:48–80, 1991.

[EST95a] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, pp. 169–184, 1995.

[EST95b] 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.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions American Math. Society*, 146:29–60, 1969.

[McA96] D. McAllester. Inferring recursive data types. Unpublished, 1996.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MTHM90] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1990.

[Pal95] J. Palsberg. Efficient inference of object types. *Inf. & Comput.*, 123:198–209, 1995.

[PJ97] J. Palsberg and T. Jim. Type inference with simple selftypes is np-complete. *Nordic Journal of Computing*, 1997.

[PJHH+93] 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.

[Pot98] F. Pottier. *Type Inference in the Presence of Subtyping: from Theory to Practice*. PhD thesis, Université Paris VII, 1998.

[RV98] D. Rémy and J. Vouillon. Objective ml: An effective object-oriented extension to ml. *Theory and Practice of Object Systems*, 1998.

[TS96] V. Trifonov and S. Smith. Subtyping constrained types. In *Proc. 3rd Int'l Static Analysis Symp.*, pp. 349–365, 1996.

[TW93] 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.

A Proofs

Proof for theorem 3.3 (Subject Reduction). By induction on the derivation $\vdash c \rightsquigarrow v$. The cases (Red Const) and (Red Object) are immediate, as in both cases $c \equiv v$. The remaining cases are discussed below.

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

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

Since $\vdash a \rightsquigarrow v'$ and $\varnothing \vdash a : A$, by induction hypothesis we have $\varnothing \vdash v' : A$. This last judgement must have been derived as follows:

$$\dfrac{\begin{array}{c}\text{(Val Object)}\\ s : A' \vdash b_i\{s\} : C_i^u \quad \vdash C_i^u \leq C_i^s \quad (\forall i \in I)\end{array}}{\begin{array}{c}\varnothing \vdash v' \equiv [\ell_i = \varsigma(s)\, b_i{}^{i\in I}] : A'\\ \vdots\ (A' \leq A)\\ \varnothing \vdash v' \equiv [\ell_i = \varsigma(s)\, b_i{}^{i\in I}] : A\end{array}}$$

for some $A' \equiv [\ell_i : (C_i^u, C_i^s)^{i\in I}]$. Since $j \in I$, from $s : A' \vdash b_j\{s\} : C_j^u$ and $\varnothing \vdash v' \equiv [\ell_i = \varsigma(s)\, b_i{}^{i\in I}] : A'$ by lemma 3.1 it follows that $\varnothing \vdash b_j\{\{v'\}\} : C_j^u$. By induction hypothesis, we have $\varnothing \vdash v : C_j^u$. Since $\vdash C_j^u \leq C_j^s$ and $\vdash A' \leq A$ and also $\vdash A \leq [\ell_j : (\bot, D)]$, it follows that $\vdash C_j^u \leq C_j^s \leq D \leq C$. Hence, using (Val Subsume) we have $\varnothing \vdash v : C$.

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

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

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

$$\dfrac{\begin{array}{c}\text{(Val Object)}\\ s : A' \vdash b_i : C_i^u \quad \vdash C_i^u \leq C_i^s \quad (\forall i \in I)\end{array}}{\begin{array}{c}\varnothing \vdash [\ell_i = \varsigma(s)\, b_i{}^{i\in I}] : A'\\ \vdots\ (A' \leq A)\\ \varnothing \vdash [l_i = \varsigma(s)\, b_i{}^{i\in I}] : A\end{array}}$$

Because $s : A \vdash b : D$ and $\vdash A' \leq A$, by lemma 3.2 it follows that $s : A' \vdash b : D$. Furthermore, since $\vdash A' \leq A \leq [\ell_j : (D, \top)]$ we have $\vdash D \leq C_j^u$ and by (Val Subsume) we derive $s : A' \vdash b : C_j^u$. 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 $\vdash A' \leq A \leq C$. □

Proof for theorem 4.4 (Preservation of Typing). By induction on the derivation $E \vdash a : A$ in $\mathbf{F}_{\leq}$. Let (Constant), (Variable), (Abstraction), (Application) and (Subsumption) be the names of the typing rules in $\mathbf{F}_{\leq}$. The proofs for **(Constant)** and **(Variable)** follow immediately from the definitions.

