Algorithmic aspects of type inference with subtypes

Patrick Lincoln *
Computer Science Department
Stanford University
Stanford, CA 94305
lincoln@cs.stanford.edu

John C. Mitchell †
Computer Science Department
Stanford University
Stanford, CA 94305
mitchell@cs.stanford.edu

Abstract

We study the complexity of type inference for programming languages with subtypes. There are three language variations that effect the problem: (i) basic functions may have polymorphic or more limited types, (ii) the subtype hierarchy may be fixed or vary as a result of subtype declarations within a program, and (iii) the subtype hierarchy may be an arbitrary partial order or may have a more restricted form, such as a tree or lattice. The naive algorithm for inferring a most general polymorphic type, under variable subtype hypotheses, requires deterministic exponential time. If we fix the subtype ordering, this upper bound grows to nondeterministic exponential time. We show that it is NP-hard to decide whether a lambda term has a type with respect to a fixed subtype hierarchy (involving only atomic type names). This lower bound applies to monomorphic or polymorphic languages. We give PSPACE upper bounds for deciding polymorphic typability if the subtype hierarchy has a lattice structure or the subtype hierarchy varies arbitrarily. We also give a polynomial time algorithm for the limited case where there are of no function constants and the type hierarchy is either variable or any fixed lattice.

1 Introduction

Subtyping is a basic feature of typed object-oriented languages, such as C^{++} and *Eiffel* [Str86, Mey88], and also occurs in many other languages in limited cases such as the relation between integer and real (or floating-point) numbers. In 1984, the second author described an algorithm for Milner-style type inference with subtyping [Mit84, Mit91]. Given a pure, untyped lambda term, this algorithm finds a most general typing statement that describes the set of possible typings with respect to any subtype hierarchy. Various aspects of the algorithm have been studied by other authors, with Fuh and Mishra elaborating algorithmic alternatives [FM90] and Wand and O'Keefe studying the computational complexity of typability [WO89]. An extension with polymorphic record operations [JM88] has been implemented by Jategaonkar [Jat89]. Unfortunately, the straightforward implementation of the type inference algorithm with subtypes requires exponential time, even in the absence of polymorphic `let` declarations (see [KMM91]). This may be an obstacle to practical type inference for object-oriented languages. It is therefore important to investigate the inherent complexity of type inference and type checking in the presence of subtypes.

The most general typing assertion about a pure lambda term (without constant symbols) may have exponential size, even using concise directed acyclic graph (dag) representations of type expressions. Consequently, it is not possible to compute most general types with subtyping in less than deterministic exponential time. However, the related "decision problem" of determining whether a term has any type might be solved more efficiently. By comparison, even though the Curry-type of a pure lambda term without subtyping can be exponential, when written as a string, there exist linear-size dag representations and linear algorithms that decide Curry-typability [KMM91]. The decision problem is relevant to practice since an efficient decision procedure could verify the absence of type errors at compile time without printing most-general types. Since this is the only form of typing problem that could be solved in less than exponential time, we focus on decision problems for type inference.

There are three language variations that have an effect on the complexity of typing:

* Supported in part AT&T Bell Laboratories Doctoral Scholarship and sources listed under †.

† Supported in part by an NSF PYI Award, matching funds from Digital Equipment Corporation, the Powell Foundation, and Xerox Corporation; NSF grant CCR-8814921 and the Wallace F. and Lucille M. Davis Faculty Scholarship.

Appeared in Proc ACM Symp. Principles of Programming Languages, 1992.

- Term constants may be polymorphic functions or restricted to monomorphic functions or atomic data (non-functions).
- The subtype hierarchy may be fixed or vary as a result of subtype declarations within a program.
- The subtype hierarchy may be an arbitrary partial order or may have a more restricted form, such as a tree or lattice.

If term constants have restricted functionality, this may simplify the type inference problem. Therefore, when possible, we prove lower bounds for restricted term constants and upper bounds for polymorphic types.

A subtle issue is the relationship between the subtype hierarchy at the point of declaration of some identifier and the subtype hierarchy at a possible point of use. For example, consider a function f of two arguments that requires the type of the first argument to be a subtype of the type of the second. An implicit assumption in [Mit84, Mit91] is that the appropriate typing statement to infer about f is some formalization of this English description, regardless of whether types A and B with A a subtype of B have been declared. The reason is that we may want to call the function f in some scope where two such types have been declared. Therefore, the type inference algorithm given in [Mit84, Mit91] deduces a most general typing statement that includes arbitrary assumptions about the relationships between types of function parameters. While this seems reasonable for pure lambda terms without constant symbols, the situation becomes more complicated in a realistic programming language. This is discussed in Section 3.

The main lower bound in this paper is that it is NP-hard to decide whether a lambda expression with constants has a type, given a set of subtyping relationships between ground (atomic) types. This applies to polymorphic and monomorphic languages, and languages without functional constants. This lower bound improves the main result of [WO89], which requires a constant with a polymorphic type. We also observe that if type parameters and subtype assumptions are given explicitly in the syntax of terms, it follows from the results in [Tiu91] that deciding whether an explicitly-typed term has a type is PSPACE hard.

We give two algorithms for the decision problem. The more general algorithm applies to terms with arbitrary constants, but assumes either that the subtype hierarchy may vary arbitrarily or that the fixed subtype hierarchy is a lattice. (Either condition makes it possible to determine in polynomial space whether an exponential-size set of subtype assumptions is satisfiable.) In the special case that there are no functional constants and the subtype hierarchy is either varying or is a fixed lattice, our second algorithm solves the problem in linear time. Since this case is NP-hard for arbitrary partial orders, our results emphasize the value of restricting the subtype relation to obtain practical typing algorithms.

Further discussion of the relevant language characteristics and their relationship to type inference is given in Section 3, following the preliminary definitions in Section 2. The lower bound is presented in Section 4 and the upper bounds in Sections 5 and 6.

For those familiar with [WO89], we note that their claim that the decision problem reduces to the partial order problem PO-SAT has been retracted [Wan91]. This invalidates both the claimed NP algorithm for the general problem and the claimed polynomial algorithm when the subtype hierarchy is a tree.

2 Preliminaries

We review the essential definitions and results from [Mit84, Mit91]. We study typing algorithms for untyped lambda terms, possibly containing constant symbols. Lambda terms are formed according to the grammar

$$M \ ::= \ x \,|\, c \,|\, M_1 M_2 \,|\, \lambda x.M,$$

where x may be any variable, c a constant symbol, $M_1 M_2$ is the application of M_1 to M_2 and $\lambda x.M$ is a lambda abstraction defining a function.

For simplicity, we only consider function types, written using type variables and type constants. Type expressions have the form

$$\tau \ ::= \ t \,|\, \theta \,|\, \tau_1 \rightarrow \tau_2$$

where t may be any type variable, θ a type constant, and $\tau_1 \rightarrow \tau_2$ is the type of functions from τ_1 to τ_2.

A subtype assertion has the form $\sigma \preceq \tau$. The standard meaning of an assertion $\sigma \preceq \tau$ is that σ is a subset of τ. An alternative interpretation that we will not discuss in any detail is that there is some coercion function $f_{\sigma \rightarrow \tau}$ which transforms values of type σ into values of type τ. Some discussion of this alternative may be found in [Mit91].

