Subtyping, consistency and derivability

Sven-Olof Nyström

May 5, 2014

Abstract

Earlier work on subtyping has focused on the problem of constructing a typing for a given program. This paper considers a slightly different problem: Given a lambda term, is the corresponding constraint system *consistent*? An $O(n^3)$ algorithm for checking the consistency of constraint systems is presented, where n is the size of the constraint system.

The paper also considers the problem of *derivability*, i.e., whether a property can be derived from the corresponding constraint system and gives an $O(n^3)$ algorithm for checking derivability of a class of constraints.

1 Introduction

We consider subtyping systems for functional programming languages.

Subtyping systems are a generalisation of traditional Hindley-Milner type systems [8, 12] in which a type may include another type.

Mitchell suggests the use of subtyping systems to allow coercions between types, the coercions might be user-defined or given in the programming language [13]. For example, if the type *int* is a subtype of the type *real*, i.e., $int \leqslant real$, any function that expects a real will also accept an integer as argument. Thatte described the use of subtyping systems to allow a universal type. A universal type corresponds to a variable or expression that may take on any value [19]. If a function expects an argument of the universal type, it will accept a value of any type.

As discussed below, for any lambda term it is straight-forward to construct a constraint system (i.e., a logic theory) that is consistent if and only if the lambda term type checks. Barendregt et al. [3] define the *type checking problem* as that of determining whether a lambda term has a given type, and the *typability problem* as that of determining whether a lambda term has

any type at all. Both problems are solved if we can determine consistency of arbitrary constraint systems. The problem of checking whether a variable or an expression in a program always has a certain type can be expressed as the problem of determining whether a certain constraint can be derived from the constraint system.

- We give an algorithm with $O(n^3)$ worst-case complexity for checking the consistency of constraint systems, where n is the size of the constraint system.
- Assume that t is a type expression of size m that does not contain any type variables and X is a type variable. We give an algorithm with $O(n^3 + n^2 m)$ worst-case complexity for checking whether constraints of the forms $X \leqslant t$ and $t \leqslant X$ can be derived from a constraint system.

1.1 Subtype systems for lambda calculus

This section, which provides background and motivation, gives subtyping rules for lambda calculus and discusses their properties. The rest of this paper does not assume detailed knowledge of lambda calculus.

Mitchell [13] defines subtyping for lambda-calculus by adding the *subsumption rule* to the standard rules for the simply typed lambda calculus.

$$\Gamma \vdash x : t \qquad \text{if } (x \mapsto t) \in \Gamma$$

$$\frac{\Gamma[x \mapsto t] \vdash M : u}{\Gamma \vdash \lambda x.M : t \to u}$$

$$\frac{\Gamma \vdash M : t \to u \qquad \Gamma \vdash N : t}{\Gamma \vdash MN : u}$$

$$\frac{\Gamma \vdash M : t \qquad t \leqslant u}{\Gamma \vdash M : u} \qquad \text{(subsumption)}$$

In his definition of subtyping, Mitchell assumes that the set of types is closed under the function type constructor $\to$, that the subtyping relation $\leqslant$ is transitive and reflexive, and that it always holds that whenever $t \leqslant t'$ and $u \leqslant u'$ we have $(t' \to u) \leqslant (t \to u')$.

Mitchell gave a *subject reduction lemma* [13] for lambda calculus with subtyping. This lemma implies that whenever a lambda term M is typable under Mitchell's subtype rules any lambda term that M might reduce to would also be typable. In Milner's words, well-typed programs cannot "go wrong" [12]. The definition of subtyping constraints given in this paper satisfies the properties listed in Mitchell's definition, but also requires that

whenever $(t \to u) \leqslant (t' \to u')$ it also holds that $t' \leqslant t$ and $u \leqslant u'$. It should be stressed that as Mitchell's subject reduction lemma does not make any assumptions on evaluation strategy, the lemma covers both lazy evaluation and eager evaluation.

As Palsberg and O'Keefe point out [15], Mitchell's subtyping rules allow for any typing an equivalent typing where every other rule is an application of the subsumption rule (see [10] for a detailed proof). Thus, given a lambda term and its corresponding constraint system (which can be determined by a linear traversal of the lambda term) the properties of the constraint system determine whether the lambda term is typable. It follows that if one is interested in, for example, algorithms for checking whether a lambda term can be typed it is sufficient to consider the properties of constraint systems.

Checking whether a lambda term is typable is equivalent to showing that the corresponding constraint system has a model. This can be done (a) directly, using an explicit construction based on some universe derived from the constraint language, or (b) by showing that the corresponding constraint system is consistent, which by Gödel's completeness theorem guarantees that the constraint system has a model. The former approach might sometimes fail to construct a model for a consistent constraint system, for example if the universe is restricted to finite types but the constraint system contains recursive equations which only have infinite solutions. Other authors use approach (a), i.e., attempt to construct a model using a given universe. This paper considers approach (b), i.e., the problem of checking consistency of the constraint system. In the context of type checking, this approach has, to the best of my knowledge, not been attempted before.

We will consider two problems over constraint systems for subtyping systems. First, *consistency*, i.e., whether a constraint system is logically consistent, and second, *derivability*, i.e., whether a particular constraint can be derived from a constraint system. The first property corresponds to checking that a program is typable and the second that (for example) a variable always has a certain type.

In contrast, other authors have looked at *solvability*; whether the constraint system have a solution in a given domain, and *entailment* of constraints; whether a constraint holds in any solution of the constraint system.

A solvable constraint system (in any domain) is always consistent, but there are constraint systems that are not solvable (however, as discussed above it is always possible to construct a model for any consistent theory). Similarly but conversely, if a constraint can be derived from a constraint system it is also entailed, but there may be constraints that are entailed but still not derivable. Thus, considering consistency instead of solvability will cause more programs to be accepted. However, thanks to Mitchell's subject

reduction lemma the usual safety guarantees still apply.