(Abstraction) Suppose $E \vdash \lambda(x)b\{x\} : A \to B$ is derivable in $\mathbf{F}_{\leq}$. Then we have $E, x : A \vdash b\{x\} : B$ and, by induction hypothesis, $[\![E, x : A]\!] \vdash [\![b\{x\}]\!] : [\![B]\!]$ is derivable in $\mathbf{Ob}^{\downarrow\uparrow}$. The judgements

(1) $[\![E]\!], s : [arg : ([\![A]\!], [\![A]\!]), val : ([\![B]\!], [\![B]\!])], x : [\![A]\!] \vdash [\![b\{x\}]\!] : [\![B]\!]$

(2) $[\![E]\!], s : [arg : ([\![A]\!], [\![A]\!]), val : ([\![B]\!], [\![B]\!])] \vdash s.arg : [\![A]\!]$

are derivable in $\mathbf{Ob}^{\downarrow\uparrow}$. Judgement (1) follows from the induction hypothesis and judgement (2) is easily provable. From (1) and (2), using lemma 3.1, we have

$$[\![E]\!], s : [arg : ([\![A]\!], [\![A]\!]), val : ([\![B]\!], [\![B]\!])] \vdash [\![b\{x\}]\!]\{\{x := s.arg\}\} : [\![B]\!].$$

Since $[\![\lambda(x)b\{x\}]\!] = [arg = \varsigma(s)s.arg, val = \varsigma(s)[\![b\{x\}]\!]\{\{x := s.arg\}\}]$, then by (Val Object) we have $[\![E]\!] \vdash [\![\lambda(x)b\{x\}]\!] : [arg : ([\![A]\!], [\![A]\!]), val : ([\![B]\!], [\![B]\!])]$. Consequently, it follows by (Val Subsume) that $[\![E]\!] \vdash [\![\lambda(s)b\{x\}]\!] : [arg : ([\![A]\!], \top), val : (\bot, [\![B]\!])]$.

(Application) Suppose $E \vdash a(b) : B$ is derivable in $\mathbf{F}_{\leq}$. Then, it must be $E \vdash a : A \to B$ and $E \vdash b : A$. By induction hypothesis, $[\![E]\!] \vdash [\![a]\!] : [\![A \to B]\!]$ and $[\![E]\!] \vdash [\![b]\!] : [\![A]\!]$. By definition, we have $[\![A \to B]\!] = [arg : ([\![A]\!], \top), val : (\bot, [\![B]\!])]$ and $[\![a(b)]\!] = ([\![a]\!].arg := [\![b]\!]).val$. Since $[\![E]\!] \vdash [\![a]\!] : [arg : ([\![A]\!], \top), val : (\bot, [\![B]\!])]$ and $[\![E]\!] \vdash [\![b]\!] : [\![A]\!]$, then it follows by (Val Update) that $[\![E]\!] \vdash [\![a]\!].arg := [\![b]\!] : [arg : ([\![A]\!], \top), val : (\bot, [\![B]\!])]$. From the last judgment, by (Val select) we conclude that $[\![E]\!] \vdash ([\![a]\!].arg := [\![b]\!]).val : [\![B]\!]$.

(Subsumption) Immediate from the induction hypothesis and from lemma 4.3. □

Proof for theorem 5.6 (Rewriting is sound). By a case analysis on the rewriting step in question.

(I-Val Var) Let ρ be a substitution such that $\rho \models (\mathsf{J}, \mathsf{C} \cup \{A \leq \alpha\})$. Then clearly $\rho \models (\mathsf{J}, \mathsf{C})$ and $\rho(A) \leq \rho(\alpha)$, and the format of the rewriting in question implies $\Gamma(x) = A$. Then $\phi(\Gamma) \vdash a : \phi(\alpha)$ is derivable by (Val Var) and (Val Subsume), and this proves the claim.

(I-Val Select) Let ρ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a : \alpha\}, \mathsf{C} \cup \{\alpha \leq [\ell : (\bot, \beta)]\})$. Then $\rho(\Gamma) \vdash a : \rho(\alpha)$ is derivable, and $\rho(\alpha) \leq [\ell : (\bot, \rho(\beta))]$. Then $\rho(\Gamma) \vdash a.\ell : \rho(\beta)$ derives from (Val Select): this proves the claim as it implies that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a.\ell : \beta\}, \mathsf{C})$.

(I-Val Update) Let ρ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a : \alpha, \Gamma, s : \alpha \triangleright b : \beta\}, \mathsf{C} \cup \{\alpha \leq \gamma, \alpha \leq [\ell : (\bot, \beta)]\})$. Then $\rho(\alpha) \leq \rho(\gamma)$ and $\rho(\alpha) \leq [\ell : (\bot, \rho(\beta))]$, and the two judgements $\rho(\Gamma) \vdash a : \rho(\alpha)$ and $\rho(\Gamma), s : \rho(\alpha) \vdash b : \rho(\beta)$ are derivable. Then $\rho(\Gamma) \vdash a.\ell \Leftarrow \varsigma(s)b : \rho(\gamma)$ derives from (Val Update) and (Val Subsume), which again proves the claim.