An *atomic* subtype assertion is a statement $a \preceq b$, where a and b are either type variables or type constants. All of our subtyping hypotheses will be atomic. Without this assumption, subtyping hypotheses such as the pair $b \preceq b \rightarrow b$ and $b \rightarrow b \preceq b$ would express "domain equations," and therefore allow all pure lambda terms (terms without constants) to be typed (see [Mit91]).

Terms and types will be written over some selected signature. A *signature* $\Sigma = \langle B, S, T \rangle$ is a triple consisting of a set B of type constants, a set S of atomic subtype assertions about type constants in B, and a set T of term constants, each with a specific type built

from type variables, type constants from B and $\rightarrow$. We say a term constant $c{:}\sigma$ is *polymorphic* if σ contains one or more type variables and *non-polymorphic* otherwise.

Intuitively, two type expressions *match* if they have the same shape. This does not involve any substitutions. More specifically, we define matching as follows: if σ is a type variable or type constant, then σ matches τ if and only if τ is a type variable or type constant; if $\sigma = \sigma_l \rightarrow \sigma_r$, then σ matches τ if and only if $\tau = \tau_l \rightarrow \tau_r$ and σ_l matches τ_l, and σ_r matches τ_r. The entailment relation, $\vdash$, on subtype assertions is defined by the following proof system. Note that if C is a set of atomic subtype assertions, and $C \vdash \sigma \preceq \tau$, then σ matches τ.

C $$C \cup \{\sigma \preceq \tau\} \vdash \sigma \preceq \tau$$

R $$C \vdash \sigma \preceq \sigma$$

T $$\frac{C \vdash \sigma \preceq \tau \qquad C \vdash \tau \preceq \gamma}{C \vdash \sigma \preceq \gamma}$$

Arrow $$\frac{C \vdash \sigma \preceq \tau \qquad C \vdash \alpha \preceq \gamma}{C \vdash \tau \rightarrow \alpha \preceq \sigma \rightarrow \gamma}$$

The **C** rule allows subtype assumptions to be used in a derivation. The **R** rule is reflexivity of $\preceq$, **T** is transitivity, and the **Arrow** rule gives subtyping for function types. Note that function types are antimonotonic in the left, or argument position, and monotonic in the right, or result position. If C and C' are sets of subtype assertions, we write $C \vdash C'$ to indicate that $C \vdash \sigma \preceq \tau$ for every $\sigma \preceq \tau$ in C'.

A *typing statement* is a formula $C, A \vdash M{:}\sigma$, where C is a set of atomic subtype assertions, A is a set of type assumptions of the form $x{:}\sigma$, where x is a term variable, M is an untyped lambda term, and σ is a type expression. The typing statement $C, A \vdash M : \sigma$ may be read as, "Under the subtype assumptions C and assumptions A about the types of variables, the term M has type σ."

The following proof rules determine typability with respect to any signature $\Sigma = \langle B, S, T\rangle$. The subtype proof system enters through the **Sub,** or "subsumption" rule. In **Const,** R may be any substitution of type expressions over Σ for type variables.

Const $$C, A \vdash c{:}\, R\sigma \qquad (c{:}\sigma \in T)$$

Var $$C, A \cup \{x : \sigma\} \vdash x : \sigma$$

App $$\frac{C, A \vdash M : \sigma \rightarrow \tau \qquad C, A \vdash N : \sigma}{C, A \vdash (M\, N) : \tau}$$

Abs $$\frac{C, A \cup \{x : \sigma\} \vdash M : \tau}{C, A \vdash (\lambda x.M) : \sigma \rightarrow \tau} \qquad (x \notin A)$$

Sub $$\frac{C, A \vdash M : \sigma \qquad C \cup S \vdash \sigma \preceq \tau}{C, A \vdash M : \tau}$$

The **Const** rule allows a typed constant from the signature to be given any substitution instance of its specified type. (If $c : \sigma$ is non-polymorphic, then the substitution R will have no effect.) The **Var**, **App**, and **Abs** rules are standard. The **Sub** rule forces a term with one type to belong to every supertype. We say that a typing statement $C, A \vdash M : \sigma$ is *provable with respect to signature* $\Sigma = \langle B, S, T\rangle$ if all of the term constants in M appear in T and all uses of **Const** and **Sub** in the derivation of the typing statment are in accordance with the signature.

As stated in [Mit84] and proved in [Mit91], one may normalize proofs of typing statements so that the only uses of the **Sub** rule are immediately following uses **Var** and **Const**. That is, the steps in any proof of a typing statement may be permuted so that the **Sub** rule only appears at the leaves of the proof. This property is important because the other four inference rules of this system are syntax-directed. That is, there is at most one normal proof of any type assertion up to uses of **Sub**. This property is used in the typing algorithms in [JM88, Mit91] and in all algorithms discussed in this paper. An alternative way of stating this proof normalization property is that the rules above are equivalent to the proof system obtained by eliminating **Sub** and replacing **Var** and **Const** by variants that allow a constant or variable to be given any supertype of its given type.

If R is a substitution of types for type variables, then we say R *respects a set* C *of atomic subtyping assertions* if, for every $a \preceq b$ in C, the type expression Ra matches Rb. If R respects atomic C, then there is a set C' of atomic subtype assertions such that $C' \vdash Ra \preceq Rb$ for every $a \preceq b$ in C, and if C'' is another set of atomic subtype assertions with this property, then $C' \vdash C''$. We write $R \cdot C$ for any such "minimal" set of atomic subtype assertions. The set $R \cdot C$ is efficiently computable from R and C, as outlined in [Mit84, Mit91]. If A is a set of assumptions about the types of variables, then RA is the set $RA = \{x{:}R\sigma \mid x{:}\sigma \in A\}$. We say $C', A' \vdash M{:}\sigma'$ is an *instance* of $C, A \vdash M : \sigma$ if there is some substitution R of types for type variables such that

$$RC \vdash C', \quad RA \subseteq A' \quad \text{and} \quad R\sigma = \sigma'$$

A typing $C, A \vdash M : \sigma$ is a *most general typing for* M, with respect to some signature, if it is derivable and has every other derivable typing statement for M is an instance.

Theorem 2.1 [Mit84, Mit91] *If M is typable with respect to some signature, then there is a most gen-*

eral typing statement for M, computable from M in exponential time.

Although the theorem given in [Mit84, Mit91] is only stated for pure lambda terms without constant symbols, the algorithm and proof are easily extended to constants with specified variable-free types. The algorithm may also be extended to terms with polymorphic constants, as described in [JM88, Jat89]. It is possible to decide whether a set of atomic subtype assertions is satisfiable in a partial order, in nondeterministic time polynomial in the size of assertion set and the presentation of the partial order. This gives us the following corollary.