The constraint system we consider is fairly standard, with a set of atomic types under a partial order and a function type.

The rest of this paper is organised as follows. Section 2 defines type expressions and constraints. Section 3 gives examples of lambda terms and their corresponding constraint systems. Section 4 defines derivation rules for constraints. It also defines how, given a constraint system G, G^* is computed. Section 6 shows that if G^* does not contain any inconsistent constraint, G must be consistent. Section 7 shows how derivability can be efficiently determined for certain classes of constraints. Section 8 gives examples of the typing of lambda terms using the algorithm presented. Section 9 discusses related work and Section 10 makes some concluding remarks.

2 Constraints

Let $X, Y \in \mathsf{TVar}$ be the set of *type variables*. (We assume that the set of type variables is infinite.) We also assume a set of *atomic types*, $A, B \in \mathsf{Atom}$, related under some partial order $\leqslant_A$. Let the set of *type expressions* $t, u, v, w \in \mathsf{TExp}$ be defined as follows:

1. $\mathsf{TVar} \subseteq \mathsf{TExp}$,
2. $A \in \mathsf{TExp}$, if a is an atomic type.
3. $(t_1 \to t_2) \in \mathsf{TExp}$, if $t_1, t_2 \in \mathsf{TExp}$,

For type expressions t and u we write $t = u$ to indicate that the two expressions are the same.

Let the set of *constraints* $\varphi \in \mathcal{C}$ be formulas of the following forms (where $\perp$ is the inconsistent constraint):

1. $t_1 \leqslant t_2$, for $t_1, t_2 \in \mathsf{TExp}$
2. $\perp$

A *constraint system* G is a set of constraints. We will always assume that G is finite. We also assume that for any constraint system G, the constraint $A \leqslant B$ is present in G whenever A and B occur in G and $A \leqslant_A B$

We define a type expression t to be a *subexpression* of another type expression u if either 1) the two expressions are the same, or 2) $u = (u_1 \to u_2)$ and t is a subexpression of u_1 or u_2. When we talk about the expressions of a constraint $t \leqslant u$, all subexpressions of t and u are intended. In the same

way, the expressions of a constraint system G include all expressions of any constraint of G.

It should be noted that we place no restrictions on the constraint systems. Thus we allow, for example, recursive constraints such as

$$(A \to X) \leqslant X.$$

Note that while the constraint language does not contain an operator for defining unions, it is still possible to specify unions implicitly. For example, assuming that the values of X and Y are given by other constraints, it is possible to introduce a variable Z such that

$$X \leqslant Z \qquad \text{and} \qquad Y \leqslant Z.$$

In other words, Z may contain any value that is contained in either X or Y.

3 Examples

We give examples of lambda terms and their corresponding constraint systems.

As a first example, consider the typing of the identity function.

$$\dfrac{\dfrac{\dfrac{[x \mapsto X] \vdash x : X \qquad X \leqslant Y}{[x \mapsto X] \vdash x : Y}}{\vdash \lambda x.x : X \to Y} \qquad (X \to Y) \leqslant Z}{\vdash \lambda x.x : Z}$$

Thus, the type of the identity function is Z, for any X and Y such that $X \leqslant Y$ and $(X \to Y) \leqslant Z$. It is easy to establish that $\vdash \lambda x.x : Z$ is satisfied exactly when there is a type W such that $(W \to W) \leqslant Z$.

The identity function can of course be typed in any type system for lambda calculus. Now consider the lambda term $\lambda f.ff$. This term will produce a consistent constraint system, as seen in Figure 1. The corresponding constraint system is $\{Y \leqslant (Z \to X), Y \leqslant Z, (W \to W) \leqslant Y\}$. A solution for this constraint system can be obtained in any type system that allows solutions to recursive constraints by letting Y be a solution to $Y = (Y \to Y)$ and $X = Y = Z = W$. (Many subtyping systems presented in the literature only allow types that can be expressed as finite type expressions and will reject this constraint system.)

Now let us consider a lambda-term that will be rejected by our type system. Suppose A and B are distinct, unrelated, atomic types, and let Γ be an environment containing the bindings $f : A \to A$ and $x : B$. (One

$$\dfrac{[x \mapsto W] \vdash x : W \qquad (W \to W) \leqslant Y}{\lambda x.x : Y}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[f \mapsto Y] \vdash f : Y \qquad Y \leqslant (Z \to X)}{[f \mapsto Y] \vdash f : Z \to X} \qquad \dfrac{[f \mapsto Y] \vdash f : Y \qquad Y \leqslant Z}{[f \mapsto Y] \vdash f : Z}}{[f \mapsto Y] \vdash ff : X}}{\vdash \lambda f.ff : Y \to X} \qquad \lambda x.x : Y}{\vdash (\lambda f.ff)(\lambda x.x) : X}$$

Figure 1: A lambda-term with a recursive type. The typing of the identity function is simplified.

$$\dfrac{\dfrac{\Gamma \vdash f : A \to A \qquad (A \to A) \leqslant (X \to Y)}{\Gamma \vdash f : X \to Y} \qquad \dfrac{\Gamma \vdash x : B \qquad B \leqslant X}{\Gamma \vdash x : X}}{\Gamma \vdash fx : Y}$$

Figure 2: A lambda-term where the corresponding constraint system is inconsistent.

might perhaps imagine that f is a function that takes integers and x is a string.) Attempting to type the application fx gives us the derivation shown in Figure 2. The corresponding constraint system is

$$\{(A \to A) \leqslant (X \to Y), B \leqslant X\}.$$

It is easy to see that this constraint system is inconsistent. The inequality $(A \to A) \leqslant (X \to Y)$ implies that $X \leqslant A$ and $A \leqslant Y$. From $X \leqslant A$ and $B \leqslant X$ follows $B \leqslant A$, which is an inconsistent constraint.