(I-Val Object) Let $\rho \models (\mathsf{J} \cup \{\Gamma, s : [\ell_i : (\beta_i, \gamma_i)^{\,i \in I}] \triangleright b_i : \beta_i\}, \mathsf{C} \cup \{[\ell_i : (\beta_i, \gamma_i)^{\,i \in I}] \leq \alpha,\ \beta_i \leq \gamma_i\})$ Then $[\ell_i : (\rho(\beta_i), \rho(\gamma_i))^{\,i \in I}] \leq \alpha$ and $\rho(\beta_i) \leq \rho(\gamma_i)$, and the judgements $\Gamma, s : [\ell_i : (\rho(\beta_i), \rho(\gamma_i))^{\,i \in I}] \vdash b_i : \rho(\beta_i)$ are all derivable. Then $\rho(\Gamma) \vdash [\ell_i = \varsigma(s)\, b_i{}^{\,i \in I}] : \rho(\alpha)$ derives from (Val Object) and (Val Subsume). □

Proof for theorem 5.6 (Rewriting is complete). By a case analysis on the rewriting step in question.

(Val Var) Let ρ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright x : \alpha\}, \mathsf{C})$. By definition $\rho(\Gamma) \vdash x : \rho(\alpha)$ is derivable. By Lemma 5.5.1 $(\rho(\Gamma))(x) = A$ for a type A such that $A \leq \rho(\alpha)$. Thus $\rho(\mathsf{J}, \mathsf{C} \cup \{A \leq \alpha\})$ as desired.

(Val Select) Let ρ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a.\ell : \beta\}, \mathsf{C})$. By definition $\rho(\Gamma) \vdash a.\ell : \rho(\beta)$ is derivable, and, by Lemma5.5.2, $\rho(\Gamma) \vdash a : A$ is also derivable for a type A such that $A \leq [\ell : (\bot, \rho(\beta))]$. Then define $\rho' = \rho \cup \{\alpha \mapsto A\}$ where α is the fresh variable chosen by the rewriting in question: now, $\rho' \models (\mathsf{J}', \mathsf{C}')$ by construction.

(Val Update) Let ρ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright a.\ell \Leftarrow \varsigma(s)b : \gamma\}, \mathsf{C})$. By definition $\rho(\Gamma) \vdash a.\ell \Leftarrow \varsigma(s)b : \rho(\gamma)$ is derivable. By Lemma 5.5.3, $\rho(\Gamma) \vdash a : A'$ and $\rho(\Gamma), s : A' \vdash b : B$ are both derivable, for two types type A' and B such that $A' \leq [\ell : (B, \top)]$ and $A' \leq A$. Then, define $\rho' = \rho \cup \{\alpha \mapsto A', \beta \mapsto B\}$ where α and β are the fresh variables chosen by the rewriting in question: as in the previous case, $\rho' \models (\mathsf{J}', \mathsf{C}')$ by construction.

(Val Object) Let ρ be a substitution such that $\rho \models (\mathsf{J} \cup \{\Gamma \triangleright [\ell_i = \varsigma(s)\, b_i{}^{\,i \in I}] : \alpha\}, \mathsf{C})$. By definition $\rho(\Gamma) \vdash [\ell_i = \varsigma(s)\, b_i{}^{\,i \in I}] : \rho(\alpha)$. By Lemma 5.5.4, the judgements $\rho(\Gamma), s : [\ell_i : (B_i^u, B_i^s)^{\,i \in I}] \vdash b_i : B_i^u$ are all derivable and $B_i^u \leq B_i^s$. Then define $\rho' = \rho \cup \{\beta_i \mapsto B_i^u, \gamma_i \mapsto B_i^s\}$ where β_i and γ_i are the fresh variables chosen by the rewriting in question: again, $\rho' \models (\mathsf{J}', \mathsf{C}')$ by construction. □

Proof for theorem 5.8 (Inference is sound and complete). Take a substitution $\rho \models \mathsf{C}$. By definition, $\rho \models (\varnothing, \mathsf{C})$, and by Lemma 5.6 (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, Γ and α as specified by the algorithm, and define a substitution ρ 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. Finally, $\rho \models (\varnothing, \mathsf{C})$, by Lemma 5.7, and hence $\rho \models \mathsf{C}$. □