Corollary 2.2 *There is a nondeterministic exponential time algorithm for deciding whether a typing $\emptyset, A \vdash M : \sigma$ is derivable with respect to a given signature.*

3 Type inference, constants and decision problems

While the algorithm given in [Mit84, Mit91] finds the most general type of any pure term, the application of this algorithm to a specific programming language is relatively subtle. If M does not contain constant symbols, then the most general typing for M will only contain type variables, and type constants do not enter into the problem. With both type and term constants, there are some questions regarding the set of subtype assumptions that might reasonably appear in a typing statement. A simple example that illustrates one of the problems with type constants is the signature with type constants *int* and *real,* with $int \preceq real$, and term constants $1 : int$, $2 : int$, $mult : int \rightarrow int \rightarrow int$ and $div : real \rightarrow real \rightarrow real$. In this signature, we can multiply integers 1 and 2 by writing $mult\,1\,2$ since both arguments have type integer, and divide by writing $div\,1\,2$ since by the assumption $int \preceq real$, both integers also have type *real*. However, consider the expression,

$$mult\,(div\,1\,2)\,2.$$

This is not well-typed, given the signature, since the subexpression $(div\,1\,2)$ only has type *real* and not type *int*. The typing algorithm in [Mit84, Mit91], when extended to constants in the simplest way, *would* produce a typing statement for this term, namely,

$$real \preceq int \;\vdash\; (mult\,(div\,1\,2)\,2) \;:\; int$$

Intuitively, this typing statements says that if *real* is a subtype of *int*, then the expression denotes an integer. This is a correct hypothetical statement, but since the hypothesis is false, it does not seem to be a useful output from the type checker. The reason that the algorithm infers a typing statement with additional subtype hypotheses is that, in general, this is the only way to obtain most general types. However, it is not reasonable to change the relationship between *int* and *real* by adding new subtypes of existing types. Therefore, as in [FM90, WO89], it makes sense to design a type checker that fails on the example expression above.

There are several reasonable restrictions on additional subtype assumptions. The first is to reject any term that requires subtype relations not given by the signature. Given a term M, we must find some typing statement $\emptyset, A \vdash M : \sigma$ with empty subtyping hypotheses. We call this the typing problem with *fixed subtype ordering* since the only subtype relations are those fixed by the signature. A second typing problem is to find a typing statement $C, A \vdash M : \sigma$ such that the only required relationships between type constants are those given by the signature. In other words, we require any inferred C to be *conservative* over the signature. We call this the typing problem with *varying subtype ordering,* since it is motivated by considering languages where the subtype ordering varies between different parts of the program. Conservativity rules out the typing for $mult\,(div\,1\,2)\,2$ above, since the signature does not imply $real \preceq int$. Both of these typing problems may be solved by computing the most general typing for a given term and then testing the set of subtyping assumptions to see if it can be made empty or conservative over the signature by applying a type substitution. For a particular programming language with subtyping, the appropriate typing problem may lie somewhere between these two extremes: additional type declarations will extend the subtype relation conservatively, but it may not be possible to obtain all conservative extensions.

The typing problems we consider in this paper are summarized in Table 1. We consider both fixed and varying subtype relations, as indicated along the top of the table. Restrictions on the signature are listed at the left. We consider arbitrary signatures, signatures in which all term constants have non-polymorphic types, and signatures in which the type of each term constant is a type constant. This gives us a two-dimensional matrix of typing problems. A third dimension is to consider possible restrictions on the subtype relation. With a variable subtype relation, the relation given by the signature has little effect. With a fixed subtype relation, we consider both arbitrary partial orders and lattices. For each of the problems, the table lists an upper bound on the upper line, and lower bound on the lower line, with trivial upper and lower bounds omitted. As the reader will readily see, we do not have matching upper and lower bounds for most of the problems listed. It is easy to show that each problem is reducible to the problem above it in

Signature	Fixed Subtype Relation				Varying Subtype Relation	
	Arbitrary		Lattice			
Constants of any type	**F1**	NEXP upper bound	**L1**	PSPACE upper bound	**V1**	PSPACE upper bound
		conservative over F2		conservative over L2		conservative over V2
No polymorphic types	**F2**	reducible to F1	**L2**	reducible to L1	**V2**	reducible to V1
		conservative over F3		conservative over L3		conservative over V3
Atomic types only	**F3**	reducible to F1	**L3**	linear time	**V3**	linear time
		NP lower bound		linear time		linear time

Table 1: Summary of problems and results. Lower bounds for F1–3 and upper bounds for L1–3 and V1–3.

the table, and conservative over the problem below it. This is because all are defined using the same proof rules. The linear upper bound for problem L3 actually holds for any order that is the disjoint union of any number of partial orders with maximum elements.

We state problems F1–3 and V1–3 in full below. Problems L1–3 are identical to F1–3, respectively, except that the subtype order must be a lattice.

F1: Given an untyped lambda term M, possibly containing constant symbols from some signature Σ, determine whether there exists a provable typing statement $\emptyset, A \vdash M : \sigma$ without additional subtyping assumptions.

F2: Given an untyped lambda term M, possibly containing constant symbols from some signature Σ, where all constants have variable-free type, determine whether there exists a provable typing statement $\emptyset, A \vdash M : \sigma$ without additional subtyping assumptions.

F3: Given an untyped lambda term M, possibly containing constant symbols from some signature Σ, where all constants have atomic type, determine whether there exists a provable typing statement $\emptyset, A \vdash M : \sigma$ without additional subtyping assumptions.

V1: Given an untyped lambda term M, possibly containing constant symbols from some signature $\Sigma = \langle B, S, T \rangle$, determine whether there exists a provable typing statement $C, A \vdash M : \sigma$ such that for all type constants $b_1, b_2 \in B$, we have $C \cup S \vdash b_1 \preceq b_2$ iff $S \vdash b_1 \preceq b_2$.

V2: Given an untyped lambda term M, possibly containing constant symbols from some signature $\Sigma = \langle B, S, T \rangle$, where all constants have variable-free type, determine whether there exists a provable typing statement $C, A \vdash M : \sigma$ such that for all type constants $b_1, b_2 \in B$, we have $C \cup S \vdash b_1 \preceq b_2$ iff $S \vdash b_1 \preceq b_2$.

V3: Given an untyped lambda term M, possibly containing constant symbols from some signature $\Sigma = \langle B, S, T \rangle$, where all constants have atomic type, determine whether there exists a provable typing statement $C, A \vdash M : \sigma$ such that for all type constants $b_1, b_2 \in B$, we have $C \cup S \vdash b_1 \preceq b_2$ iff $S \vdash b_1 \preceq b_2$.

An example may help clarify the difference between problems F1 and V1. The term $(\lambda v.((\lambda x.(v\, c_1))(v\, c_2)))$ is typable in the signature with two constants $c_1 : \theta_a$, $c_2 : \theta_b$ and empty subtyping relation, according to the constraints of V1 but not F1. The reason is that the variable v must have type $\sigma \to \tau$ for some σ greater than θ_a and θ_b.

A variation we will not consider is to give more information about a term to be typed. For example, we could give term M and type σ, and ask whether there is a provable typing statement $C, A \vdash M : \sigma$. This might appear easier than the type decision problem, since the added information could narrow the range of possibilities to consider. However, it is easy to see that an arbitrary term M has a type iff the term $\lambda x.\, KxM$ has type $\tau \to \tau$, where K is the lambda term $\lambda x.\, \lambda y.\, x$. Therefore, it does not help to supply a type.