4 Derivations

Given a constraint system G and a constraint φ we say $G \vdash \varphi$ if $\varphi \in G$ can be derived from G using the rules of Figure 3. We say that a constraint system G is *inconsistent* if $G \vdash \bot$, and that it is *consistent* if $\bot$ is not derivable. We also say that a constraint φ is *inconsistent* if $\{\varphi\} \vdash \bot$.

We view a derivation $G \vdash \varphi$ as a (finite) tree where each node is labelled with a constraint, the root is φ, each node is either due to a derivation rule or the constraint system. If a node labelled ψ is due to a derivation rule it

has children labelled $\psi_1, \ldots, \psi_n$ and corresponds to an instance

$$\frac{\psi_1, \ldots, \psi_n}{\psi}$$

of some derivation rule. If φ is given by the constraint system we have $\varphi \in G$ (and the node has no children).

4.1 Notes on the derivation rules

The derivation rules of Figure 3 state that the inequality relation is reflexive (R) and transitive (T). Further, the function type constructor $\rightarrow$ is monotone in its second argument and anti-monotone in its second (C). These rules can be found in many formalisations of subtyping for lambda-calculus, for example [13].

The fourth and fifth rules (WL and WR) state that if two function types are related then the arguments to the function type constructor are related. Barendregt et al. say that a type structure is *invertible* if it satisfies this property [3]. Most work on subtyping systems for lambda calculus define the types and the subtyping relation inductively. In that case, it follows immediately that the type structure is invertible.

The following three rules state that atoms are not related to function types (AW and WA) and that atoms that are not related under the partial order are not related under subtyping (AA).

Finally, we include the standard logic rule that any constraint can be derived from the inconsistent constraint (F).

5 Construction of G^*

The construction of G^* that follows plays a central role in this paper. As is discussed below, for any constraint system G, G^* can always be computed in polynomial time. It is shown in Theorem 6.4 of next section that G is consistent iff every constraint in G^* is consistent, thus consistency of G can be determined by computing G^*. For many constraints, derivability can also be determined by inspecting G^*. The construction of G^* was influenced by a similar definition in [15].

Definition 5.1. *Given a constraint system G, define $(G)_n$, for $n \geq 0$, to be the smallest sets that satisfy the following:*

1. $(G)_0 = G$,

$$\frac{}{t \leqslant t} \tag{R}$$

$$\frac{t \leqslant u, \qquad u \leqslant v}{t \leqslant v} \tag{T}$$

$$\frac{t \leqslant t', \qquad u \leqslant u'}{(t' \to u) \leqslant (t \to u')} \tag{C}$$

$$\frac{(t \to t') \leqslant (u \to u')}{u \leqslant t} \tag{WL}$$

$$\frac{(t \to t') \leqslant (u \to u')}{t' \leqslant u'} \tag{WR}$$

$$\frac{A \leqslant (t \to u)}{\bot} \tag{AW}$$

$$\frac{(t \to u) \leqslant A}{\bot} \tag{WA}$$

$$\frac{A \leqslant B \qquad A \not\leqslant_A B}{\bot} \tag{AA}$$

$$\frac{\bot}{\varphi} \tag{F}$$

Figure 3: Derivation rules for constraints. The rules define the relation $\vdash$.

2. *for all* n, $G_{n+1} \supseteq G_n$,

3. *for all even* $n > 0$, *if* $(G)_{n-1}$ *contains the constraint*

$$(t \to u) \leqslant (t' \to u'),$$

then $(G)_n$ *contains the constraints* $t' \leqslant t$ *and* $u \leqslant u'$, *and*

4. *for all odd* $n > 0$, *if* $(G)_{n-1}$ *contains the constraints* $t \leqslant X$ *and* $X \leqslant u$ *then* $(t \leqslant u) \in (G)_n$.

Let $G^* = \bigcup_n (G)_n$.

The complexity of constructing G^* can be determined by an argument similar to one used by Heintze [5]. First, note that G^* only contains type expressions present in G. Thus if the size of G is n, and G contains no more than n expressions, there are less than n^2 inequalities in G^*, which sets a bound to the space used by the construction. When an inequality $t \leqslant u$ is added to the constraint system, the algorithm must check for the presence of inequalities of the forms $t' \leqslant t$ and $u \leqslant u'$ (in the *odd* step). This may, at worst, require work proportional to the number of expressions in G, thus the cost of adding a constraint is $O(n)$ and the worst-case complexity of the algorithm is $O(n^3)$.

Proposition 5.2. *Let* G *be some constraint system.* $G^* = (G)_n$, *some* n.

Proof. The set of expressions in G is finite. The constraint system $(G)_i$, any i, only contains expressions that are present in G. Thus there are only a finite number of distinct constraint systems $(G)_i$. □

Definition 5.3. *We say that a constraint is* immediately inconsistent *if it is of one of the forms* $A \leqslant (t \to u)$, $(t \to u) \leqslant A$, *or* $A \leqslant A'$, *where* $A \not\leqslant_A A'$.

A constraint system G *is* locally consistent *iff* G^* *does not contain any immediately inconsistent constraints.*

Note that an immediately inconsistent constraint is always inconsistent, but the converse does not always hold. Checking whether a constraint is immediately inconsistent can be done in constant time, thus checking whether a constraint system is locally consistent can be done in linear time.

We will show in Theorem 6.4 that a constraint system is locally consistent exactly when it is consistent. The following propositions concern chains of constraints.

Proposition 5.4. *Let G be a constraint system containing constraints*

$$t_1 \leqslant t_2, t_2 \leqslant t_3, \ldots, t_{m-1} \leqslant t_m.$$

For some n, there are constraints

$$u_1 \leqslant u_2, u_2 \leqslant u_3, \ldots, u_{k-1} \leqslant u_k$$

in $(G)_n$ such that $u_1 = t_1$, $u_k = t_m$ and none of the type expressions $u_2, \ldots, u_{k-1}$ is a variable.

(The proof follows from the construction of (G).)