4 Subtype Inference is NP-Hard

Wand and O'Keefe give an argument for the NP-hardness of type inference which requires the use of a constant of polymorphic type, specifically, a constant T with polymorphic type $\forall\alpha.(\alpha \to \alpha \to \alpha)$ [WO89], roughly corresponding to our problem F1. In this section we improve their lower bound by proving that the strictly weaker problem F3 is NP-hard. Since F2 and F1 are conservative over F3, this lower bound also applies to these problems.

We will reduce POL-SAT, stated as follows, to F3. Given a partial order $(P, \leq)$ and a set of inequalities I of the form $p \leq w$, $w \leq w'$, where w and w' are variables, and p is a constant drawn from P, is there is an assignment from variables to members of P that satisfies all the inequalities I? This problem is very similar to PO-SAT, proven NP-complete by Wand and

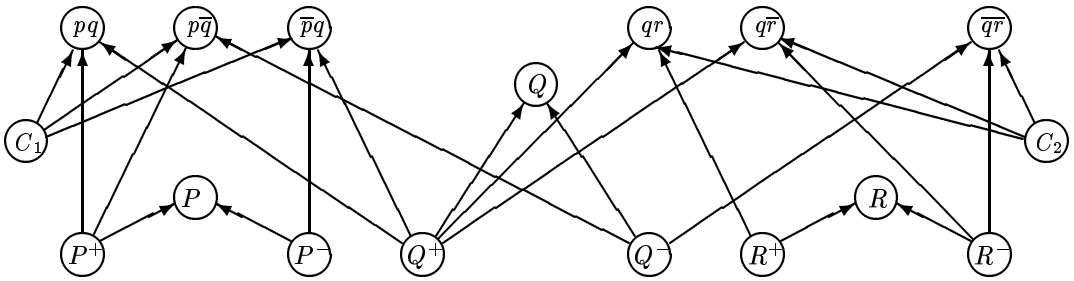

Figure 1: poset for $(P \vee Q) \wedge (Q \vee \neg R)$

O'Keefe [WO89]. PO-SAT differs from POL-SAT in that it allows inequalities of the form $w \leq p$, which amount to upper bounds. PO-SAT may also be described as the satisfiability problem for inequations over a poset. Similarly, POL-SAT is also the satisfiability problem for inequations over a poset with the added restriction that no inequations have the form $w \leq p$ for variable w and constant p. We first show that POL-SAT is NP-complete, and then show that POL-SAT reduces to F3.

Lemma 4.1 *POL-SAT is* NP*-complete.*

Proof. It is easy to see that this problem is in NP, since one may simply guess an assignment of constants to variables, and check that every inequality in I is satisfied.

To show that this problem is NP-hard, we give a reduction from 3-SAT. We begin with the empty set A, and for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$, we add the element named C_i to A, and further add 7 more elements to A, one for each truth assignment which satisfies the clause. For convenience, we name these 7 elements by simply concatenating the names of the clauses with the names of the variables they contain, using overbars to denote negation: "$C_iP_{i1}P_{i2}P_{i3}$", "$C_iP_{i1}P_{i2}\overline{P_{i3}}$", "$C_iP_{i1}\overline{P_{i2}}P_{i3}$", etc. For each propositional variable P_j, we add three elements to A, named "P_j", "P_j^+", and "P_j^-". Intuitively, these stand for the j-th proposition being undecided, true, and false, respectively.

With the above set of constants, we define a partial order relation $\leq$ on them as follows. We define the relation R_{prop} to include, for each proposition P_i, $P_i^+ \leq P_i$ and $P_i^- \leq P_i$. We define the relation R_{clause} to include, for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$ occurring in the 3-SAT problem, and each truth assignment which satisfies the clause, $C_i \leq C_iP_{i1}P_{i2}P_{i3}$. We also define the relation R_{true} to include, for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$, and each proposition in that clause P_{ij}, a relation $P_{ij}^+ \leq C_iP_{i1}P_{i2}P_{i3}$ for each of the 3 or 4 clause elements which correspond to P_{ij} being true. Similarly, we define the relation R_{false} to include, for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$, and each proposition in that clause P_{ij}, a relation $P_{ij}^- \leq C_i\overline{P_{i1}P_{i2}P_{i3}}$ for each of the 3 or 4 clause types which correspond to P_{ij} being false. The final partial order of interest will be $(A, R_{prop} \cup R_{clause} \cup R_{true} \cup R_{false})$. The partial order has height one, and contains 8 ($= 2^3$) elements for each 3-SAT clause, plus three elements for each proposition. Figure 1 displays the partial order produced for the SAT problem $(P \vee Q) \wedge (Q \vee \neg R)$. Clauses of length two were used, and the name pq was used in place of C_1pq, for example, in Figure 1 to improve readability.

We use a set of variables, one wp_j and one wu_j for each proposition P_j, and one wc_j for each clause $Clause_j$. We define a set of inequations I_{clause} to include, for each clause $Clause_i = P_{i1} \vee P_{i2} \vee P_{i3}$, the inequality $C_i \preceq wc_i$, and for each proposition P_{ij} in that clause, $wp_{ij} \preceq wc_i$. We also define a set of inequations I_{prop} to include, for each proposition P_i, $wp_i \preceq wu_i$ and $P_i \preceq wu_i$. Thus there are four inequalities in I_{clause} per 3-SAT clause, and two inequalities in I_{prop} for each proposition. Continuing with our simple example, $(P \vee Q) \wedge (Q \vee \neg R)$, the inequations $I_{clause} = \{C_1 \preceq wc_1, wp_p \preceq wc_1, wp_q \preceq wc_1, C_2 \preceq wc_2, wp_q \preceq wc_2, wp_r \preceq wc_2\}$, and $I_{prop} = \{wp_p \preceq wu_p, wp_q \preceq wu_q, wp_r \preceq wu_r, P \preceq wu_p, Q \preceq wu_q, R \preceq wu_r\}$.

We claim that the POL-SAT problem given by the partial order $(A, R_{prop} \cup R_{clause} \cup R_{true} \cup R_{false})$, with the inequalities $I_{prop} \cup I_{clause}$ has a solution if and only if the original 3-SAT problem has one. This may be observed by noting that every wc_i must be assigned some $C_iP_{i1}P_{i2}P_{i3}$, since wc_i must be greater than C_i and some propositions. Also, the only $C_iP_{i1}P_{i2}P_{i3}$ which exist in A correspond to assignments of propositions which satisfy the clause. Further, wu_j must be assigned P_j, and wp_j must be assigned either P_j^+ or P_j^-. We claim there is a correspondence between a proposition P_j being assigned true (or false, resp.) in the 3-SAT problem, and w_j being assigned P_j^+ (P_j^-, resp.) in the POL-SAT problem. Thus one may see that a solution to the 3-SAT may be derived from any solution to the constructed

POL-SAT problem and vice-versa. ∎

This construction may be simplified somewhat, by omitting the inequalities I_{prop}, and the elements P_j (nodes labeled P, Q, and R in the example poset). In this case the correctness of the reduction is more difficult to establish. However, in either case the constructed poset has depth one, and both the poset and the set of inequalities have size linear in the input 3-SAT problem.

Lemma 4.2 POL-SAT *reduces to F3.*

Proof. A POL-SAT problem is given with partial order $(P, \leq)$, set of variables W, and set of inequalities I. We define the set of type constants $B = \{\theta_i | p_i \in P\}$, and a set of constants and their typings $T = \{c_i : \theta_i | p_i \in P\}$. That is, for each element of the POL-SAT partial order, we define a type θ_i, and a constant of that type c_i.

We define S to be a set of atomic coercions such that for each $p_i \leq p_j$ in the POL-SAT partial order, $\theta_i \preceq \theta_j$ is in S. That is, we simply copy the partial order from the POL-SAT problem into a set of subtype assertions about corresponding type constants.

We then collect the above together into a signature $\Sigma = \langle B, S, T \rangle$.

For each variable w_i appearing in any inequality in the POL-SAT problem, we define the notation for two lambda term variables v_i and u_i. We number the m inequalities I in the POL-SAT problem $i_1, \cdots, i_m$, and define the translation $[i_j]$ of inequalities as follows:

$$\begin{array}{lcl} [p_y \leq w_x] & = & (v_x\, c_y) \\ [w_x \leq w_y] & = & (v_y\, (v_x\, u_x)) \end{array}$$

Finally, we build the term

$$\begin{array}{l} (\lambda u_1. \cdots (\lambda u_n.(\cdots ((\lambda v_1. \cdots (\lambda v_n. \\ ((\lambda x.[i_1])((\lambda x.[i_2]) \cdots ((\lambda x.[i_{m-1}])[i_m]) \cdots)) \\ \qquad\qquad (\lambda x.x))(\lambda x.x)) \cdots (\lambda x.x)) \cdots))) \end{array}$$

In words, we encode each lower bound on a variable as an application of that variable to the corresponding constant, and encode each relation between variables v_1 and v_2 as an application of v_1 to the result of applying v_2 to u_2. The variable u_2 only serves as a dummy variable to which one can apply v_2. If the partial order has a bottom element, one could replace all uses of u variables with a single constant with the type of the bottom element of partial order. We build an abstraction over the set of function variables v_i, with a body that includes a subterm for each inequality, but throws all the results away except the first. We tie the upper and lower bounds on each function variable together by applying each abstraction to the identity function $(\lambda x.x)$, and finally abstract over the u variables.

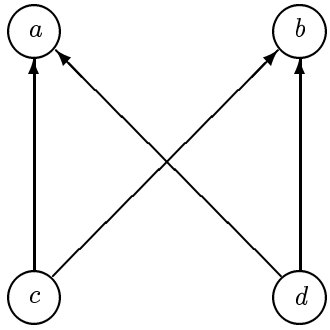

Figure 2: Fixed NP-hard poset

We claim without proof that this term is typable if and only if the corresponding POL-SAT problem has a solution. ∎

Thus we have shown that F3 suffices to capture the essential NP-hardness of type checking with subtypes. As stated above, Wand and O'Keefe show that PO-SAT reduces to F1 [WO89]. However, there also exists a straightforward extension of the above into a reduction from PO-SAT to F2: define the constant c_i^- to be of type $\theta_i \rightarrow \theta_i$, and $[w_x \leq p_y] = (c_y^-\, (v_x\, u_x))$. Of course, there are reductions from F3 to F2, and F2 to F1, since the problems strictly subsume each other.

Theorem 4.3 *F3, F2, and F1 are* NP*-hard.*

Proof. F3 is NP-hard from the above two lemmas, and F2 and F1 are conservative over F3, so the result follows immediately. ∎

Recent work by Pratt and Tiuryn [PT91] has shown that PO-SAT remains NP-complete for certain fixed posets. Our construction builds a different poset and set of inequations for each 3-SAT problem. Pratt's construction builds a different set of inequalities over a fixed poset, although it uses inequations of the form $w \leq p$. Thus Pratt's result subsumes the NP-hardness of PO-SAT. With a simple modification, Pratt's NP-hardness result can be extended to cover the case of restricted inequalities, which corresponds to POL-SAT.

Pratt shows that PO-SAT is NP-complete even over the fixed poset containing only four elements, drawn in Figure 2. The following reduction from PO-SAT to POL-SAT, although not sound in general, is correct for this particular poset. Thus POL-SAT is also NP-hard for this poset. Given a set of inequations, we must translate them into a form where no upper bounds $w \leq p$ appear for variable w and constant p. We add two new variables, w_a and w_b, and the new inequalities $\{a \leq w_a, b \leq w_b\}$. We then translate all upper bounds (which are disallowed in POL-SAT) as:

$$\begin{array}{l} [w \leq a] = \{w \leq w_a\} \\ [w \leq b] = \{w \leq w_b\} \end{array}$$

If c is used as an upper bound on some variable w, then simply replace w by c in the entire set of inequa-

tions, and similarly for d. Thus even for fixed posets with as few as four elements POL-SAT is NP-complete. Through the reduction stated formally in Lemma 4.2, we therefore have the result that F3 is NP-hard even for fixed posets.

Theorem 4.4 *F3, F2, and F1 are* NP*-hard for a fixed posets with four elements.*

Note that the above NP-hardness results for POL-SAT and PO-SAT make critical use of non-lattice partial orders. In fact, we have the following properties:

Proposition 4.5 PO-SAT *is solvable in polynomial time over a lattice.*

Proposition 4.6 POL-SAT *is solvable in polynomial time over a lattice.*

These results lead to a polynomial algorithm for L3, as stated later in Proposition 6.2. At an intuitive level problem V3 allows one to complete the given partial order into a lattice, leading to a similar polynomial time algorithm for V3 as well. Thus the NP-hardness results of this section apply only to the problems F1-3, and do not directly apply to L1-3 nor to V1-3.

5 Subtype Inference in PSPACE

In this section we investigate the computational complexity of problems V1 and L1. We give a PSPACE algorithm for V1 and then show that the same algorithm also solves problem L1.

The algorithms proposed in earlier papers to solve V1 (or F1) suffer from two sources of inefficiency. The first source of inefficiency is the non-lattice structure of subtype orders in the signature. These lend a certain NP flavor to the decision problem. The second source of inefficiency is the MATCH (and SIMPLIFY) algorithm, which forces subtype relationships between complex types into sets of subtype relationships between atomic types. The expansion of type inequalities "to the leaves" causes an exponential blowup in the inequalities, and thus causes previous algorithms to use exponential space and time. To overcome this obstacle, we develop a data structure of linear size and associated naming convention for new type variables which allow us to represent the required subtype relationships succinctly. Using this approach, we may decide typability in PSPACE.