Proposition 5.5. *Given a constraint system G and type expressions $t_1, \ldots, t_n$. such that $(t_i \leqslant t_{i+1}) \in G^*$ all $i < n$. Suppose the constraint $t_1 \leqslant t_n$ is immediately inconsistent. There is some immediately inconsistent constraint in G^*, thus G is not locally consistent.*

Proof. Since the constraint $t_1 \leqslant t_n$ is immediately inconsistent neither t_1 nor t_n is a type variable. By Proposition 5.4 we can assume that the type expressions $t_2, \ldots, t_{n-1}$ are not type variables, thus none of the type expressions is a type variable.

In the following, we'll assume that all constraints $t_i \leqslant t_{i+1}$ are not immediately inconsistent and show that when this holds $t_1 \leqslant t_n$ is not immediately inconsistent, thus deriving a contradiction.

Suppose t_1 is an atomic type. It follows by the assumption that the chain is not immediately inconsistent that the type expressions $t_2, \ldots, t_n$ are also atoms, and ordered such that $t_i \leqslant_A t_j$ whenever $i < j$. By transitivity of the $\leqslant_A$ ordering we have $t_1 \leqslant_A t_n$ which implies that this constraint is not immediately inconsistent.

If t_1 is a function type $u_1 \to v_1$ it follows that each t_i is some function type. Thus both t_1 and t_n are function types and the constraint $t_1 \leqslant t_n$ is not immediately inconsistent. □

To make reasoning about derivations more straight-forward, we introduce a second format for derivations.

Definition 5.6. *For a constraint system G and a constraint φ write $G \vdash_{RTC} \varphi$ if φ can be derived from G via a single use of one of the rules R, T or C.*

For constraint systems $G_1, \ldots, G_n$ and a constraint φ, write $G_1, \ldots, G_n \Rightarrow \varphi$ if there are constraints $\psi_1, \ldots, \psi_{n-1}$ such that

1. *for all $i < n$, $G_i^* \vdash_{RTC} \psi_i$,*

2. *for all* $i < n$, $G_{i+1} = G_i \cup \{\psi_i\}$, *and*

3. $\varphi \in G_n^*$.

Proposition 5.7. *Suppose* $G_1, \ldots, G_n \Rightarrow \varphi$ *and* $G'_1, \ldots, G'_m \Rightarrow \varphi'$. *It follows that there are some constraint systems* $H_1, \ldots, H_k$ *such that*

1. $H_1 = G_1 \cup G'_1$,

2. $H_1, \ldots, H_k \Rightarrow \varphi$ *and*

3. $H_1, \ldots, H_k \Rightarrow \varphi'$.

Proof. We will only give the construction of the constraints systems $H_1, \ldots, H_k$. The proof that this sequence of constraint systems satisfies the desired properties is straight-forward.

Let $H_1, \ldots, H_k$ be defined as follows.

$$\begin{aligned} k &= n + m - 1 \\ H_i &= G_i \cup G'_1, \text{for } i \leq n \\ H_{n+j-1} &= G_n \cup G'_j, \text{for } j \leq m \end{aligned}$$

□

Lemma 5.8. *Suppose that* G *is a constraint system, and that there is a derivation* $G \vdash \varphi$ *that does not mention* $\perp$. *There are constraint systems* $G_1, \ldots, G_n$ *such that* $G_1 = G$ *and* $G_1, \ldots, G_n \Rightarrow \varphi$.

Proof. The proof is by induction on the size of the derivation. If $\varphi \in G$ we have immediately that $G \Rightarrow \varphi$ as $\varphi \in G^*$.

Suppose the last rule of the derivation is

$$\frac{\psi_1 \ldots \psi_m}{\psi}.$$

By the induction hypothesis there are for each $i \leq m$ some constraint systems $G_{1i}, \ldots, G_{n_i i}$ such that $G_{1i}, \ldots, G_{n_i i} \Rightarrow \psi_i$ and $G_{1i} = G$, all i. Thus by Proposition 5.7 there are $G_1, \ldots, G_k$ such that $G_1 = G_{1i}$ and $G_1, \ldots, G_k \Rightarrow \psi_i$, all $i \leq m$.

If the last rule of the derivation is one of WL or WR it follows immediately that $\varphi \in G_k^*$ and thus $G_1, \ldots, G_k \Rightarrow \varphi$.

If the last rule is R, C, or T we have $G_k^* \vdash_{\text{RTC}} \varphi$ and thus with $G_{k+1} = G_k \cup \{\varphi\}$ we have $G_1, \ldots, G_k, G_{k+1} \Rightarrow \varphi$.

The last rule cannot be one of AW, WA, AA, or F because of the assumption that the derivation does not mention $\perp$. □

So, for any derivation $G_1 \vdash \phi$ (that does not involve $\perp$) we can find constraint systems $G_2, \ldots, G_n$ such that $G_1, \ldots, G_n \Rightarrow \varphi$. Conversely, if the latter holds we can of course always construct a derivation.

6 Consistency

We first establish some properties of the R, T and C rules.

Proposition 6.1 (Rule R)**.** *Suppose that G is a constraint system, t some type expression and φ an inequality. Let $H = G \cup \{t \leqslant t\}$.*

Whenever $\varphi \in (H)_n$ it holds either that

1. *$\varphi \in (G)_n$, or*
2. *$\varphi = (u \leqslant u)$, some subexpression u of t.*

Proof. By induction on n. The case when $n = 0$ follows immediately.

Suppose that $n > 0$ and the proposition holds for $n - 1$ and that $(u \leqslant u') \in (H)_n$ but $(u \leqslant u') \notin (H)_{n-1}$.

If n is even there is some constraint of either the form $(\ldots \to u) \leqslant (\ldots \to u')$ or $(u' \to \ldots) \leqslant (u \to \ldots)$ in $(H)_{n-1}$. We will only consider the first case as the two are symmetric. By the induction hypothesis either $(\ldots \to u) \leqslant (\ldots \to u') \in (G)_{n-1}$ or $(\ldots \to u)$ is a subexpression of t. It is easy to complete the proof in either case.