Rather than present a deterministic PSPACE algorithm directly, we give a nondeterministic PSPACE algorithm that recognizes untypable terms. Since NPSPACE = PSPACE and PSPACE is closed under complement, this gives us a PSPACE upper bound. Our algorithm begins by building a proof up to uses of **Sub**. As discussed earlier this amounts to a normal form for the type derivation proof, except that the proofs above **Sub** are left incomplete. Next the DAG representation of the Curry-type of the term is computed, as if the type of each constant and variable were renamed with new type variables at each leaf occurrence. In [Wan87] an algorithm similar to ours up to this point is presented. However, in our algorithm, the **Sub** rule presents a new kind of relation, and we actually solve the equations generated by the algorithm in [Wan87] with unification, producing a DAG which represents the types of all subterms. Note that the unifications performed at this step never fail, due to type renaming, as is the case in [Hin89]. At each leaf a constraint $\sigma \preceq \tau$ is generated by the **Sub** rule, which we encode as a "dashed" arc on the DAG. Note that these inequalities (represented by dashed arcs) may involve terms such as $\alpha \to \beta$ containing function types. We will call the arcs forming the original DAG *descendent arcs* and dashed arcs due to uses of the **Sub** rule *sub arcs.*

We say a term M is *Curry-typable* over signature $\Sigma = \langle B, S, T\rangle$, if M is typable over the signature $\Sigma' = \langle B, S', T\rangle$, where S' is the complete relation (all atomic types are related, and thus all atomic types are interchangeable).

Lemma 5.1 *Given a term M, possibly containing constant symbols from some signature $\Sigma = \langle B, S, T\rangle$, then M is Curry-typable over Σ if and only if there is a provable typing statement $C, A \vdash M : \sigma$ over Σ where C may have any relationship to S.*

Lemma 5.2 *Given a Curry-typable untyped lambda term M, possibly containing constant symbols from some signature $\Sigma = \langle B, S, T\rangle$, then V1 is solvable for M if and only if the most general typing statement $C, A \vdash M : \sigma$ for M provable with respect to Σ is such that $\forall\, b_1, b_2$. if $C \cup S \vdash b_1 \preceq b_2$ then $S \vdash b_1 \preceq b_2$.*

Thus there are two kinds of type failure for V1. The first is failure of Curry-typability, which occurs when a type variable is required to match its own ancestor or descendant because of coercions. For example, terms with self application, such as $\lambda x.(x\, x)$, are impossible to type. The second type of failure, implication of nonexistent coercion, occurs if there is some chain or sequence of implied coercions $\theta_i \preceq \sigma_1$, $\sigma_1 \preceq \sigma_2, \cdots, \sigma_{n-1} \preceq \sigma_n, \sigma_n \preceq \theta_j$ such that it is *not* the case that $S \vdash \theta_i \preceq \theta_j$.

The first type of failure is relatively easy to detect, and may be checked in linear time. If one considers the sub arcs of the DAG to be undirected, the first type of failure occurs if and only if the DAG contains a cycle which contains at least one descendent arc. This condition may be checked in linear time by considering the the sub arcs to be equations between parts of the DAG made up of descendent arcs. The DAG and the

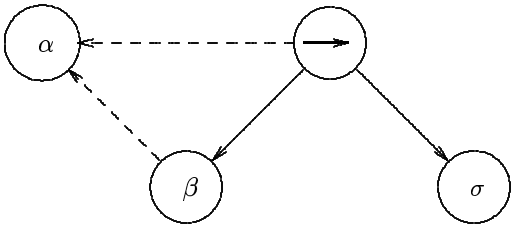

Figure 4: Untypable Self Application's DAG

resulting unification problem are of linear size, and unification may be performed in linear time [PW78].

For example, consider the attempted typing of $\lambda x.(x\,x)$ shown in Figure 3. This is the unique syntax-directed proof, up to Π and Δ, which are left incomplete by the algorithm. However, the coercions $\alpha \preceq \beta \to \sigma$ and $\alpha \preceq \beta$ have a derived inconsistency. That is, β must match $\beta \to \sigma$, and thus no substitution of types for type variables can satisfy those inequations. Figure 4 displays the two color DAG our algorithm builds for this term. The DAG which represents the type of all subterms is represented with descendent arcs shown as solid arcs in Figure 4. The dashed arcs in that figure represent sub arcs.

Assuming that the first type of failure does not occur, we must detect the second. One could imagine converting all subtype relations between non-atomic types into relations between atomic types, and then searching for a solution to that easier problem. However, an exponential number of atomic subtype relations may be generated by such a procedure. The algorithm presented below avoids this blowup.

For each type variable α, we use the notation α_l and α_r, where their relationship with α is defined by $\alpha = \alpha_l \to \alpha_r$. This is simply notation; we do not explicitly construct all such type variables, and the notation is meaningless if α is of atomic type. We define a *path* to be a string on the alphabet $\{l, r\}$, and we use $\circ$ as path concatenation. We define the *relation implied by a sub arc from α to σ through path p* as follows. If p is empty, then the sub arc simply signifies that $\alpha \preceq \sigma$. If $p = r \circ p'$, then the sub arc implies the same relation as the sub arc from α_r to σ_r through path p'. If $p = l \circ p'$, then the arc implies the same relation as the arc from σ_l to α_l through path p'. Note that because of the antimonotonic **Arrow** rule, the sub arc in the l case has changed direction.

The algorithm begins by guessing two atomic type constants θ_1 and θ_2 which are *not* in the relation $\theta_1 \preceq \theta_2$ in the given signature. Then the algorithm guesses two types σ and α and a path p such that there is a dashed arc from σ to α and the relation implied by this arc from σ to α through p is $\theta_1 \preceq \tau_1$. Or the algorithm guesses that τ_1 is a type constant θ_i such that $S \vdash \theta_1 \preceq \theta_i$.

The algorithm then repeatedly guesses types and paths in this way such that for guess i, the relation $\tau_{i-1} \preceq \tau_i$ is implied. Finally, the algorithm guesses types such that $\tau_n \preceq \theta_2$ is implied. If this algorithm succeeds in all these steps, the term is not typable. That is, the algorithm as described nondeterministically checks "un-typability" in PSPACE.

As an example of the second type of failure, consider the attempted derivation of a type for the term $((\lambda v.(odd?\ (v\ 5.7)))\ (\lambda x.x))$ in the signature with three types constants, *int*, *real*, and *bool*, with the only subtype assumption enforcing that $int \preceq real$, and two constants, $odd? : int \to bool$ and $5.7 : real$. We give the derivation in parts, leaving the proofs numbered $1 \cdots 4$ incomplete, just as our algorithm would do, in Figures 5, 6, and 7.

We may now see that this term has no type satisfying the restrictions of V1. From the required subtype relation marked 1 in the above proof display, $int \to bool \preceq \alpha \to \beta$. From this, by the arrow rule, we must have $bool \preceq \beta$ and $\alpha \preceq int$. Similarly, from the required subtype relation marked 2 in the above proof display, $\sigma \to \gamma \preceq \tau \to \alpha$, which implies $\gamma \preceq \alpha$ and $\tau \preceq \sigma$. Using these subtype relations, and those from 3 and 4, $real \preceq \tau$ and $\sigma \preceq \gamma$, we may build the following chain of subtype relations:

$$real \preceq \tau \preceq \sigma \preceq \gamma \preceq \alpha \preceq int$$

We now describe how our algorithm would discover this inconsistency with the given partial order. First, it would build the proof up to the applications of **Sub**, creating the DAG as described above, a fragment of which is represented in Figure 8. The algorithm then guesses that a derived inconsistency lies in the subtype relation $real \preceq int$. It then guesses the sub arc from τ to *real*, and the empty path. This implies $real \preceq \tau$. It then guesses the sub arc from $\tau \to \alpha$ to $\sigma \to \gamma$, and the path l. This implies $\tau \preceq \sigma$. The next guess is the sub arc from γ to σ, with the empty path, implying $\sigma \preceq \gamma$. Then the sub arc from $\tau \to \alpha$ to $\sigma \to \gamma$ is guessed again, this time with path r, implying $\gamma \preceq \alpha$. The last guess is the sub arc from $\alpha \to \beta$ to $int \to bool$, with path r, implying $\alpha \preceq int$. The last step is through a part of the DAG not represented in Figure 8. Thus the algorithm finds a chain of types which together imply $real \preceq int$, contradicting the given signature.

Theorem 5.3 *Problem V1 is solvable in* PSPACE.

Proof. The "untypability" algorithm described above operates in polynomial space, since the preprocessing phases building the DAG may be completed in linear time and space, and the nondeterministic sequence of choices can be made with only linearly bounded storage space, since the depth of the curry-type is linearly bounded. Because PSPACE is closed

$$\dfrac{\dfrac{\dfrac{}{C,\{x:\alpha\}\vdash x:\alpha}\,\textbf{Var}\quad\dfrac{\vdots}{C\vdash\alpha\preceq\beta\to\sigma}}{C,\{x:\alpha\}\vdash x:\beta\to\sigma}\,\textbf{Sub}\qquad\dfrac{\dfrac{}{C,\{x:\alpha\}\vdash x:\alpha}\,\textbf{Var}\quad\dfrac{\vdots}{C\vdash\alpha\preceq\beta}}{C,\{x:\alpha\}\vdash x:\beta}\,\textbf{Sub}}{\dfrac{C,\{x:\alpha\}\vdash(x\,x):\sigma}{C,\emptyset\vdash\lambda x.(x\,x):\alpha\to\sigma}\,\textbf{Abs}}\,\textbf{App}$$

Figure 3: Untypable Term's Attempted Derivation

$$\dfrac{\dfrac{}{C,\{v:\sigma\to\gamma\}\vdash odd?:int\to bool}\,\textbf{Const}\qquad\overset{1}{C\cup S\vdash int\to bool\preceq\alpha\to\beta}}{C,\{v:\sigma\to\gamma\}\vdash odd?:\alpha\to\beta}\,\textbf{Sub}$$

Figure 5: Partial derivation Δ

$$\dfrac{\dfrac{\dfrac{}{C,\{v:\sigma\to\gamma\}\vdash v:\sigma\to\gamma}\,\textbf{Var}\qquad\overset{2}{C\cup S\vdash\sigma\to\gamma\preceq\tau\to\alpha}}{C,\{v:\sigma\to\gamma\}\vdash v:\tau\to\alpha}\,\textbf{Sub}\qquad\dfrac{\dfrac{}{C,\{v:\sigma\to\gamma\}\vdash 5.7:real}\,\textbf{Const}\qquad\overset{3}{C\cup S\vdash real\preceq\tau}}{C,\{v:\sigma\to\gamma\}\vdash 5.7:\tau}\,\textbf{Sub}}{C,\{v:\sigma\to\gamma\}\vdash(v\ 5.7):\alpha}\,\textbf{App}$$

Figure 6: Partial derivation Π

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta}{\vdots}\qquad\dfrac{\Pi}{\vdots}}{C,\{v:\sigma\to\gamma\}\vdash odd?:\alpha\to\beta\qquad C,\{v:\sigma\to\gamma\}\vdash(v\ 5.7):\alpha}\,\textbf{App}}{C,\{v:\sigma\to\gamma\}\vdash(odd?\ (v\ 5.7)):\beta}\,\textbf{Abs}}{C,\emptyset\vdash(\lambda v.(odd?\ (v\ 5.7))):(\sigma\to\gamma)\to\beta}\qquad\dfrac{\dfrac{\dfrac{}{C,\{x:\sigma\}\vdash x:\sigma}\,\textbf{Var}\qquad\overset{4}{C\cup S\vdash\sigma\preceq\gamma}}{C,\{x:\sigma\}\vdash x:\gamma}\,\textbf{Sub}}{C,\emptyset\vdash\lambda x.x:\sigma\to\gamma}\,\textbf{Abs}$$
$$\textbf{App}$$
$$C,\emptyset\vdash((\lambda v.(odd?\ (v\ 5.7)))\ (\lambda x.x)):\beta$$

Figure 7: Whole derivation

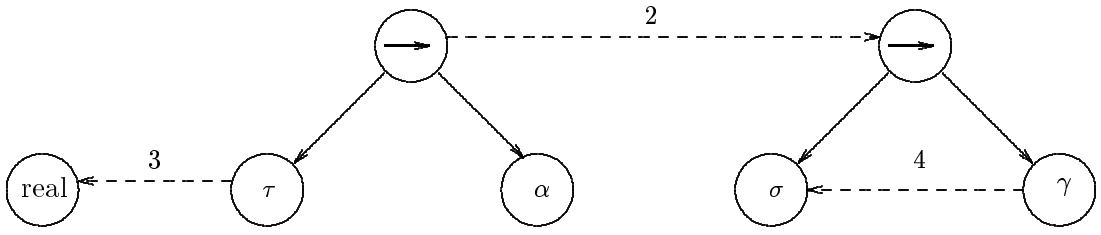

Figure 8: Fragment of Untypable Term's DAG

under complement, this demonstrates the existence of a PSPACE algorithm. ∎

We now turn our attention to problems L1-3.

Lemma 5.4 *L1 and F1 are solvable for term M over signature Σ if and only if M is Curry typable and $C, A \vdash M : \sigma$, the most general typing statement for M, is such that C is satisfiable over Σ.*

Proof. Immediate from the problem definitions and Theorem 2.1. The key here is that if C is satisfiable over Σ, then there is some substitution R and provable instance $\emptyset, RA \vdash M : R\sigma$. ∎

The following lemma states that problem L1 is essentially the same as problem V1 if the partial order is already a lattice.