If n is odd there are constraints $u \leqslant X$ and $X \leqslant u'$ in $(H)_{n-1}$, for some type variable X. By the induction hypothesis there are three possibilities. If $(G)_{n-1}$ contains $u \leqslant X$ and $X \leqslant u'$ the proposition follows immediately. If $u = X$, u is a subexpression of t, and $X \leqslant u'$ in $(G)_{n-1}$ it follows immediately that $u \leqslant u'$ is contained in $(G)_n$. The third case, with $X = u'$, is similar. □

Proposition 6.2 (Rule T)**.** *Suppose that G is locally consistent and contains the constraints $t \leqslant t'$ and $t' \leqslant t''$. Let $H = G \cup \{t \leqslant t''\}$.*

Whenever a constraint $u \leqslant v$ occurs in $(H)_n$, there are type expressions $w_1, w_2, \ldots, w_m$ and an integer k such that $w_1 = u$, $w_m = v$, and the constraint $w_i \leqslant w_{i+1}$ occurs in $(G)_k$, for $i < m$.

Proof. The proof is by induction on n. For $n = 0$, the proof is immediate.

Suppose that $n > 0$ and the constraint $u \leqslant v$ occurs in $(H)_n$ but not in $(H)_{n-1}$. If n is even we have that a constraint of one of the forms $(\ldots \to u) \leqslant (\ldots \to v)$ or $(v \to \ldots) \leqslant (u \to \ldots)$ occurs in $(H)_{n-1}$. We will only consider the first case as the two cases are symmetric. By the induction hypothesis there is a k such that there are constraints $w_i \leqslant w_{i+1}$ occurring in $(G)_k$, for

$i < m$, such that $w_1 = (\ldots \to u)$ and $w_m = (\ldots \to v)$. By Proposition 5.4 we can assume that none of the intermediate type expressions is a variable. From the assumption that G is locally consistent follows that all w_i are function type expressions, thus there are w_i'' and w_i' such that $w_i = (w_i'' \to w_i')$, for $i \leq m$. By the construction of (G) we have $w_i' \leqslant w_{i+1}'$ in $(G)_{k+2}$ all $i < m$, with $w_1' = u$ and $w_m' = v$.

If n is odd there is a type variable X such that that the constraints $u \leqslant X$ and $X \leqslant v$ occur in $(H)_{n-1}$. By the induction hypothesis there are type expressions $u_1, \ldots, u_m, v_1, \ldots, v_l$ such that $u_1 = u$, $u_m = X$, $v_1 = X$, and $v_l = v$, and the constraints $u_i \leqslant u_{i+1}$ and $v_j \leqslant v_{j+1}$ occur in $(G)_k$, for $i < m$ and $j < l$. Since $u_m = v_1$ this forms the required chain of constraints. □

Proposition 6.3 (Rule C)**.** *Let G be a constraint system containing the constraints $t \leqslant t'$ and $u \leqslant u'$. Let $\varphi = ((t' \to u) \leqslant (t \to u'))$ and $H = G \cup \{\varphi\}$. It follows that whenever a constraint ψ occurs in $(H)_n$, we have either*

1. *ψ in $(G)_n$, or*
2. *$\psi = \varphi$.*

Proof. The proof is by induction on n. For $n = 0$, immediate.

Suppose that $n > 0$ and the constraint $v \leqslant w$ occurs in $(H)_n$ but not in $(H)_{n-1}$.

If n is even, there is a constraint of one of the forms $(\ldots \to v) \leqslant (\ldots \to w)$ or $(w \to \ldots) \leqslant (v \to \ldots)$ in $(H)_{n-1}$. We will only consider the first case as the two cases are symmetric. By the induction hypothesis, we have either that $(\ldots \to v) \leqslant (\ldots \to w)$ occurs in $(G)_{n-1}$, or the constraint $(\ldots \to v) \leqslant (\ldots \to w)$ is φ. In the first case, it follows by the construction of (G) that $v \leqslant w$ occurs in $(G)_n$. In the second case, the proposition also follows from the construction of (G).

If n is odd, there is a type variable X such that the constraints $v \leqslant X$ and $X \leqslant w$ occur in $(H)_{n-1}$. By the induction hypothesis, the two constraints also occur in $(G)_{n-1}$, and by the construction of (G), we have $v \leqslant w$ in $(G)_n$. □

We are now ready to state one of the main results of this paper.

Theorem 6.4. *A constraint system G is consistent iff G is locally consistent.*

Proof. If G is *not* locally consistent it follows immediately that G is not consistent. We consider the converse and show that if G is inconsistent there must be an immediately inconsistent constraint in G^*. It follows by the

derivation rules for constraints that there is some immediately inconsistent inequality $t \leqslant u$ such that $G \vdash t \leqslant u$, and by Lemma 5.8 constraint systems $G_1, \ldots, G_n$ such that $G_1 = G$ and $G_1, \ldots, G_n \Rightarrow t \leqslant u$.

We will show by induction on n that whenever $G_1, \ldots, G_n \Rightarrow t \leqslant u$, some immediately inconsistent constraint $t \leqslant u$, G_1 is *not* locally consistent.

If $n = 1$ the proof follows immediately.

Suppose $n > 1$ and the proof holds for $n - 1$. If the last RTC-rule is R it follows by Proposition 6.1 that $t \leqslant u \in G^*_{n-1}$ and the induction hypothesis applies.

If the last RTC-rule is T there are constraints $w \leqslant w'$ and $w' \leqslant w''$ in G^*_{n-1} such that $G_n = G_{n-1} \cup \{w \leqslant w''\}$. Suppose G^*_n contains an immediately inconsistent constraint $t \leqslant u$. Since the constraint is immediately inconsistent, it follows that neither t nor u is a type variable. By Proposition 6.2, there is an m such that there is a chain of constraints $v_1 \leqslant v_2, v_2 \leqslant v_3, \ldots, v_{m-1} \leqslant v_m$ in G^*_{n-1}, with $v_1 = t$ and $v_m = u$. It follows by Proposition 5.5 that there is a constraint in G^* which is not immediately inconsistent, thus G_{n-1} is not locally consistent.