Lemma 5.5 *If $\langle B, S\rangle$ forms a lattice and C is a set of atomic subtyping assertions, possibly involving constants from B, then C is satisfiable over $\langle B, S\rangle$ if and only if for every pair of constants b_1, b_2 from B, if $C \cup S \vdash b_1 \leq b_2$ then $S \vdash b_1 \leq b_2$.*

Proof. Suppose that for every pair of constants b_1, b_2 from B, if $C \cup S \vdash b_1 \leq b_2$ then $S \vdash b_1 \leq b_2$. We show C is satisfiable by giving a satisfying assignment. For each variable x in C, let $LB(x)$ be the set of elements b of B such that $C \cup S \vdash b \leq x$. We assign variable x the least upper bound of the set $LB(x)$. This upper bound exists since $\langle B, S\rangle$ is a lattice. To show that this assignment satisfies C, we consider three cases:

- For $b \leq x$, it is the case that $b \in LB(x)$, so x is assigned an element of B that is greater than or equal to b.
- For $x \leq y$, we have by transitivity that $LB(x) \subset LB(y)$, so y is assigned an upper bound of $LB(x)$, so x is assigned some element of B that is less than or equal to y.
- For $x \leq b$, we have to use the hypothesis of the lemma. By hypothesis, $\forall b' . b' \in LB(x)$, then $C \cup S \vdash b' \leq b$ and so $b' \leq b$. Therefore the least upper bound of $LB(x)$ is less than or equal to b.

∎

Theorem 5.6 *Problems L1, L2, and L3 are solvable in* PSPACE.

Proof. We begin by observing that the only properties specific to problem V1 that are used in the proof of Theorem 5.3 are stated in Lemmas 5.1 and 5.2. Since Lemmas 5.4 and 5.5 characterize problem L1 in exactly the same way, the proof of Theorem 5.3 also shows that L1 also may be solved in PSPACE. By the obvious conservativity, we have the result for L2 and L3. ∎

6 L3,F3 are Polynomial

In this section we present a linear time algorithm for L3, the restricted case of problem F3 where the given subtype order is a lattice. This algorithm also extends to F3 where the order is the sum of partial orders, each with its own top element. That is, partial orders with the property that there is a unique least upper bound of the upper bounds of any element. Also, this algorithm extends to V3 over any partial order. Since problem F3 is NP-hard in general, we see that minor assumptions about the subtype order permit great reductions in the computational complexity of the associated decision problems. Also, small amounts of flexibility in the subtype order (V3) permit similar reductions in the computational complexity.

In [WO89], it is claimed that subtype inference may be performed in low order polynomial time if the given subtype order in the signature happens to be a tree. However, we have found examples where the algorithm suggested in [WO89] uses exponential space and time.

Lemma 6.1 *Let Σ be a signature in which all constants have atomic type. If M is an untyped term over Σ, then in the most general typing for M, all subtype assumptions have the form $p \leq w$, $w \leq w'$, where w and w' are variables, and p is a type constant from Σ.*

Proposition 6.2 *L3 is solvable in linear time.*

Proof. Again, we solve the decision problem without producing a most general typing. By Lemma 6.1, the only relevant subtyping constraints are lower bounds on the types of terms. Thus one could choose to build all types of subterms from the topmost type constant. Since all types are subsumed by the topmost type constant this choice will not lead to any type errors which are not inevitable. One may view this as collapsing the entire poset down to the single topmost point.

Thus our algorithm can be described as follows: replace all constants in the given term by a single fixed constant of topmost type and then apply the well-known linear algorithm for determining Curry-typability. If the resulting term is Curry-typable, then the given term is typable with subtypes. If the modified term is not Curry-typable, then the original term is not typable. ∎

We now consider a somewhat more general problem. A *connected component* of signature is a subset of the elements of the signature which is connected if the subtype relation is taken to be bidirectional.

Proposition 6.3 *F3 is solvable in linear time if every connected component of the signature has a topmost element.*

Proof. Similar to 6.2. In this case, replace each constant of type τ with a constant of the topmost type connected to τ. One may view this as collapsing all connected components into their individual topmost elements. The result is a completely flat partial order, over which Curry-typability works in linear time with small modification. ∎

Special cases of this class of "easy" signatures include flat partial orders, lattices, trees, forests, etc.

Proposition 6.4 *V3 is solvable in linear time.*

Intuitively, V3 allows new elements to be added to the signature. Thus we may simply add a top element to the signature, and then check typability as above in Proposition 6.2. Thus F3 is NP-hard over certain partial orders, but V3 is solvable in linear time over any partial order.

7 Conclusion

We identify and study several variations on the type inference problem for languages with subtypes. We give a single NP lower bound for three of these problems, improving the previous lower bound of [WO89]. Since the size of the most general typing of a term may be exponentially larger than the given term, any algorithm which prints the most general typing with subtypes must take exponential time. However, we show that it is possible to determine whether a term is typable at all using only PSPACE for V1–3 and L1–3. We do not know whether this algorithm can be improved to run in NP, and have no useful lower bound on V1–2 or L1–2.

The most promising indication for practical applications is that typing over special partial orders (such as lattices) and varying subtype relations (as would arise in languages with subtype declarations) may be far simpler than typing over arbitrary partial orders. We have seen this in the difference between problem F3 and L3 and V3: while L3, over arbitrary partial orders, is NP-hard, the restriction, L3, to lattices may be solved in linear time and so may the corresponding problem, V3, for languages with varying subtype relations. These results show that the complexity of type inference is sensitive to the kind of subtype relation that may occur in a given programming language, and whether this order may vary.

In designing type inference algorithms for languages with type declarations (and therefore varying subtype relations), we believe it will be useful to take into account the ways that the subtype relation may change. To give an concrete example, suppose that in language L subtype declarations may only add new subtypes, not supertypes of existing types. Then in defining a type checker for language L, we would like to reject any declaration that will only make sense when supertypes of existing types are added. In general, we expect to find typing problems that are special cases of both our fixed and varying subtype problems, with only certain kinds of subtype relations definable by programs, and only certain kinds of variations achievable by additional type declarations.

References

[FM90] Y. Fuh and P. Mishra. Type inference with subtypes. *Theor. Computer Science*, 73, 1990.

[Hin89] J.R. Hindley. BCK-combinators and linear λ-terms have types. *Theor. Comp. Sci.*, 64:97–105, 1989.

[Jat89] L. Jategaonkar. ML with extended pattern matching and subtypes. Master's thesis, MIT, 1989.

[JM88] L. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 198–212, July 1988.

[KMM91] P.C. Kanellakis, H.G. Mairson, and J.C. Mitchell. Unification and ML type reconstruction. In *Computational Logic, essays in honor of Alan Robinson*, page to appear. MIT Press, 1991.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[Mit84] J.C. Mitchell. Coercion and type inference (summary). In *Proc. 11th ACM Symp. on Principles of Programming Languages*, pages 175–185, January 1984.

[Mit91] J.C. Mitchell. Type inference with simple subtypes. *J. Functional Programming*, 1(3):245–286, 1991.

[PT91] V. Pratt and J. Tiuryn. Satisfiability of inequations in a poset. Manuscript, October 1991.

[PW78] M.S. Paterson and M.N. Wegman. Linear unification. *JCSS*, 16:158–167, 1978.

[Str86] B. Stroustrop. *The C++ Programming Language.* Addison-Wesley, 1986.

[Tiu91] J. Tiuryn. Solving term inequalities is PSPACE-hard. Manuscript, October 1991.

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[Wan91] M. Wand. Personal communication, 1991.

[WO89] M. Wand and P. O'Keefe. On the complexity of type inference with coercion. In *Proc. ACM Conf. Functional Programming and Computer Architecture*, pages 293–298, 1989.