If the last RTC-rule is C there are type expressions v, v', w and w' such that $(v \leqslant v'), (w \leqslant w') \in G^*_{n-1}$ and $G_n = G_{n-1} \cup \{(v' \to w) \leqslant (v \to w')\}$. Suppose G^*_n contains an inconsistent constraint $t \leqslant u$. If both t and u are function types; $t = (t_1 \to t_2)$ and $u = (u_1 \to u_2)$, G^*_n must contain the constraints $u_1 \leqslant t_1$ and $t_2 \leqslant u_2$. At least one of the two constraints must be inconsistent, and since we can repeat this step it follows that there is an inconsistent constraint $t' \leqslant u'$ in G^*_n such that not both t' and u' are function types. We can apply the first case of Proposition 6.3 and find that $(t' \leqslant u') \in G^*_{n-1}$ and by the induction argument G_1 is not locally consistent. □

7 Derivability

As shown in the previous section, a constraint system is consistent iff it is locally consistent, i.e., $G \vdash \bot$ iff $\bot \in G^*$. In this section we will consider an algorithm for determining whether a constraint φ is derivable.

Proposition 7.1. *If G is a consistent constraint system and A and B are atomic types such that $G \vdash A \leqslant B$, then $A \leqslant_A B$.*

Proof. Suppose $A \not\leqslant_A B$. It follows by rule AA that the constraint system G is not consistent. □

Proposition 7.2. *Suppose G is consistent, X a type variable and A an atom.*

1. *If $G \vdash X \leqslant A$ then there is some atomic type A' such that $(X \leqslant A') \in G^*$ and $A' \leqslant_A A$.*

2. *If $G \vdash A \leqslant X$ then there is some atomic type A' such that $(A' \leqslant X) \in G^*$ and $A \leqslant_A A'$.*

Proof. We will only show the first half of the proposition as the second part is similar to the first. By Lemma 5.8 there are $G_1, \ldots, G_n$ such that $G_1 = G$ and $G_1, \ldots, G_n \Rightarrow X \leqslant A$. The proof is by induction on n. If $n = 1$ the proposition follows immediately. Suppose the proposition holds for $n - 1$. If the last RTC-rule is R the proposition follows from Proposition 6.1.

If the last RTC-rule is T we have some $t_1, \ldots, t_m$ such that $t_1 = X$, $t_m = A$ and $G_1, \ldots, G_{n-1} \Rightarrow (t_i \leqslant t_{i+1})$ all $i < m$. By Proposition 5.4 we can assume that t_2 is not a variable. Since G is assumed to be consistent it follows that v_2 is some atom A''. By Proposition 7.1 we have $A'' \leqslant_A A$ and by the induction hypothesis there is an atomic type A' such that $(X \leqslant A') \in G^*$ and $A' \leqslant_A A''$. Since the relation $\leqslant_A$ is transitive we are done.

If the last RTC-rule is C we have by Proposition 6.3 that $(X \leqslant A) \in G^*$. □

Lemma 7.3. *Suppose that G is a consistent constraint system.*

1. *If $G \vdash X \leqslant (t \to u)$, there are type variables t' and u' such that $G \vdash t \leqslant t'$, $G \vdash u' \leqslant u$ and $X \leqslant (t' \to u')$ in G^*.*

2. *If $G \vdash (t \to u) \leqslant X$, there are type variables t' and u' such that $G \vdash t' \leqslant t$, $G \vdash u \leqslant u'$ and $(t' \to u') \leqslant X$ in G^*.*

Proof. We will only consider the first part of the lemma. By Lemma 5.8 there are constraint systems $G_1, \ldots, G_n$ such that $G_1 = G$ and

$$G_1, \ldots, G_n \Rightarrow X \leqslant (t \to u).$$

The proof is by induction on n. If $n = 1$ we have immediately that $X \leqslant (t \to u)$ in G^*.

Suppose that $n > 1$. If the last RTC-rule is R, the lemma follows by Proposition 6.1 and the induction hypothesis.

If the last RTC-rule is T, there are by Proposition 6.2 type expressions $v_1, \ldots, v_m$ such that $(v_i \leqslant v_{i+1}) \in G^*_{n-1}$, all $i < m$, where $v_1 = X$ and $v_m = (t \to u)$. By Proposition 5.4 we can assume that v_2 is not a variable. Since G is assumed to be consistent it follows that $v_2 = (t' \to u')$, some t' and u'. Since $G \vdash v_2 \leqslant (t \to u)$ it also holds that $G \vdash t \leqslant t'$ and $G \vdash u' \leqslant u$. We have $G_1, \ldots, G_{n-1} \Rightarrow X \leqslant (t' \to u')$ which allows us to apply the induction hypothesis.

Suppose the last RTC-rule is C. By Proposition 6.3 the induction hypothesis can be applied. □

Say that a constraint system G is in *normal form* if every constraint in G is of the form $t \leqslant t'$ where t and t' is either an atomic type, a variable, or of the form $X \to Y$ where X and Y are type variables. Any constraint system can be converted to one in normal form by introducing extra variables. Also note that when G is in normal form so is G^*.

The definition below describes how, given a constraint system G and a variable-free type expression t, it is possible to determine the variables X for which $G \vdash t \leqslant X$ and $G \vdash X \leqslant t$. In the construction, $X \in L_i$ iff $G \vdash X \leqslant t_i$ and $X \in U_i$ iff $G \vdash t_i \leqslant X$.

Definition 7.4. *Let G be a consistent constraint system in normal form and t a variable-free type expression. Let $\{t_1, \ldots, t_n\}$ be the subexpressions of t ordered so that any proper subexpression of the type expression t_i occurs before t_i.*

For $i \leq n$, define the sets L_i and U_i as follows:

1. *If t_i is an atom, let*

$$\begin{aligned} L_i = \{Z \mid\ & A \leqslant_A t_i, (Z \leqslant A) \in G^*, \\ & A \text{ is an atomic type}\}, \\ U_i = \{W \mid\ & t_i \leqslant_A A, (A \leqslant W) \in G^*, \\ & A \text{ is an atomic type}\}. \end{aligned}$$

2. *If t_i is a function type $t_j \to t_k$, let*

$$\begin{aligned} L_i = \{Z \mid\ & (Z \leqslant (X \to Y)) \in G^*, \\ & X \in U_j, Y \in L_k\}, \\ U_i = \{W \mid\ & ((X \to Y) \leqslant W) \in G^*, \\ & X \in L_j, Y \in U_k\}. \end{aligned}$$

Theorem 7.5. *Suppose G is a consistent constraint system in normal form, X is a type variable, t is a variable-free type expression with subexpressions $\{t_1, \ldots, t_n\}$ in subexpression order, and for $i \leq n$ the sets L_i and U_i are defined as above. We have*

1. *$G \vdash X \leqslant t_i$ iff $X \in L_i$, and*

2. *$G \vdash t_i \leqslant X$ iff $X \in U_i$.*

Proof. The proof is by induction on i. If t_i has no proper subexpression it must be an atomic type and the theorem follows from Proposition 7.2

If t_i is a function type $t_j \to t_k$, we have $j, k < i$. Thus the sets L_j, L_k, U_j and U_k are already computed. Suppose that there is some type variable Y such that $G \vdash Y \leqslant t_i$. By Lemma 7.3 there are u, v such that $Y \leqslant (u \to v)$ in G^* and $G \vdash t_j \leqslant u$ and $G \vdash v \leqslant t_k$. Since G is in normal form G^* is also in normal form. This implies that u and v are variables. By the induction hypothesis we can determine $G \vdash t_j \leqslant u$ and $G \vdash v \leqslant t_k$ by examining the sets U_j and L_k. If (the variables) u and v are present in these sets and $Y \leqslant (u \to v)$ occurs in G^* we have $Y \in L_i$. □

Suppose that L_j and U_j has already been computed, for $j < i$, Assuming that the size of G is n and the size of the variable-free type expression t is m, the cost of computing L_i and U_i is proportional to the size of G^* which is $O(n^2)$. Thus the total worst-case cost of determining whether a constraint $X \leqslant t$ or $t \leqslant X$ is derivable is $O(n^3 + n^2 m)$. If we make the reasonable assumption that the type expression t is smaller than the constraint system G the cost is $O(n^3)$.

8 Examples: Determination of consistency

In this section we will consider the typing of a number of lambda terms using the computation of G^*, as described in Section 4. We begin with the examples of Section 3.

Consider the identity function $\lambda x.x$ which has the the corresponding constraint system

$$\begin{aligned} G = \{ & X \leqslant Y, \\ & (X \to Y) \leqslant Z\} \end{aligned}$$

It follows immediately that $G^* = G$ and thus the constraint system is consistent.

As discussed in Section 3 the lambda-term

$$(\lambda f.ff)(\lambda x.x)$$

has the type X, given the constraint system

$$\begin{aligned} G = \{ & Y \leqslant (Z \to X), \\ & Y \leqslant Z, \\ & (W \to W) \leqslant Y\} \end{aligned}$$

$$\dfrac{\Gamma \vdash \lambda x.x : X \to X \qquad \dfrac{\Gamma \vdash f : A \to A \qquad (A \to A) \leqslant X}{\Gamma \vdash f : X}}{\Gamma \vdash (\lambda x.x)f : X}$$

$$\dfrac{\dfrac{\Gamma \vdash (\lambda x.x)f : X \qquad X \leqslant (Y \to Z)}{\Gamma \vdash (\lambda x.x)f : Y \to Z} \qquad \dfrac{\Gamma \vdash y : B \qquad B \leqslant Y}{\Gamma \vdash y : Y}}{\Gamma \vdash ((\lambda x.x)f)y : Z}$$

Figure 4: A slightly more complex lambda-term where the corresponding constraint is inconsistent.

Here

$$\begin{aligned} G^* = G \cup \{ & (W \to W) \leqslant (Z \to X), \\ & (W \to W) \leqslant Z, \\ & Z \leqslant W, \\ & W \leqslant X, \\ & (W \to W) \leqslant W, \\ & (W \to W) \leqslant X, \\ & Z \leqslant X, \\ & Y \leqslant W, \\ & Y \leqslant X\} \end{aligned}$$

which is locally consistent, thus G must be consistent.

Section 3 also gave an example of a lambda-term fx which would not type, as f is a function over the atomic type A and x a variable of atomic type B, and the two atomic types are not related.

$$\begin{aligned} G = \{ & (A \to A) \leqslant (X \to Y), \\ & B \leqslant X\} \end{aligned}$$

$$\begin{aligned} G^* = G \cup \{ & X \leqslant A, \\ & A \leqslant Y, \\ & B \leqslant A\} \end{aligned}$$

Here G^* is not locally consistent as it contains the inconsistent constraint $B \leqslant A$, thus G is not consistent.

In Figure 4 we consider the derivation of the type of a slightly more complex lambda-term that can not be typed: $((\lambda x.x)f)y$, similar to the previous example, but f is passed to the identity function before being applied to its argument. (The derivation of the type of the identity function is not shown.) We assume that Γ assigns f the type $A \to A$ and y the type B, where A and

B are distinct, unrelated, atomic types.

$$\begin{aligned} G = \{&(A \to A) \leqslant X, \\ &X \leqslant (Y \to Z), \\ &B \leqslant Y\} \end{aligned}$$

and

$$\begin{aligned} G^* = G \cup \{&(A \to A) \leqslant (Y \to Z) \\ &Y \leqslant A, \\ &A \leqslant Z, \\ &B \leqslant A\} \end{aligned}$$

Again, G^* contains an inconsistent constraint $B \leqslant A$. Thus G^* is not locally consistent, G is inconsistent and the term is rejected.

9 Related work

Many authors have focused on the problem of *solving* a given constraint system, i.e., constructing a model for a constraint system using a given domain. The approach assumes a domain consisting of expressions in the language (if restricted to finite trees, this corresponds to the term algebra, but sometimes the domain is extended to infinite trees) and attempts to construct a model in this domain. If a model can can be constructed using this domain, it follows immediately that the constraint system is consistent. However, the converse does not hold in general; there are constraint languages which allow consistent constraint systems where a model cannot be constructed in this manner. Another issue that has received attention is that of *entailment* of constraints; given a constraint system and some constraint, is the constraint satisfied in any solution (as above) to the constraint system?

Aiken and Wimmers [1] consider a system with union and intersection and solutions to recursive equations but no function types. Their algorithm computes a closure over the constraint system, similar to the construction of G^* in Section 4. Their algorithm will discover inconsistencies in the constraint system, as in the current paper, and as it is shown that any constraint system that has not been discovered to be inconsistent is solvable it follows that the algorithm must recognise all inconsistent constraint systems. (Interestingly, this suggests that consistency and solvability coincide for this class of constraint systems.) They show that the problem solved by their algorithm is EXPTIME-hard. A similar system which also includes function types is presented in [2] but its complexity is not discussed.

Kozen, Palsberg and Schwartzbach [10] consider the problem of constructing a model with the term algebra as domain for a given constraint

language. They consider a constraint language with a universal type (but no empty type) and give an algorithm for checking that an expression in lambda-calculus can be typed in $O(n^3)$ time. They point out that the algorithm can also used for a type system that allows recursive types. For a given lambda term, their algorithm tries to construct a finite state automaton. If the construction succeeds, the expression can be typed under the subtyping rules. Palsberg, Wand and O'Keefe [16] describe a similar solution for a *non-structural subtyping system*, i.e., one which allows both an empty and a universal type. Palsberg and O'Keefe [15] give an algorithm based on flow analysis [9, 17, 5] for checking that a lambda term can be typed under a subtyping system.

Tiuryn [20] looks at a subtyping system without empty and universal types but with a set of atomic types related according to a partial order. In the general case the solvability problem is PSPACE-hard, but if the atomic types are organised in a disjoint set of lattices the problem can be determined in PTIME. Henglein and Rehof [7] consider the problem of entailment for a simple subtyping system. They show that the problem of determining entailment is in general coNP-complete. Niehren, Priesnitz and Su [14] consider the subtyping problem over arbitrary partial orders. By exploiting a relationship with propositional dynamic logic they show that in a system with recursive types the problem of structural subtype satisfiability is DEXPTIME-hard while the non-structural subtype satisfiability problem is DEXPTIME-complete.

Su et al. [18] consider a subtyping system with conjunction, negation and quantifiers and show that solvability is in general undecidable. In contrast, Kuncak and Rinard show that when restricted to non-recursive types, the problem is decidable [11].

It should be stressed that the results presented in this paper are not directly comparable to the ones summarised above. Since we consider the question of whether a constraint system is consistent a value may be present in a model of the constraint system even though there is no corresponding expression in the constraint language. Thus, a model may contain values corresponding to solutions of recursive definitions, unions, the empty type and the universal type even though they are not present in the type language. This also implies that the dichotomy between structural and non-structural typing does not apply when one considers consistency.

10 Conclusions

This paper presents a new approach to subtyping. Instead of modelling the types of a program using the elements present in the syntax, the subtyping system considers the consistency of the corresponding constraint system. If the constraint system is consistent the program is accepted. The result is a subtyping system which is more general in that it accepts programs that would not be accepted by many other subtyping systems but still offers the usual safety guarantees.

The worst-case complexity of the algorithm for consistency checking is $O(n^3)$. It remains to be seen whether this is efficient enough for use in a programming language implementation, but it is worth noting that scalable implementations exist for similar algorithms where the worst-case complexity is cubic and where the computational effort is dominated by the cost of computing the transitive closure of a graph where edges may be added during the computation [6, 4].

The paper presents an $O(n^3)$ method for checking derivability of constraints of the forms $X \leqslant t$ and $t \leqslant X$, where X is a type variable and t a type expression with no variables. This can be used, for example, to allow a compiler to query the type system for the types of variables in a program.

References

[1] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints. In *Logic in computer science*, pages 329–340, June 1992.

[2] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[3] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[4] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Programming Language Design and Implementation (PLDI)*, pages 290–299, 2007.

[5] Nevin Heintze. Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[6] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Programming Language Design and Implementation (PLDI)*, pages 254–263, 2001.

[7] Fritz Henglein and Jakob Rehof. The complexity of subtype entailment for simple types. In *In Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS*, pages 352–361. IEEE Computer Society Press, 1997.

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

[9] Neil D. Jones. Flow analysis of lambda expressions. In *Automata, Languages and Programming*, pages 114–128, 1981.

[10] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

[11] Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *Logic in computer science*, pages 96–107, June 2003.

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

[13] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[14] Joachim Niehren, Tim Priesnitz, and Zhendong Su. Complexity of subtype satisfiability over posets. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 357–373, 2005.

[15] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM transactions on programming languages and systems*, 17(4):576–599, July 1995.

[16] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal aspects of computing*, 9:49–67, 1997.

[17] Olin Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, 1988.

[18] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'02, pages 203–216, New York, NY, USA, 2002. ACM.

[19] Satish Thatte. Type inference with partial types. In Timo Lepistö and Arto Salomaa, editors, *Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 615–629. Springer Berlin Heidelberg, 1988.

[20] Jerzy Tiuryn. Subtype inequalities. In *Logic in Computer Science*, pages 308–315, 1992.